Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Querying Distilled Code Changes to Extract Executable Transformations

Reinout Stevens · Tim Molderez · Coen De Roover

Received: date / Accepted: date

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. We show that our approach is able to recall different implementation variants of the same source code evolution in histories of different software projects. We also evaluate the expressiveness and ease-of-use of our approach in a user study.

Keywords Change Distilling · Change Querying · Logic Meta-Programming

Reinout Stevens
Maxflow BVBA
E-mail: reinout@reinoutstevens.be

Tim Molderez
Software Languages Lab
Vrije Universiteit Brussel
E-mail: tim.molderez@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
E-mail: coen.de.roover@vub.be

1 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 (Fluri et al 2007; Chawathe et al 1996; Palix et al 2015; Stevens and De Roover 2014) 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 (Christophe et al 2014; Meng et al 2013; Lin and Whitehead 2015; Negara et al 2014; Martinez et al 2013).

In this paper we address a problem faced by many MSR researchers; the problem of recognizing and extracting transformations of interest in the output of a change distiller, such that these transformations are represented as a *minimal* and *executable* edit script (i.e., a sequence of changes). 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 prevents the script from being executed, or renders 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 realize. A straightforward tool might enable users to specify the sought-after transformation in terms of changes. Such a tool would however 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 two problems:

– First, different change sequences can implement the same code transformation, as illustrated by Figure 1. The corresponding problem is two-fold. On

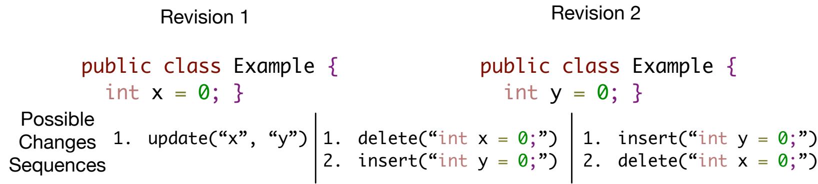

Fig. 1: Three different change sequences that each rename the field `x` of revision 1 into the field `y` of revision 2.

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*.

– Second, 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 made. As such, a change can refer to three different ASTs; the original source code, the target source code, and a copy of the original source code (denoted *source'*) that will eventually 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 Java 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 Java 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 performing two experiments and a user study. In the first experiment we specify, detect and extract minimal executable transformations of three refactorings across different open-source projects. In the second experiment we use our approach in two industrial projects, in the context of commit untangling. The approach is applied to extract all instances of a similar modification occurring in a systematic edit. The user study was performed to evaluate the usability and expressiveness of our approach's query language.

This paper is an extension of our paper published at the 24th edition of the "International Conference on Software Analysis, Evolution and Reengineering" (Stevens and De Roover 2017). The following is a summary of the main changes that were made:

- The second experiment involving two industrial projects was added. Evolution queries are used to describe the effect of a systematic edit. Executing such a query results in a minimal, executable edit script, which effectively untangles the systematic edit from all other changes in the commit.
- The user study was added to assess the query language's usability and expressiveness.
- Additional visualizations of the change dependency graph of each experiment have been added. These visualizations provide better insight into the topology of these graphs, and how strongly connected changes depend on each other. They also depict the changes that are part of the solution, illustrating the dependencies between solution and non-solution changes.
- Finally, additional detail has been added to chapters 3 and 4, which respectively define fine-grained changes and describe our approach.

2 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 change 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 the Ekeko logic program query language, developed by De Roover and Stevens (2014).

Figure 3 depicts an overview of our approach. Two subsequent revisions of the same file, called Rev1 and Rev2, are given as input. Note that, if the user is interested in examining a range of revisions for a file, the approach can be applied multiple times on each pair of subsequent revisions in this range. The goal of our approach is to detect instances of a user-specified evolution in the changes between Rev1 and Rev2. To this end, the files are passed as inputs to a change distiller called ChangeNodes, detailed in Section 3. 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 4.2 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.

