Extracting Executable Transformations from Distilled Code Changes

Reinout Stevens
Software Languages Lab
Vrije Universiteit Brussel, Belgium
resteven@vub.ac.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
cderoove@vub.ac.be

***Abstract*—Change distilling algorithms compute a sequence of fine-grained changes that, when executed in order, transform a given source AST into a given target AST. The resulting change sequences are used in the field of mining software repositories to study source code evolution. Unfortunately, detecting and specifying source code evolutions in such a change sequence is cumbersome. We therefore introduce a tool-supported approach that identifies minimal executable subsequences in a sequence of distilled changes that implement a particular evolution pattern, specified in terms of intermediate states of the AST that undergoes each change. This enables users to describe the effect of multiple changes, irrespective of their execution order, while ensuring that different change sequences that implement the same code evolution are recalled. Correspondingly, our evaluation is two-fold. Using examples, we demonstrate the expressiveness of specifying source code evolutions through intermediate ASTs. We also show that our approach is able to recall different implementation variants of the same source code evolution in open-source histories.**

I. Introduction

The use of a Version Control System (VCS) has become an industry best practice for developing software. Researchers in the field of mining software repositories (MSR) leverage the resulting revision histories to study the evolution of software systems. However, most VCSs record their revisions of the source code in a code-agnostic manner. Differences between two revisions are therefore only available from the VCS as lines of text that have been changed. A more fine-grained representation, e.g., in terms of source code constructs that have changed, is not readily available.

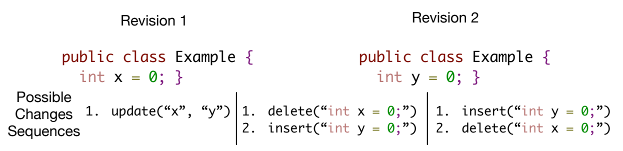
A change distilling algorithm (e.g., [1], [2], [3], [4]) can be used to obtain more fine-grained information about the differences between two revisions. Such an algorithm takes two Abstract Syntax Trees (ASTs) as input, called the source AST and the target AST respectively. It returns a sequence of change operations that, when applied *in order*, transforms the source AST into the target AST. A change is either an insert, a move, a delete or an update of an AST node. These changes provide fine-grained information about how the source code constructs might have changed. Analyzing or querying such change sequences is an essential ingredient in many a MSR study (e.g., [5], [6], [7], [8]).

In this paper we address a problem faced by many MSR researchers; the problem of recognizing in and extracting from the output of a change distiller, a *minimal* and *executable* edit script (i.e., a sequence of changes) that implements a specified transformation of interest. Executable means that applying the edit script on the initial source file results in edits corresponding to the specified transformation. Minimal means that removing any change from the script either preclude the script from being executed, or render the resulting transformation of the source file incomplete.

Manually recognizing a sought-after transformation in a distilled change sequence is challenging. Change sequences tend to be large, with every change potentially depending on the output of its predecessor. Changes that contribute to the transformation need to be isolated from the others, while the resulting edit script needs to remain executable.

Automated tool support is in order, but far from trivial to provide. A straightforward tool might enable users to specify the sought-after transformation in terms of changes. Such a tool, however, would suffer from several problems. Figure 1 depicts an example transformation, in which a field of the class `Example` is renamed between two revisions. We want to extract the changes that perform this field rename. Three potential change sequences that can be returned by a distilling algorithm are shown at the bottom. In order to extract the desired changes, the tool need would need to overcome the following problems:

- Different change sequences can implement the same code transformation, as illustrated by Figure 1. The corresponding problem is two-fold. On the one hand, due to the heuristic nature of the distiller there is no straightforward way to know beforehand what change sequence will be output by a distilling algorithm. Very different change sequences can be distilled for similar modifications to similar files. On the other hand, it is not practical for a user to enumerate all the change sequences that could possibly be distilled. We call this problem the *Change Equivalence Problem.*
- A change sequence must be applied *in order* as any change potentially depends on a predecessor. Extracting an executable subset of changes means that these dependencies must be incorporated in the resulting edit script. Detecting these dependencies is not straightforward. Change distilling algorithms internally immediately apply each change as it is distilled. In order not to modify the original source code a copy is taken. As such, a change can refer to three different ASTs; the original source code, the target source code,



```
public class Example {
  int x = 0; }
```

```
public class Example {
  int y = 0; }
```

```
1. update("x", "y")
```

```
1. delete("int x = 0;")
2. insert("int y = 0;")
```

```
1. insert("int y = 0;")
2. delete("int x = 0;")
```

Fig. 1: Three different change sequences that each rename the field x of revision 1 into the field y of revision 2.

and a copy of the original source code (denoted *source'*) that will look identical to the target source code after the execution of the algorithm. For example, a delete can only be represented using nodes from the source AST, as the node is not present in the target AST (otherwise it would not have been removed), nor is it present in source' as the delete has already been applied. An insert on the other hand only has nodes that are present in source' and target, but not in source (otherwise it would not have been inserted). Computing dependencies between changes needs to account for these three different ASTs, as comparing nodes from different ASTs for equality produces incorrect results. We call this problem the *Change Representation Problem*.

We present an approach that enables users to specify evolutions of source code (e.g., a method rename refactoring was performed, the signature of a method was modified and its callers are updated, etc.), and that returns a minimal, executable source code transformation from a sequence of distilled changes. Example use cases are creating higher-level changes that provide the intent of groups of changes, detecting what additional changes are needed to execute a desired transformation or detecting what non-transformation related changes were applied to the source code.

This paper makes the following three contributions:

- First, we present a dedicated approach for specifying and extracting executable code transformations from a distilled sequence of source code changes. We introduce the term "evolution query" for queries in this approach that describe the sought-after change subsequences as a sequence of source code characteristics rather than as a sequence of changes. For a given transformation, an evolution query describes the desired intermediate ASTs characteristics that should hold along any sequence of changes that realizes this transformation.
- Second, we describe how to execute evolution queries against a distilled change sequence, which is represented as a graph of intermediate AST states. This AST graph is, in turn, constructed from a change dependency graph in which the order dependencies among changes in a change sequence are made explicit. As such, our approach supports describing *and* evaluating evolution queries in a manner that is agnostic to the concrete change sequence computed by a distilling algorithm.
- Third and finally, we evaluate our approach by means of specifying, detecting and extracting minimal executable transformations of three refactorings across different open-source projects.

II. Overview of the Approach

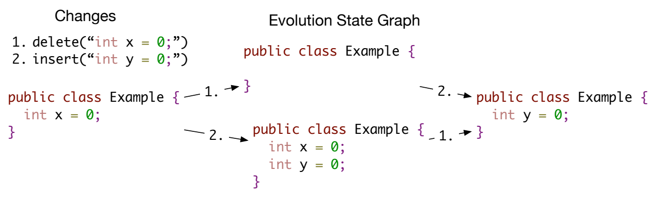
We propose an approach for specifying and extracting executable code transformations in a distilled sequence of changes between a source and a target AST. Unique to this approach is that it enables specifying an evolution query in terms of source code characteristics; those that intermediate ASTs,constructed by applying subsets of the sequence of distilled changes, must exhibit. This shields users from the problems that specifying evolution queries in terms of distilled changes gives rise to.

To make the notion of an evolution query more clear: it describes paths in a so-called Evolution State Graph (ESG), constructed from a distilled change sequence. Figure 2 depicts the ESG for the distilled sequence in the middle of Figure 1. The nodes of the ESG, called Evolution States (ES), contain an AST state and the specific changes that transformed the source AST into this state. An evolution query describes both the path and the code elements that need to be present in the nodes along this path. These code elements are described using Ekeko [9], a logic program query language.

Figure 3 depicts an overview of our approach. There are two revisions of the same file, called Rev1 and Rev2. The goal is to detect instances of a user-specified evolution in the changes between them. To this end, the files are passed as inputs to a change distiller called ChangeNodes, detailed in Section III. The distiller's output is a sequence of changes that transform the AST of the source file into the AST of the target file. We convert this sequence into an auxiliary Change Dependency Graph (CDG) that makes the dependencies among individual changes explicit. For instance, the CDG encodes the fact that an AST node cannot be inserted by a change operation if its parent node has not been inserted by a preceding change operation. Section IV-A details all change dependencies. Next, the Evolution State Graph is constructed. The process starts from a single Evolution State node containing the original AST of the source file. Future ES are created by executing changes without any unresolved dependencies, for which the CDG is consulted. The solution to such a declarative evolution query is an ES, containing an *executable* script of changes that implements the evolution pattern specified by the user. This script consists of the minimal amount of changes needed to implement the evolution pattern, and any changes that would no longer be executable if the rest of the solution were executed. As such, our approach ensures that the remainder of the change sequence remains executable as well.

A. Ekeko, *A Declarative Program Querying Language*

In order to specify the source code characteristics intermediate AST should exhibit we use the program query language Ekeko [9]. Ekeko is a Clojure library for applicative logic meta-programming against an Eclipse workspace. It provides a library of logic predicates that can be used to query programs. These predicates reify the basic structural, control flow and data flow relations of the queried Eclipse projects, as well as higher-level relations that are derived from the basic ones.



```
1. delete("int x = 0;")
2. insert("int y = 0;")

public class Example {
  int x = 0;
}

public class Example {
}

public class Example {
  int x = 0;
  int y = 0;
}

public class Example {
  int y = 0;
}
```

Fig. 2: Evolution state graph (ESG) for a sequence of distilled changes. Edge labels correspond to applied changes.

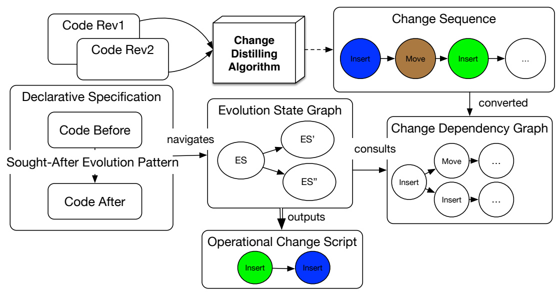


Fig. 3: Graphical overview of the approach.

EKEKO features several predicates that reify structural relations computed from the Eclipse JDT and that we use throughout this paper. Binary predicate `(ast ?kind ?node)`, for instance, reifies the relation of all AST nodes of a particular type. Here, `?kind` is a Clojure keyword denoting the decapitalised, unqualified name of `?node`'s class. Solutions to the query `(ast :MethodInvocation ?inv)` therefore comprise all method invocations in the source code.

Throughout this paper, we use a naming convention of predicates that reify an n-ary relation consist of n components separated by a -, each describing an element of the relation. Vertical bars | separate words within the description of a single component. For example, binary predicate `method-string|named` unifies its first argument with a method declaration and its second argument with the string representation of the name of that method.

B. Motivating Example Revisited: Querying the ESG

Figure 4 illustrates the specification language of our approach. Depicted is a logic query that finds instances of a field rename by navigating the ESG. The query works by describing an AST in which the field is present, and a later AST in which a field is added and removed. To this end, line 3 launches an evolution query through the `query-changes` construct. It takes as input an ESG and unifies its second argument with the end state of a matching path. Its third argument is a collection of logic variables, made available to the remainder of its arguments. These comprise a sequence of instructions that either verify that the current ES adheres to the given logic conditions, or navigate to another ES in the ESG. Lines 4–6 describe the initial state using `in-current-es`, which introduces two variables `es` and `ast`. `es` is bound to the current ES of the query, `ast` is bound to the AST of that ES[1]. Lines 5–6 describe the source code of that AST, in which a `?field` needs to be present at some depth [2]. Next, line 7 applies an arbitrary, non-zero, amount of changes using the operator `change->+`. This will change the current ES for the remainder of the expression. Finally, lines 8–12 state that the current ES needs to have a newly field `?renamed`. To this end, lines 11–12 ensure that `?field` is not present in the current AST, and that `?renamed` is not present in the original AST. This is done based on the name of the field using the EKEKO predicate `ast-field|absent`.

Notice how this query does not suffer from the change equivalence and representation problems. The query supported by our approach only required the user to describe source code characteristics. The changes resulting in this source code are returned as part of the query's result. Users can therefore abstract away from the concrete changes that were distilled. By describing ASTs instead of changes we solve the problem of different change sequences implementing the same change pattern. This also makes it clear in which AST a node resides. Where necessary, auxiliaries are provided to retrieve the corresponding node in a different intermediate AST.

```
1 (defn field-rename [esg]
2   (run* [?es]
3     (query-changes esg ?es [?orig-ast ?field]
4       (in-current-es [es ast]
5         (== ?orig-ast ast)
6         (ast-field ast ?field))
7       change->+
8       (in-current-es [es ast]
9         (fresh [?renamed ?new-name]
10          (ast-field|ast ast ?renamed)
11          (ast-field|absent ast ?field)
12          (ast-field|absent ?orig-ast ?renamed)))))
```

Fig. 4: Querying an ESG for changes renaming a field.

C. Example Applications and the Corresponding Queries

We illustrate the advantages of our approach through two example applications. In the first example, we are tasked with determining whether and which changes are responsible for introducing a new method in between two revisions. In the second, more complex example, we detect which changes from a distilled change sequence are responsible for eliminating a code clone.