2.1 Ekeko, A Declarative Program Querying Language

In order to specify the source code characteristics intermediate AST should exhibit we use the Ekeko program query language by De Roover and Stevens (2014). Ekeko is a Clojure library for applicative logic meta-programming against an Eclipse workspace. The library is primarily targeted to querying and transforming Java code, but other language bindings (e.g. AspectJ and PHP)

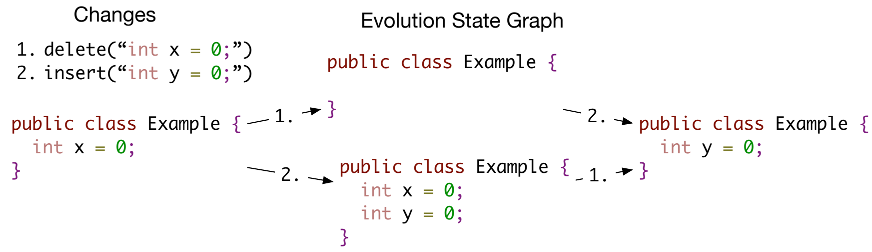

Fig. 2: Evolution state graph (ESG) for the middle sequence of distilled changes of Figure 1. Edge labels correspond to applied changes.

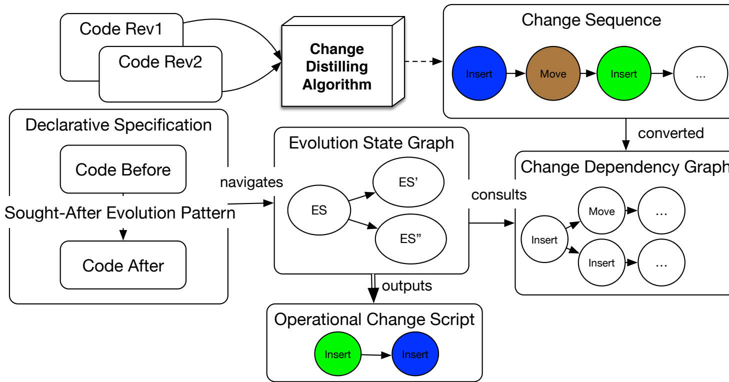

Fig. 3: Graphical overview of the approach: two code revisions are converted to a change dependency graph. An evolution query, provided in a declarative specification, consults this change dependency graph to produce an operational change script that implements the desired evolution pattern.

have been implemented as well. Ekeko features a variety of different basic logic predicates to query source code: this ranges from structural predicates (e.g. to relate classes to subclasses, classes to their members, variable uses to declarations, ...), to control flow predicates (to navigate control-flow graphs) and data flow predicates (e.g. to perform alias analysis). The latter two types of predicates are provided by Ekeko's integration with the Soot framework for bytecode analysis by Vallée-Rai et al (1999). Additional predicates can be implemented by composing existing ones, or by interacting directly with the Eclipse JDT and/or Soot framework.

The use of Ekeko in our approach does entail that users need to have some basic knowledge on logic programming, and on Eclipse's AST representation of Java source code. We consider this a reasonable requirement, as our approach is primarily targeted at (MSR) researchers.

Several EKEKO predicates are used 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 unqualified name of `?node`'s AST node type. 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.

2.2 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 declaration `?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 and the type of the field using the EKEKO predicate `ast-field|absent`.

As this query was used as an initial example to introduce the syntax of evolution queries, it does not yet take into account the fact that any field accesses need to be renamed as well. A more refined version of this query, which does consider field accesses, is presented later in Section 5.2.3.

Finally, 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

[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.

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]
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.

2.3 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.

2.3.1 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 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.

2.3.2 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 3. We assume that the code clone has already been detected using an existing tool such as CCFinder Kamiya et al (2002), and that the ESG has been created. We are only tasked with finding the specific changes that implemented the clone removal.