1) Introduction of a Method: We first consider the problem of identifying the changes in a change sequence that are responsible for introducing a new method in between two revisions of a file. We define a method as newly introduced if no method with the same name was present in the original code. At first sight, it might suffice for a solution to the problem to query the change sequence for a single insert operation that added a `MethodDeclaration`. Inadvertently, however, a change sequence will be encountered in which the name of an existing `MethodDeclaration` has been changed by an update or a move operation. Before long, the query will have evolved into a large enumeration of potential change operations with a similar

[1]These are variables only visible in the body of `in-current-es`. If these variables need to be available in other parts of the query a user needs to explicitly bind them to a logic variable.

[2]Throughout this paper, logic variables are prefixed with a question mark.

effect. Operations that change the signature of the method, for instance, might also have to be accounted for.
Instead, it is much easier to detect an intermediate AST in which a new method is present, and retrieve the changes that led to the creation of this AST. Figure 5 depicts a function that launches such an evolution query. The function takes as input an ESG for a particular change sequence, and returns pairs of a method that has been introduced and the corresponding evolution state (ES). To this end, the function launches a query on line 2 that will find solutions for a pair of variables `?method` and `?es`. Lines 3–9 describe a path through the ESG that ends in an evolution state `?es`. Lines 3–5 describe the initial state on this path, for which a logic variable `?absent` is introduced. Line 5 binds this variable to the source AST, as so far no changes have been executed on the path. To this end, we use `in-current-es`, which introduces two new variables `es` and `ast`, bound to the current ES and its corresponding AST. Line 6 executes an arbitrary, non-zero amount of changes using `change->+`. Lines 7–9 verify that a new method is added to the current ES. To this end, we bind `?method` to any method declaration in the current ES, and verify that that method was not present in the original AST using `ast-method|absent`. The query returns all different ES that exhibit these characteristics.

```
1 (defn introduced-method [esg]
2   (run* [?method ?es]
3     (query-changes esg ?es [?absent]
4       (in-current-es [es ast]
5         (== ?absent ast))
6       change->+
7       (in-current-es [es ast]
8         (ast-method ast ?method)
9         (ast-method|absent ?absent ?method)))))
```

Fig. 5: Querying an ESG for changes introducing a method.

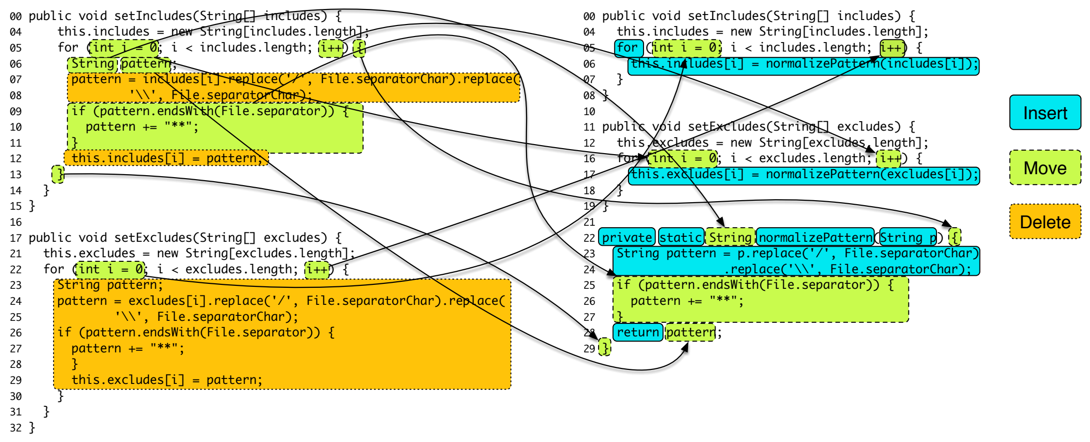
2) Code Clone Elimination: For the final example application, we are tasked with finding the changes in between two revisions that resulted in the removal of a code clone. Such an application may be interesting to MSR researchers to detect how code clones are removed, and what additional changes were performed next to the clone removal. We will look for evidence of a removal technique involving the *extract method* refactoring: the cloned code is extracted to a new method, and each clone instance is replaced by a method invocation to the newly introduced method. A concrete example of such a clone removal exists in the APACHE ANT[3] project. Commit `6bdc259c2e818e1c86f944cbd8950e670294d944` removes a code clone from file `DirectoryScanner.java`. Figure 6 depicts the changes distilled from this commit. We only show a small snippet of the original source file, which is slightly over 1500 lines of code.The semantics of these changes are explained in section III. We assume that the code clone has already been detected using an existing tool such as CCFINDER [10], and that the ESG has been created. We are only tasked with finding the specific changes that implemented the clone removal.
The first line of Figure 7 defines a function that takes as input an ESG, the names of two methods containing cloned code, and the extracted method AST node. The body of the function launches a logic query on line 2 returning a collection of all possible bindings for `?es`, which is the end node of a path throughout the ESG. Line 3 describes the shape of this path through a regular path expression. Lines 5–9 bind `cloneA` and `cloneB` to the clones detected in the source AST (i.e., `setIncludes` and `setExcludes` in the *left revision* in Figure 6). The `child+` predicate unifies the given logic variable with any node of the given AST. Line 10 navigates to a different node of the ESG by applying an arbitrary, non-zero amount of changes. Lines 11–19 specify a strict implementation of the extract method refactoring. Lines 12–13 require the presence of a method `?extracted` in the current ES that is identical to the method given as the `extracted` parameter to the function (i.e., `normalizePattern` in the *right revision* in Figure 6). This ensures that an ES node of the ESG has been reached in which all the changes extracting the cloned code have been applied. If not, the query will backtrack to line 10 and another change will be applied. Next, lines 14–15 retrieve the version in that ES of the methods in which the two instances of the cloned code resided originally (i.e., `setIncludes` and `setExcludes` in the *right revision* in Figure 6). They use auxiliary construct `ast-method-method|corresponding` to this end, which returns the corresponding method from an ast for a given method. Finally, lines 16–19 ensure that the methods containing cloned code have been extracted.

III. Detailed Change Definitions

To compute the changes made in between two revisions of a file, we rely on our own freely available[4] change distiller called CHANGENODES [5], [11]. At its heart lies the algorithm presented by Chawathe et al. [2], on which the CHANGEDISTILLER [1] tool was based as well. The main difference between both implementations is that CHANGENODES works on the actual nodes provided by the Eclipse Java Development Tools (JDT) project, while CHANGEDISTILLER uses a language-agnostic representation of nodes (to which JDT nodes are translated).
Accessing the children of a node is done through properties. For example, an if statement has three properties; an expression property, a then and an else property. Some properties may return a collection instead of a single item. Such properties 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. Our ESG construction algorithm relies on them to ensure the legality of the constructed intermediate AST states. As such, porting our approach to a different language or source representation requires a means to ensure that an AST represents syntactically legal code. 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

[3]https://ant.apache.org/

[4]https://github.com/ReinoutStevens/ChangeNodes



```java
public void setIncludes(String[] includes) {
    this.includes = new String[includes.length];
    for (int i = 0; i < includes.length; i++) {
      String pattern;
      pattern = includes[i].replace('/', File.separatorChar).replace(
              '\\', File.separatorChar);
      if (pattern.endsWith(File.separator)) {
        pattern += "**";
      }
      this.includes[i] = pattern;
    }
  }
}

public void setExcludes(String[] excludes) {
  this.excludes = new String[excludes.length];
  for (int i = 0; i < excludes.length; i++) {
    String pattern;
    pattern = excludes[i].replace('/', File.separatorChar).replace(
            '\\', File.separatorChar);
    if (pattern.endsWith(File.separator)) {
      pattern += "**";
      }
      this.excludes[i] = pattern;
    }
  }
}
```

```java
public void setIncludes(String[] includes) {
  this.includes = new String[includes.length];
  for (int i = 0; i < includes.length; i++) {
    this.includes[i] = normalizePattern(includes[i]);
  }
}

public void setExcludes(String[] excludes) {
  this.excludes = new String[excludes.length];
  for (int i = 0; i < excludes.length; i++) {
    this.excludes[i] = normalizePattern(excludes[i]);
  }
}

private static String normalizePattern(String p) {
  String pattern = p.replace('/', File.separatorChar)
                    .replace('\\', File.separatorChar);
  if (pattern.endsWith(File.separator)) {
    pattern += "**";
  }
  return pattern;
}
```

Fig. 6: Two revisions of class from the ANT project in between which a code clone is extracted into a separate method `normalizePattern`, to which two invocations are added. Overlaid is the output of our CHANGENODES change distilling algorithm.

```
(defn clone-elimination [esg nameA nameB extracted]
  (run* [?es]
    (query-changes esg ?es
      [?cloneA ?cloneB ?extracted ?aInvoc ?bInvoc ?aCurr ?bCurr]
      (in-current-es [es ast]
        (child+ ast ?cloneA)
        (child+ ast ?cloneB)
        (method-string|named ?cloneA nameA)
        (method-string|named ?cloneB nameB))
      change->+
      (in-current-es [es ast]
        (ast :MethodDeclaration ?extracted)
        (ast-ast|same ?extracted extracted)
        (ast-method-method|corresponding ast ?cloneA ?aCurr)
        (ast-method-method|corresponding ast ?cloneB ?bCurr)
        (child+ ?aCurr ?aInvoc)
        (child+ ?bCurr ?bInvoc)
        (method-invocation|invokes ?extracted ?aInvoc)
        (method-invocation|invokes ?extracted ?bInvoc)))))))
```

Fig. 7: Querying an ESG for extract method refactorings.

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 ...

We now define the semantics of the different types of change operations produced by the change distiller. Note that these change operations and are also used in the construction of the ESG. As a distilling algorithm applies changes during its execution, thereby modifying the AST, a copy of the source AST is taken. Throughout this paper we call this copy `source'` and indicate the nodes inside with a quote as well. ESG construction assumes the following changes:

`insert(node',original,parent',property,index)` A `node'` is inserted as `property` in node `parent'`. In case `property` is a list property, the node is inserted at `index`. `original` is the parent node in the original AST, and can be `null` if the insert is part of another change operation[5].

`move(node',original,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`. `original` represents the moved node in the original AST.

`update(node',original,property,value)` The value of node `node'` at location `property` is updated to `value`. This value is a Java object, and not an AST node. `original` is the representation of `node'` in the original AST.

`delete(node',original,property,index)` A node `node'` and its complete subtree are removed. In case `property` is a list property, `index` indicates the index of `node'` in its list. Note that `node'` will not be present in source' as the change has already been applied. `original` is the representation of `node'` in the original AST.

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.

IV. CONCEPTUAL IMPLEMENTATION

Having presented the necessary background on changes and change distilling, we are ready to present our approach in a more detailed manner. We target the problem of identifying executable subsequences in a distilled change sequence that implement an evolution pattern of interest. Our approach recalls different subsequences that implement the same evolution pattern, specified as paths through a graph of intermediate AST states. This spares users the "Change Equivalence" and "Change Representation" problems identified in Section I.

An Evolution State Graph (ESG) is constructed, against which our approach evaluates evolution queries. The evolution queries themselves feature *regular path expressions* [12], [13] for describing paths through the ESG, and the source code characteristics that have to be encountered along this path. In general, a regular path expression describes a path through a graph, for which conditions have to hold in nodes along that path. They are akin to regular expressions, except that a) their

[5]This is because the parent itself is not yet constructed, and thus does not exist in `source`.

elements are evaluated against the nodes of a graph rather than the characters of a string; and that b) some elements can explicitly navigate to another node of the graph, against which the next element of the regular path expression will be evaluated.

Our evolution queries are logic queries with embedded regular path expressions. Table I provides an overview of our particular embedding. We provide constructs for navigating an ESG, such as `change->` which moves evaluation to a successor of the current node in the ESG. We also provide constructs such as `in-current-es` for evaluating logic conditions against the current node of the ESG. Such embedded conditions comprise the primary means for describing the source code characteristics that need to hold along a path of the ESG.

A. Construction of a Change Dependency Graph

Section IV-B will detail an algorithm for constructing the Evolution State Graph (ESG) against which our approach evaluates evolution queries. The algorithm relies on a model of the order dependencies among the changes in a distilled change sequence. Even though such a sequence is by definition ordered (i.e., the distiller guarantees the sequence transforms the source AST into the target AST when the changes are executed in order), additional order dependencies are required because evolution queries are to identify (possibly non-continuous) *sub*sequences that implement a pattern of interest. Individual changes in such a subsequence, can depend on any change that preceded them in the distilled sequence.

A dependency $A \rightarrow B$ between changes A and B denotes that in order to execute change B, one needs to execute change A first. We gather all dependencies among the changes in a change sequence in a Change Dependency Graph (CDG), of which the nodes correspond to changes and the directed edges to dependencies. We compute the following kinds of dependencies:

Parent Dependency There is a parent dependency $A \rightarrow_p B$ between changes A and B if the subject of B is introduced by the application of A. Nodes can be introduced either by an insert or by a move operation. We determine this dependency by checking whether the subject of change B is part of the subtree created by the application of change A.

Move Dependency There is a move dependency $A \rightarrow_m B$ between changes A and B if B removes part of A, rendering it impossible to move. This can happen either by a delete or by an insert that overwrites the part of the AST in which the node-to-be-moved resides.