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

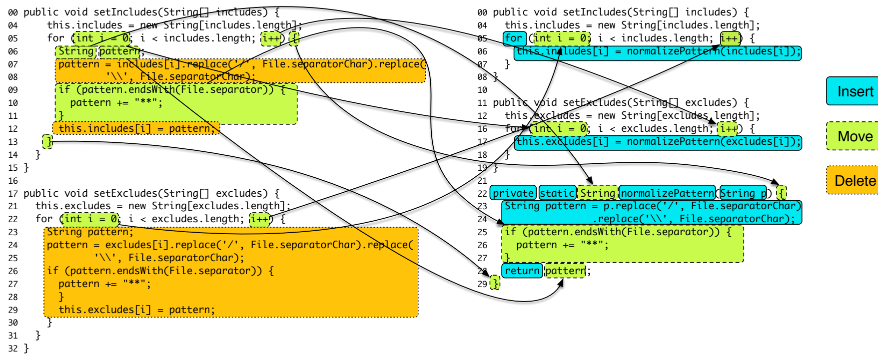

Fig. 6: Two revisions of class from the ANT project in between which a code clone is extracted into a separate method `normalizePattern`, to which two invocations are added. Overlaid is the output of our ChangeNodes change distilling algorithm.

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.

```
1 (defn clone-elimination [esg nameA nameB extracted]
2   (run* [?es]
3     (query-changes esg ?es
4       [?cloneA ?cloneB ?extracted ?aInvoc ?bInvoc ?aCurr ?bCurr]
5       (in-current-es [es ast]
6         (child+ ast ?cloneA)
7         (child+ ast ?cloneB)
8         (method-string|named ?cloneA nameA)
9         (method-string|named ?cloneB nameB))
10      change->+
11      (in-current-es [es ast]
12        (ast :MethodDeclaration ?extracted)
13        (ast-ast|same ?extracted extracted)
14        (ast-method-method|corresponding ast ?cloneA ?aCurr)
15        (ast-method-method|corresponding ast ?cloneB ?bCurr)
16        (child+ ?aCurr ?aInvoc)
17        (child+ ?bCurr ?bInvoc)
18        (method-invocation|invokes ?extracted ?aInvoc)
19        (method-invocation|invokes ?extracted ?bInvoc)))))))
```

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

3 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 (Christophe et al 2014; Stevens 2015). At its heart lies the algorithm presented by Chawathe et al. Chawathe et al (1996), on which the ChangeDistiller (Fluri et al 2007) 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 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 are also

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

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 made. 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',removed',property,index)`
A `node'` is inserted at location `property` of node `parent'`. In case `property` is a child list property, the node is inserted at index `index`. Applying the insert will only add a minimal representation of `node'` to `parent'`. Node `original` is the parent in the original AST, and can be `null` if the insert is part of the subtree introduced by another change operation. Finally, `removed'` is the node that the insert operation overwrote in case a node was already present in `original` at location `property`. If such a node was present its child nodes are added to `node'` so that later change operations can use them. If no such node was present this value is `null`. During the execution of the algorithm the removed node is still accessible through the matching data structure. For example, the overwritten node can still be moved to a different location.

`move(node',original,parent',preparent',property,index)`
A `node'` is moved from `preparent'` to location `property` of node `parent'`. In case `property` is a list property, the node is moved to index `index`. Only a minimal representation of the node will be moved. Its original location is replaced by a placeholder node that still contains the subtree located at `node'`. Thus, only the node is moved, and not the node and its complete subtree. `original` is the representation of `node'` in the original AST.

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

`delete(node',original,parent',property,index)`
A node `node'` and its complete subtree are removed at the value of `property` in `parent'`. 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. As such, `node'` will not have a parent node. The parent node before the application of the delete is captured by `parent'`. `original` is the representation of `node'` in the original AST.

Both a move and an 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.

4 Conceptual Implementation

Having presented the necessary background on changes and change distilling, we are ready to present our approach in more detail. 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 1.

An Evolution State Graph (ESG) is constructed, against which our approach evaluates evolution queries. The evolution queries themselves feature *regular path expressions* (de Moor et al 2002; Liu et al 2004) 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 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.

Like the Ekeko logic metaprogramming library that is used to evaluate source code characteristics, the evaluation of these regular path expressions is also implemented using logic programming. That is, finding matches of a regular path expression in a graph is expressed as a unification problem: the ESG itself, source code characteristics and the evolution query are all described in a declarative manner, as a database of facts and relations. The unification algorithm (of the core.logic library we used) is only tasked to find the possible values for each logic variable in the query such that all facts and relations hold.

Table 1 provides an overview of the constructs/relations that can be used in our regular path expressions. We provide constructs for navigating an ESG, such as `change->+` which moves evaluation to either a direct or indirect successor of the current node in the ESG. Keep in mind that an ESG is a directed acyclic graph, which implies that all outgoing edges of an ES point to its successor evolution states. 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.