List Dependency There is a list dependency $A \rightarrow_l B$ between changes A and B if they operate on elements of the same list, but the element of B has a lower index than the element of A. Although changes A and B can be applied independently of each other, the index of A will change depending on whether B has already been applied or not.

The subsequent ESG construction algorithm will require the CDG to be acyclic. Particular combinations of the above dependencies can, however, induce cycles in the graph. For example, the combination of two moves performing a swap operation result in a cycle as the application of either move renders the other one inapplicable. Such cycles are removed by replacing one of these moves with an insert.

B. Construction of the Evolution State Graph

We now explain how the Change Dependency Graph (CDG) from the previous section enables constructing the Evolution State Graph (ESG) that is navigated through by a regular path expression. The ESG represents the possible ASTs that can be constructed by applying some of the distilled changes. A single ESG node wraps a syntactically legal AST and an ordered sequence of changes that were applied to construct that AST. Two ESG nodes are connected if there exists an unapplied change that transforms the AST of one into the AST of the other. The resulting edge is labeled by the applied change. A single change can appear on multiple edges in the graph.

The ESG has one source node (i.e., the node containing the original source code with no applied changes) and one sink node (i.e., the node containing the target source code and no unapplied changes). The graph is constructed using the information provided by the CDG. Initially, the source node is constructed from the source AST. Successors of the source node are constructed by applying a change without dependencies. The CDG facilitates the retrieval of applicable changes given a set of applied changes. The ESG is constructed on-demand; nodes and their ASTs are only created as needed when executing an evolution query.

V. Evaluation

The examples in Section II-C served to demonstrate the expressiveness of the specification language of our approach. We now seek to answer the following research questions:

RQ1 Can a *single* evolution query identify the same evolution pattern in *different* change sequences?

RQ2 Is a *minimal* and *executable* change script returned, and can the *remaining distilled changes* still be executed after the change script has been executed?

RQ3 How does our approach compare to directly querying the output of a distilled change sequence with respect to solution size, precision and the number of changes that need be executed?

To answer these questions, we will use a data set of commits from open-source repositories that each contain —among many other changes— one of three well-known refactorings (Section V-A). We aim to extract the exact changes contributing to each refactoring among all of the changes distilled for each commit.

For each refactoring, we will attempt to specify the state of the source code before and after the refactoring by means of a declarative evolution query (Section V-B). Each solution to such a query is an executable script of changes. When executed, this script will transform the source code from *before the commit* to a state that matches the specified state of the code *after the refactoring*. In other words, the extracted changes will perform the specified refactoring. The remaining changes distilled for the commit will, when executed, in turn

TABLE I: Language for specifying evolution patterns through ESG-navigating regular path expressions.

Navigation through the ESG	
`change->`	`change->` is a goal that moves the current state to the next one by applying one of the applicable changes.
`change->?`	`change->?` is a goal that either stays in the current state or that moves to the next one by applying one of the applicable changes.
`change->*`	`change->*` is a goal that changes the current state by applying an arbitrary, including zero, number of changes.
`change->+`	`change->+` is similar to `change->*`, except that at least one change will be applied.
`change==>`	`change==>` is a goal that moves the current state to a successive one by applying one of the applicable changes and all of its dependent changes.
`change==>*`	`change==>` is a goal that changes the current state by applying an arbitrary, including zero, number of changes and their dependent changes.
Characteristics of an ES	
`(in-current-es [node ast] & goals)`	`in-current-es` binds `es` to the current es of the evolution query, and `ast` to the intermediate AST of that state. It verifies whether the logic goals `goals` hold in this intermediate state. These goals can be any EKEKO predicate.
Launching an Evolution Query	
`(query-changes esg ?end [&vars] & goals)`	`query-changes` launches a path query over `esg` and binds `?end` to the end node of that query. Logic variables `vars` are introduced and available in the scope of the path query. `goals` is a sequence of the aforementioned predicates that are proven for the given ESG.

transform the state of the code *after the refactoring* to the state of the code *after the commit*.

The approach computes a minimal solution — that is the smallest subset of changes that implements the code evolution. To this end, it retrieves the ES that matches the evolution query with the smallest amount of applied changes. We manually verify the solutions depicted in Table II on their minimality. We also compute various metrics pertaining to each research question (Section V-C).

A. Data Set of Commits Containing Refactorings

Our evaluation proceeds on a data set of commits that each contain, among other changes, a "*Replace Magic Constant*", "*Remove Unused Method*" or "*Rename Field*" refactoring. This random selection of refactorings is sufficiently varied in the number of changes required to perform them, as well as in the types of AST nodes affected by them. Table II lists the identifier of each commit, the open source project repository it originates from, the name of the refactoring it contains, the name of the class affected by the refactoring, and the oracle according to which the commit contains the refactoring. The oracle is indicated by the number in the refactoring column. We have used two such oracles:

- The first oracle, denoted by a 1 subscript, corresponds to a data set[6] produced by the REF-FINDER [14] tool which recognizes refactorings in commit histories using coarse-grained change information (e.g., changes in the subtyping relation). We manually inspected all occurrences of the "*Replace Magic Constant*" refactoring in this data set, discarded the false positives, and —without loss of generality— discarded the commits that span multiple files. The latter because our prototype implementation is currently limited to querying changes between two revisions of the same file. The commits listed in Table II are all such commits in the RF data set.
- The second oracle, denoted by a 2 subscript, corresponds to a data set[7] resulting from a study my Murphy-Hill et. al [15] of logs of interactions of developers with the refactoring functionality of their IDE. Each commit in this data set has already been cross-checked by the authors with the interaction logs. After filtering commits that span multiple files, we are left with 3 instances each of the "*Remove Unused Method*" or "*Field Rename*" refactorings in Table II.

[6]http://web.cs.ucla.edu/~miryung/inspected_dataset.zip
[7]http://multiview.cs.pdx.edu/refactoring/experiments/

Note that the identifiers listed in Table II differ depending on the data set the commit stems from. For commits with subscript 1, the short identifier from the project's GitHub repository is used. For commits with subscript 2, we use the same identifier as the authors of the original study.

B. Queries for Changes Implementing Refactorings

We describe the queries used to identify the exact changes contributing to the "replace magic constant" and "rename field" refactoring. The "remove unused method" query is not provided, but detects whether the initial ES contains an unused method, and that that method is removed (based on its name) in a successive ES. The query results will be discussed in the next section.