4.1 Usage of the evolution query language

Before presenting the construction of CDGs and ESGs, we will briefly discuss the language constructs in Table 1, to give users of our approach a better idea on how to make use of these constructs to write evolution queries.

Table 1: Language for specifying evolution patterns through ESG-navigating regular path expressions.

Navigation through the ESG	
`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 changes the current state by applying an arbitrary, including zero, number of changes and their *dependent* changes.
`change==>+`	`change==>+` is similar to `change==>*`, except that at least one change will be applied.
Characteristics of an ES	
`(in-current-es [node ast] & goals)`	`in-current-es` binds `es` to the current evolution state 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.

The `query-changes` construct is needed to launch an evolution query. The query itself, i.e. the `goals` parameter of `query-changes`, consists of a sequence of `in-current-es` constructs that are interposed with navigation constructs. The `in-current-es` construct is used to reason about properties of the current evolution state, and navigation constructs are used to move the current evolution state within an ESG. Informally, an evolution query describes a sequence of evolution states that are of interest, where multiple changes can take place between these states.

Most commonly, there are only two states that are of interest, one describing a "before" situation and one describing the "after" situation. This is the case for all queries that are shown in this paper. However, we would like to mention that some queries do involve more than two evolution states of interest. For instance, a query to detect reverted bug fixes involves three states: one state in which the bug is present, one where it is fixed, and one where the fix is reverted. Another example with several evolution states involves detecting refactoring plans (Pérez 2013), i.e. a specific sequence of refactorings that is applied towards e.g. removing a particular code smell.

The most common navigation constructs are `change->*` and `change->+`. Both of these move the current evolution state by applying multiple changes, until an evolution state is found that matches description of the next `in-current-es` in the query.

Note that Table 1 does not provide a navigation construct that applies exactly one change. There is little use for such a construct in our current approach, as it effectively forbids any other changes from taking place between two states of interest. Given that our ESG is constructed from distilled changes, it considers all possible sequences to transform one AST into an-

other. Consequently, in between two states of interest, it is likely that other changes can be applied that are irrelevant to the query. This is not an issue for `change->*` and `change->+`, as they can apply any number of changes.

Finally, there are the `change==>` and `change==>*` constructs, which respectively are variants of `change->` and `change->*` that also apply any dependent changes. These two constructs are implemented as a more coarse-grained means to navigate an ESG, as they reduce how often `in-current-es` constructs need to be executed to test specific properties for the current state. Because some EKEKO predicates to describe the desired properties of an ES are more computationally expensive than others, `change==>` and `change==>*` can be used to increase runtime performance, considering the trade-off that not all solutions may be found.

4.2 Construction of a Change Dependency Graph

Section 4.3 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.

4.3 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 is a directed acyclic graph that 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.

5 Evaluation - Change extraction experiments

After presenting our approach, we now seek to evaluate our approach in two different ways. First, this section focuses on evaluating whether our approach works as intended, and to gain insights into the output produced by the tool. This is done by means of two experiments that extract changes in existing data sets. Second, we also evaluate the usability and expressiveness of our query language by means of a user study. This second part of the evaluation will only be discussed in the next section (Sec. 6). In this section, we will focus first on answering the following five 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 the structure of the CDG relate to the approach's runtime performance?
RQ4 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?
RQ5 In the solutions produced by the approach, how many changes in the solution are directly relevant to the evolution query, compared to the other changes in the solution?

The motivation behind these questions is as follows: RQ1 and RQ2 are meant to verify that our approach works as intended: there can be multiple solutions to an evolution query. The shortest solution effectively cannot be minimized any further, while remaining an executable change sequence. RQ3 was added to gain insight into the main factors that affect the approach's performance. We hypothesize the structure of CDG plays a large role to determine to what extent the approach scales to complex changes. RQ4 and RQ5 are intended to examine the solutions produced by the approach in more detail. That is, RQ4 compares our approach to a naive one, where the solution would include *all* changes between the first ES, and the ES where the full evolution pattern is found. Such a solution would also include any changes irrelevant to the evolution pattern, i.e. not of interest to the user. RQ5 looks into the solutions produced by our approach, to get a better idea how many changes are of interest to the user, i.e. how many changes directly contribute to the evolution pattern.