1) Query for "Replace Magic Constant": Figure 8 depicts the query that identifies changes from a commit that implement a "*Replace Magic Constant*" refactoring. This refactoring extracts a literal string or number from the body of a method to a field, and updates the method such that it references the newly introduced field. The first line of Figure 8 launches the query for a path ending in an Evolution State `?es` through Evolution State Graph `esg`. Lines 2–3 introduce additional logic variables used throughout the query. Lines 4–8 describe the initial Evolution State of the source code. Line 5 unifies the original AST with `?absent`, so that it can be used later to determine whether a fresh field has been introduced. Lines 6–8 identify a method `?method` that contains a constant value `?value`. Line 9 uses the `change->*` operator to apply an arbitrary number of changes. Lines 10–15 describe a future Evolution State, in which a new field has been introduced to replace the constant value. To this end, the `ast-ast-field|introduced` ensures that its third argument `?field` is absent from its first AST argument, but present in its second. Line 12 ensures that this field features the constant `?value` as its initializer expression. Line 13 uses `ast-method-method|corresponding` to retrieve a method `?cmethod` in the current Evolution State

```
1 (query-changes esg ?es
2  [?not-present ?method ?literal value
3   ?cmethod ?field ?field-access]
4   (in-current-es [es ast]
5     (== ast ?absent)
6     (ast-method ast ?method)
7     (child+ ?method ?literal)
8     (literal-value ?literal ?value))
9   change->*
10  (in-current-es [es ast]
11    (ast-ast-field|introduced ?absent ast ?field)
12    (field-value|initialized ?field ?value)
13    (ast-method-method|corresponding ast ?method ?cmethod)
14    (child+ ?cmethod ?field-access)
15    (field-name|accessed ?field ?field-access)))
```

Fig. 8: Evolution query for those changes in a commit that implement a “*Replace Magic Constant*” refactoring.

```
1 (query-changes esg ?es
2  [?original ?field ?accesses
3   ?count ?renamed ?renamed-accesses]
4  (in-current-es [es ast]
5    (== ast ?original)
6    (child+ ast ?field)
7    (ast-field ast ?field)
8    (ast-field-list|accesses ast ?field ?accesses)
9    (length ?accesses ?count))
10 change->*
11 (in-current-es [es ast]
12   (child+ ast ?renamed)
13   (ast-field ast ?renamed)
14   (ast-field|absent ?original ?renamed)
15   (ast-field|absent ast ?field)
16   (ast-field-list|accesses ast ?renamed ?renamed-accesses)
17   (length ?renamed-accesses ?count)
18   (ast-field|unaccessed ast ?field)))
```

Fig. 9: Evolution query for those changes in a commit that implement a *“Rename Field”* refactoring.

that corresponds to `?method` in the original one. The names and signatures of the methods are required to match, but not their bodies. Finally, lines 14–15 ensure that this method now accesses the newly introduced field.

2) Query for “Rename Field”: Figure 9 depicts the evolution query that identifies changes implementing a *“Rename Field”* refactoring. Lines 4–9 describe an initial ES in which a field is present. Lines 11–21 describe a later ES in which that field and its accesses are absent, and in which a new field has been introduced that has the same number of accesses. Line 5 unifies `?original` with the AST of that ES, so that it can be used in future ES. Next, lines 6–7 unify `?field` with a field declaration of that AST. Finally, lines 8–9 retrieve all the uses of that field in a list `?accesses` with length `?count`. Line 10 uses `change->*` to apply an arbitrary number of changes. Lines 11–18 describe the later ES in which the refactoring has been completed. To this end, lines 12–13 unify `?renamed` with a field declaration. Line 14 uses `ast-field|absent` to ensure that `?renamed` is absent from the original AST, while line 15 ensures that `?field` is absent from the current AST. Next, lines 16–17 verify that this new field is used as often as the original variable. Finally, the last line ensures that no accesses to the original field are present in the AST.

C. Query Results

As stated before, Table II depicts the results of our validation. The first part describes the detected refactoring and the data set from which this refactoring stems. The second part of the table depicts metrics about the distilled changes and corresponding CDG. Column *#Ch* depicts the total number of distilled changes for the file. Next, column *LP* depicts the length of the longest path throughout the CDG. Column *MP* depicts the median length of the paths throughout CDG. Both indicate, using our approach, how many changes that need to be applied before a given change becomes applicable. Directly using the output of a change distiller this would be all preceding changes. Finally, Column *#Co* depicts the number of connected components inside the CDG. These indicate the clusters of independent changes. Thus, the columns *LP*, *MP* and *#Co* illustrate the number of actual dependencies an arbitrary changes has.

The last part of the table describe the found minimal solution. Column *#Sol* depicts the number of changes the minimal, executable solution returned by the query consists of. Next, Columns *LS* and *MS* depict respectively the longest and median span, indicating how many changes were distilled between two successive changes in the solution. Column *#DS* depicts the total number of changes that would have been applied if the distilled output is used directly before the described evolution pattern would be present. Thus, the columns *LS*, *MS* and *#DS* indicate how many unneeded changes would be applied when not using our approach, while *#Sol* depicts the total number of changes that actually need to be applied. The next three columns depict a manual classification of the solution into either *Evolution Implementing* (*#EI*), *Evolution Supporting* (*#ES*) and *Evolution Linking* (*#EL*) ones:

Evolution Implementing An EI change is an integral part of the sought-after evolution pattern. In the *“Rename Field”* refactoring, for example, the change modifying the name of the field is considered as evolution inherent.

Evolution Supporting An ES change is not an integral part of the sought-after evolution pattern, but is depended on by one of its EI changes. Without the ES change, the EI change would no longer be executable. For example, an EI change inserting a field access into a method body depends on ES changes preparing that method’s body.

Evolution Linking An EL change is included in the minimal solution, but is neither an EI nor an ES change. EL changes ensure that the remainder of the distilled changes can still be executed after each change in the solution has been executed. As such, they link the solution to the rest of the distilled changes. For example, when parts of a method that fell victim to the *“Remove Method”* refactoring are moved and subsequently changed elsewhere, the minimal solution will include these moves as EL changes. These changes could be removed from the solution by our approach.

Finally, the last column indicates the total run-time in seconds of distilling the changes, constructing the CDG and finding the minimal solution.

Figure 10 illustrates this classification of the changes in the the solution to the “*Replace Magic Constant*” evolution query against commit `8275917`. Before the commit, method `getUrl()` contained the constant `0` twice: once as a magic constant

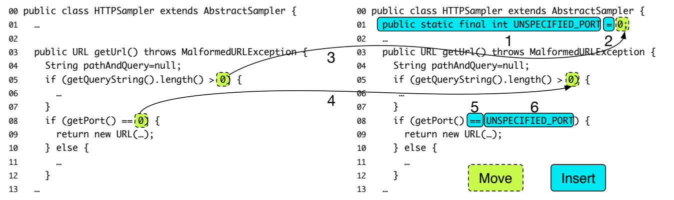
TABLE II: Table depicting the result of our validation. The first four columns describe the used data and detected refactoring. The next four columns describe the distilled changes and their corresponding CDG. The final columns describe the minimal solution and its changes.

Ref.	**Project**	**Id**	**Class**	**#Ch**	**LP**	**MP**	**#Co**	**#Sol**	**LS**	**MS**	**#DS**	**#EI**	**#ES**	**#EL**	**Time(s)**
Co_1	ant	d97f4f3	`WeblogicDeploymentTool`	202	14	8	14	8	29	8	82	4	4	0	33
Co_1	ant	34dc512	`Jar`	74	10	3	13	4	11	7	23	4	0	0	19
Co_1	ant	a794b2b	`FixCRLF`	1244	20	6	2	10	300	68	1000	4	6	0	2054
Co_1	JMeter	b57a7b3	`AuthPanel`	245	10	4	31	5	135	23	199	4	1	0	106
Co_1	JMeter	3a53a0a	`HTTPSampler`	149	7	2	54	5	96	8.5	118	4	1	0	1985
Co_1	JMeter	8275917	`HTTPSampler`	25	5	2	5	6	4	3	14	4	1	1	285
Me_2	jdt.ui	678	`JavaEditor`	11	2	1	10	2	1	0.5	1	1	0	1	748
Me_2	jdt.ui	2910	`JavaNavigatorContentProvider`	5	1	1	5	1	0	0	0	1	0	0	3
Me_2	jdt.ui	2722	`StubUtility2`	291	8	2	114	39	22	4.5	264	2	0	37	1536
Fi_2	jdt.ui.test	0277	`MarkerResolutionTest`	10	4	2.5	5	3	3	2	9	3	0	0	53
Fi_2	jdt.ui	2810	`SourceAnalyzer`	63	7	2	25	13	13	3	57	13	0	0	8842
Fi_2	jdt.ui	2810	`SourceProvider`	27	6	2	13	15	3	1	24	10	5	0	3133
Fi_2	jdt.ui	2810	`InlineMethodRefactoring`	221	14	3	61	6	77	29	213	6	0	0	5757

on line 8, and once as part of a check for an empty list on line 5. In the depicted solution, change 1 inserts a new field "`private static int UNSPECIFIED_PORT;`", change 2 inserts an initializer expression "`...= null;`" into the field, and change 3 moves the latter 0 to replace the `null` in the initializer, leaving a copy of the value behind on line 5 as it is a mandatory node (cf. the minimal representation of AST nodes discussed in Section III). Change 4 then moves the former 0 from line 8 to replace the one on line 5. Change 5 overwrites the infix expression on line 8 as its textual representation differs too much between both revisions. The left hand side is kept, while change 6 inserts a new field access in the right hand side. Changes 1, 2, 3 and 6 in the solution are EI changes implementing the actual sought-after refactoring. Change 5 is an ES change as it is depended upon by change 6. Change 4 is an EL change as it would no longer be applicable after the application of change 6, which overwrites the node-to-be-moved. It is not required for the sought-after refactoring, Note that we performed this classification manually, ensuring that the sum of *#EI*, *#ES*, and *#EL* is always *#Sol*.

1) Results for "Replace Magic Constant": For each of the refactoring commits from projects `ant` and `JMeter`, the evolution query depicted in Figure 8 reports a minimal solution consisting of the 4 sought-after EI changes: two for inserting a new field declaration and its name, one for copying the magic constant to the field initializer, and one for replacing the constant with an access to the inserted field. The remaining ES changes always prepare a parent node for this field access. The EL change in the minimal solution for commit `8275917` was explained by Figure 10.

Note that the size of the change sequence distilled for the entire commit varies wildly, as does their complexity. The CDG hugely reduces the number of changes that need to be applied for any given change. Thus, the returned minimal solutions always consist out of a very small number of changes. The minimal solution for commit `a794b2b`, for example, counts only 11 of the 1244 changes distilled in total. More than doubling class `FixCRLF` from 429 to 972 lines of code, this commit contains many changes unrelated to the sought-after ones. Here, we also find the largest span in the distilled change sequence between any two solution changes: 300 successive



```
00 public class HTTPSampler extends AbstractSampler {
01   …
02
03   public URL getUrl() throws MalformedURLException {
04     String pathAndQuery=null;
05     if (getQueryString().length() > 0) {
06       …
07     }
08     if (getPort() == 0) {
09       return new URL(…);
10     } else {
11       …
12     }
13   …
```

```
00 public class HTTPSampler extends AbstractSampler {
01   public static final int UNSPECIFIED_PORT = 0;
02   …
03   public URL getUrl() throws MalformedURLException {
04     String pathAndQuery=null;
05     if (getQueryString().length() > 0) {
06       …
07     }
08     if (getPort() == UNSPECIFIED_PORT) {
09       return new URL(…);
10     } else {
11       …
12     }
13   …
```

Fig. 10: Code snippet from the `HTTPSampler` class, in which a field is introduced (1,2). This field is initialized via a move of a constant (3), which itself is replaced by another move (4). Move 4 is an EL change as its node-to-be-moved is overwritten by a later insert (6). Insert 5, unnecessarily, overwrites the parent location of change 6, and is classified as an ES change.

changes would have to be searched through to find the next change that is part of the solution, and 1000 changes would be applied in total, compared to 10 changes using our approach.

2) Results for "Remove Unused Method": The sought-after refactoring can be performed by a single change, namely a delete of the unused method. Inspecting the results we note that this only holds for a single case. The returned solution solution for `StubUtility2` even contains 39 changes in total. This is due to parts of the removed method being moved to different locations by other changes. These moves are part of the solution, and are classified as EL changes. We also note that there are 2 EI changes: two methods with the same name are removed. This is due to the declarative specification, requiring that no methods with the same name are present. A stricter specification would prevent this from happening.

3) Results for "Rename Field": The final results are for the "Rename Field" refactoring. The number of changes in the solution differ across the different instances. This is due to the nature of the refactoring, as it requires that every access is updated to reflect the name change. Implementing this query without our approach, but by directly querying the distilled changes, would be hard as the number of changes is not known beforehand. These changes can also span the entire change sequence, as the accesses can happen throughout the whole AST. We note that the run-time for all but one example are high compared to the other refactorings. This can be attributed to the nature of the declarative description of the source code, which takes several seconds to run on a single ES.