To answer these five questions, we perform two experiments, each of which considers a different context to demonstrate that our approach can be applied in multiple applications. In the first experiment, we extract changes that perform a well-known refactoring from commits of open-source projects. In the second experiment we extract changes that represent a systematic edit, i.e. a group of similar changes, from commits of two industrial projects.

In each experiment, we will attempt to specify the state of the source code before and after the sought-after transformation by means of a declarative evolution query (Section 5.2). It is known in advance which evolution patterns should be found in each experiment, so RQ1 can be answered by verifying that our approach does find the expected patterns. Each solution to such a query should be an executable script of changes. When executed, this script will transform the source code *before the commit* to a state that matches the specified state of the code *after the transformation*. In other words, the extracted changes will perform only the transformation specified in the query. The remaining changes (those not included in the script) in the commit will, when executed, transform the state of the code *after the script's transformation* to the state of the code *after the entire commit*. Part of answering RQ2 consists of effectively executing this script, followed by the remaining changes, to verify that this execution is valid does not produce any errors. If the execution were invalid, this could be caused, for instance, by a change that tries to delete a node that does not exist.

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. To complete the answer to RQ2, we will manually verify whether the solutions (depicted in Table 5) are minimal. We also compute various metrics pertaining to each research question (Section 5.5).

To answer RQ3, RQ4 and RQ5, various metrics are measured and discussed regarding the structure of the change dependency graph, the solutions produced by the approach, and the time taken to run an evolution query.

Finally, note that in these experiments, we do not consider a notion of false positive (or negative) results. For instance, in the first experiment we consider extracting specific refactorings from changes. It is well possible that, given some evolution query, the query may not describe all refactorings that the writer of that query intended to find, which can be considered false positives. However, this is not due to evaluation process of evolution queries are processed, as it considers all possible change sequences produced by a change distiller. Instead, this simply indicates that the evolution query itself needs to be modified, such that all desired results are described. Whether or not the current constructs in evolution queries are sufficiently expressive to accurately describe any desired evolution patterns is only illustrated with examples in this paper, but not evaluated in these experiments.

In the remainder of this chapter, we will first describe the data set and the queries that specify the sought-after transformations for each experiment, followed by the experiments' results and discussion.

5.1 Data Set - Refactoring Extraction Experiment

Our evaluation of the first experiment 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 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. Additionally, we would like to note that each of these refactorings represents a source code transformation within one file. This is due to the fact that evolution queries currently reason about the changes in one file. Supporting multiple files is left as future work, and is discussed in more detail in Section 7.

Table 2 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 subscript in the first column. We have used two such oracles:

- The first oracle, denoted by a $_1$ subscript in Table 2, corresponds to a data set[5] produced by the REF-FINDER (Prete et al 2010) tool which recognizes refac-

[5] `http://web.cs.ucla.edu/~miryung/inspected_dataset.zip`

Table 2: Data set of the refactoring extraction experiment

Id	Ref.	Project	Commit	Class
1	Constant_1	ant	d97f4f3	`WeblogicDeploymentTool`
2	Constant_1	ant	34dc512	`Jar`
3	Constant_1	ant	a794b2b	`FixCRLF`
4	Constant_1	JMeter	b57a7b3	`AuthPanel`
5	Constant_1	JMeter	3a53a0a	`HTTPSampler`
6	Constant_1	JMeter	8275917	`HTTPSampler`
7	Method_2	jdt.ui	678	`JavaEditor`
8	Method_2	jdt.ui	2910	`JavaNavigatorContentProvider`
9	Method_2	jdt.ui	2722	`StubUtility2`
10	Field_2	jdt.ui.test	0277	`MarkerResolutionTest`
11	Field_2	jdt.ui	2810	`SourceAnalyzer`
12	Field_2	jdt.ui	2810	`SourceProvider`
13	Field_2	jdt.ui	2810	`InlineMethodRefactoring`

torings 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 5 are all such commits in the RF data set.

– The second oracle, denoted by a $_2$ subscript, corresponds to a data set[6] resulting from a study by Murphy-Hill et al (2012) of logs of developer interactions 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 "*Rename Field*" refactorings in Table 5.

Note that the commit identifiers listed in Table 2 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.

5.2 Queries - Refactoring Extraction Experiment

We describe the queries used to identify the exact changes contributing to the "replace magic constant", "remove unused method" and "rename field" refactoring. The query results of this experiment, as well as the second experiment, will be discussed later in section 5.5.

5.2.1 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

[6] `http://multiview.cs.pdx.edu/refactoring/experiments/`

```
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.

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 one or more 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 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.

5.2.2 Query for "Remove Unused Method"

Figure 9 depicts the query that identifies changes implementing a "*Remove Unused Method*" refactoring. The query describes an initial Evolution State containing an unused method, and a later Evolution State in which the method is no longer present. Lines 3–6 describe the initial ES, in which method `?method` is unused. Line 6 uses `method|unused/1`, which implements a straightforward name-based resolution mechanism to verify that ES does not contain an invocation of this method. Line 7 applies one or more changes using `change->+`. Lines 8–10 describe a successive ES, in which no method with the same name as the name of `?method` can be found. It also ensures that there is no call introduced to the removed method.

```
1 (query-changes esg ?end
2   [?method]
3   (in-current-es [es ast]
4     (child+ ast ?method)
5     (ast :MethodDeclaration ?method)
6     (method|unused ?method))
7   change->+
8   (in-current-es [es ast]
9     (ast-method|absent ast ?method)
10    (method|unused ?method)))
```

Fig. 9: Evolution query for those changes in a commit that implement a *"Remove Unused Method"* 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. 10: Evolution query for those changes in a commit that implement a *"Rename Field"* refactoring.

5.2.3 Query for "Rename Field"

Figure 10 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 one or more 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.

Table 3: Data of the second commit untangling experiment

Id	Sys. Edit	Project	#Inst.
14	Introduce runnable	TP Vision	7
15	Remove cast	FPS Finance	37
16	Introduce fields and accessors	FPS Finance	21

5.3 Data Set - Untangling Systematic Edits Experiment

The second data set of our evaluation comes from the code base of two industrial partners. The first partner is TP Vision. 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. The second partner is FPS Finance, the federal public service of finances in Belgium. It carries out various tasks in field of taxes and finances, such as ensuring correct taxation is levied in due time, supervising flows of goods, maintaining public property records and preventing fraud.

Part of the data set stems from a previous study in which we used frequent itemset mining on the output of ChangeNodes to detect instances of similar groups of changes, also called systematic edits (Molderez et al 2017). In this evaluation we aim to extract all the instances of such a systematic edit in order to untangle it from that commit. Note that the tool used in this study was configured such that changes were grouped by method, and only one instance of a systematic edit can be found per group. This implies that instances found by the study are not larger than a method. Given this configuration, we selected three of the most complex systematic edits for this experiment.

The three cases studied in this experiment are listed in Table 3. The columns respectively represent describe the extracted systematic edit, in which project it is located, and how many instances of the systematic edit are present.

Similar to the first experiment, we extract an executable change sequence that, when applied, only the systematic edit is performed. That is, the code is transformed such that the different instances of the systematic edit are present. In order to untangle the commit we first create a commit in which only the systematic edit has been applied, and a second final commit containing the remaining changes.

5.4 Queries - Untangling Systematic Edits Experiment

In this section we describe the queries used to identify the exact changes contributing to all the instances of the “introduce runnable”, “remove cast” and “introduce field and accessors” systematic edits.

```
1     - optionsfocuschange(OPTIONS_SHOW_ALL);
   1  + new Handler().post(new Runnable() {
   2  +   @Override
   3  +   public void run() {
   4  +     optionsfocuschange(OPTIONS_SHOW_ALL);
   5  +   }
   6  + }
```

Fig. 11: Figure depicting a single instance of the "introduce runnable" systematic edit, in which a call to `optionfocuschange` is wrapped in a `Runnable`.