From this validation we can answer our three RQ:

a) RQ1: We have successfully used a single evolution query to identify the same pattern in different change sequences created from source code that exhibits the desired pattern. As such, we can positively answer RQ1.

b) RQ2: We have manually inspected the returned change sequences and classified these changes. Some solutions may still contain EL changes that are, strictly speaking, not part of the minimal solution. Such changes must be applied before some other solution changes so that the EL changes or non-solution changes can be applied. It is left to the user to remove any unwanted EL changes. The returned solutions are relatively small compared to solutions returned by directly querying the distilled changes, making a manual inspection feasible. Next, we have also made sure the remaining changes can still be applied after applying the solution. As such, we can positively answer RQ2.

c) RQ3: We have shown several metrics regarding our computed solutions, the distilled changes and the CDG. From the LS, MS and #DS – indicating how solution changes are interspersed between non-solution ones – we deduce that replaying the distilled change sequence until a desired ES is found results in much larger solutions. We do note that our returned solutions may still contain EL changes that a user, if desired, wants to filter out. Nonetheless, the number of changes that would have to be inspected is a lot lower than using a direct approach. We also want to stress that our approach focuses on finding a *minimal* solution; if the goal is to know whether a change sequence implements a certain code evolution simply replaying the distilled changes until a desired state is encountered is recommended.

VI. Related Work

Our work lies at the intersection of multiple domains: Program and History Querying Tools, History Querying Tools, Change Distilling Algorithms and Change Dependencies. Program Querying Tools identify source code elements that exhibit user-specified characteristics. Enabling users to specify these characteristics in logic-based languages has proven to result in expressive, yet descriptive specifications. This requires reifying code as data in a logic language. Examples of such logic-based program querying tools include CodeQuest [16], PQL [17] and SOUL [18]. History Querying Tools extend the idea of Program Querying Tools by allowing querying the history of a software project instead of a single revision. Early History Querying Tools, such as SCQL [19] and V-Praxis [20], extended a PQL by adding a revision argument to each predicate. More recent tools feature dedicated specification languages. The Boa platform [21] allows efficient querying of the history of a program by using MapReduce. History Querying Tools provide a history of the source code, but offer no support to query concrete source code changes that were performed. They do provide a good starting point to integrate an evolution query language in.

ChangeDistiller [1] is a widely used implementation of a distilling algorithm that has been implemented as a plugin in the Evolizer platform. The algorithm itself is based on the algorithm presented by Chawathe et al. [2]. gumtree [22] is another distilling algorithm that proposes a hybrid approach between line-based differencing and tree-based differencing to improve the performance of the algorithm. In this paper we make use of ChangeNodes, a distilling algorithm operating directly on Eclipse JDT nodes. All algorithms provide similar output, and thus feature the same problems as directly querying the output of ChangeNodes.

VII. Discussion and Future Work

The presented work facilitates users in expressing and detecting evolution patterns. Without our approach a user would have to manually inspect the changes of a distilled change sequence in order to identify the changes implementing the sought-after pattern. While cumbersome, such a manual approach results in a minimal set of changes implementing a pattern. Our approach enables users to express evolution characteristics through a declarative specification.

Currently, we have only used our approach to detect refactorings, for which the result is present in the target AST. In theory a sought-after transformation could only be present in some ES, but not in the final ES. It is ill-advised to detect such ES. First, the construction of the ES depends on the distilled change sequence. As such, there is no way to know beforehand whether the desired ES will actually be present, as an unexpected change sequence may be generated. Second and finally, the worst-case performance in detecting a specific ES is an issue. For a given set of changes with size N, $N!$ different sequences with length N can be constructed in case no change has a dependency. As such, detecting such ES requires replaying the different change sequences. To partially solve this issue, we have introduced coarse-grained navigation predicates that apply multiple changes at once, limiting the search space at the cost of removing ES that may contain the solution.

We want to investigate further applications of our approach. We want to investigate whether we can cherry-pick changes from a commit, for example to extract a single feature. This feature can be expressed as an evolution query, and our approach returns minimal executable edit script. We want to see whether such edit script can be applied on similar source code, such as code from a different branch. To this end, we can use our approach to detect the differences between the source code across the two branches.

VIII. Conclusion

We have presented an approach to extracting a minimal executable edit script from distilled change sequences. An evolution query describes the source code prior and after the sought-after code transformation. Such a query can be matched against any distilled code sequence. The approach detects whether the transformation is present, and if so, returns a minimal executable edit script.

References

[1] 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.

[2] 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.

[3] N. Palix, J. Falleri, and J. Lawall, "Improving pattern tracking with a language-aware tree differencing algorithm," in *Proc. of the 22nd Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER15)*, 2015, pp. 43–52.

[4] R. Stevens and C. De Roover, "Querying the history of software projects using QWALKEKO," in *Proc. of the 30th Int. Conf. on Software Maintenance and Evolution*, 2014.

[5] 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.

[6] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *Proc. of the 35th Int. Conf. on Software Engineering (ICSE13)*, 2013.

[7] Z. Lin and J. Whitehead, "Why power laws?: An explanation from fine-grained code changes," in *Proc. of the 12th Working Conf. on Mining Software Repositories (MSR15)*, 2015.

[8] 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 Int. Conf. on Software Engineering (ICSE14)*, 2014.

[9] C. De Roover and R. Stevens, "Building development tools interactively using the ekeko meta-programming library," in *Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR14)*, 2014.

[10] 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*, 2002.

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

[12] O. de Moor, D. Lacey, and E. V. Wyk, "Universal regular path queries," *Higher-Order and Symbolic Computation*, pp. 15–35, 2002.

[13] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu, "Parametric regular path queries." in *Proc. of the Conf. on Programming Language Design and Implementation (PLDI04)*, 2004.

[14] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proc. of the 2010 Int. Conf. on Software Maintenance (ICSM10)*, 2010.

[15] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *Transactions on Software Engineering*, vol. 38, pp. 5–18, 2012.

[16] E. Hajiyev, M. Verbaere, and O. D. Moor, "Codequest: Scalable source code queries with datalog," in *Proceedings of the 20th European conference on Object-Oriented Programming (ECOOP06)*, 2006.

[17] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," in *Proc. of the 20th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA05)*, 2005.

[18] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with Eclipse," in *Proc. of the 9th Int. Conf. on Principles and Practice of Programming in Java (PPPJ11)*, 2011.

[19] A. Hindle and D. M. German, "SCQL: A formal model and a query language for source control repositories," in *Proc. of the 2005 Working Conf. on Mining Software Repositories (MSR05)*, 2005.

[20] A. Mougenot, X. Blanc, and M.-P. Gervais, "D-Praxis: A peer-to-peer collaborative model editing framework," in *Proc. of the 9th Int. Conf. on Distributed Applications and Interoperable Systems (DAIS09)*, 2009.

[21] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proc. of the Int. Conf. on Software Engineering (ICSE13)*, 2013.

[22] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and accurate source code differencing," in *Proc. of the 29th Int. Conf. on Automated Software Engineering (ASE14*, 2014.

[23] K. Maruyama, T. Omori, and S. Hayashi, "Slicing fine-grained code change history," *IEICE Transactions*, vol. 99-D, no. 3, pp. 671–687, 2016.