```
1 (let [target (:target esg)]
2   (query-changes esg ?es
3     change->*
4         (in-current-es [es ast]
5           (ast-methodinvoc|all-optionfocus ast ?options)
6           (ast-methodinvoc|all-optionfocus target ?all-options)
7           (list-list|same-length ?options ?all-options)
8           (fails
9                 (fresh [?focus]
10                  (membero ?option ?options)
11                  (optionsfocus|not-wrapped-in-runnable ?option))))))
```

Fig. 12: Evolution query for those changes in a commit that are an instance of the "Introduce Runnable" systematic edit.

```
1 (defn ast-methodinvoc|all-optionfocus [?node]
2   (fresh [?name]
3     (ast :MethodInvocation ?node)
4     (has :name ?node ?name)
5     (name|simple-string ?name "optionsfocuschange")))

6 (defn optionsfocus|not-wrapped-in-runnable [?options]
7   (all
8     (methodinvocation|optionsfocus ?options)
9     (fails
10      (fresh [?expr ?block ?run ?anonymous ?class]
11        (ast-methodinvoc|all-optionfocus ?options)
12        (ast-parent ?options ?expr)
13        (ast-parent ?expr ?block)
14        (ast-parent ?block ?run)
15        (methoddeclaration|name ?run "run")
16        (ast-parent ?run ?anonymous)
17        (ast-parent ?anonymous ?class)
18        (classinstance|name ?class "Runnable")))))
```

Fig. 13: Additional EKEKO predicates used by the evolution query

5.4.1 Query for the "Introduce Runnable" Systematic Edit

Figure 11 depicts one instance of the "introduce runnable" systematic edit. Each instance of this edit modifies a call to `optionfocuschange` so that is wrapped inside a `Runnable` such that the call becomes asynchronous.

Figure 12 depicts the evolution query that detects the changes that contribute to all the instances of this systematic edit. The first line introduces

a variable `target`, bound to the the AST of the target commit. Lines 2–11 launch an evolution query in which an arbitrary number of changes are applied until an ES is encountered that contains the same amount of calls to `optionfocuschange` as the target source code. This is specified by lines 5–7. Lines 8–11 specify that every call to `optionfocuschange` must be wrapped in a `Runnable`. To this end, the query uses negation-as-failure to ensure no call exists that is not wrapped in a `Runnable`. Finally, note that two new EKEKO predicates were defined, specifically for this evolution query, `ast-methodinvoc|all-optionfocus` and `optionsfocus|not-wrapped-in-runnable`. Just to illustrate their definitions, they are given in Figure 13. In short, both of these predicates are composed of basic EKEKO predicates to navigate an AST, or check that an AST node has a specific name or type.

5.4.2 Query for the "Remove Cast" Systematic Edit

Figure 14 depicts an instance of the "remove cast" systematic edit. Each instance removes the cast to type `Long` for the return value of `dto.getCodeValue`. Figure 15 specifies the evolution query that detects changes that contribute to all the instances of this systematic edit. It works similar to the previous query. It uses a custom predicate `ast-invocs|getCodeValue` (composed of basic EKEKO predicates) to unify `?all-invocs` with all the invocations of `getCodeValue` in the target AST, and ensures that the sought-after ES has the same amount of invocations. Lines 9–11 ensure that no call to `getCodeValue` is wrapped in a cast expression.

```
1   - if (Long.valueOf((Long)dto.getCodeValue("A8476")) > 0) {
2   -   bcdMap.put("100",Long.valueOf((Long)dto.getCodeValue("A8476")));
  1 + if (dto.getCodeValue("A8476") > 0) {
  2 +   bcdMap.put("100",dto.getCodeValue("A8476"));
```

Fig. 14: Figure depicting a single instance of the systematic edit, in which the value of a call to `dto.getCodeValue` is no longer cast to type `Long`.

5.4.3 Query for "Introduce Field and Accessors" Systematic Edit

Each instance of the final systematic edit introduces a new field, together with a getter and a setter. This systematic edit occurs in practice whenever the code of FPS FINANCE needs to be updated to reflect new tax laws, and several fields are introduced at once to represent new or updated entries in tax forms. Figure 16 depicts the evolution query that detects changes that contribute to all instances of this systematic edit. Lines 1–2 introduce two variables `original` and `target`, bound respectively to the original and target AST. Lines 3–15 specify the evolution query, which applies changes

```
1 (let [target (:target esg)]
2   (query-changes esg ?es
3     change->*
4     (in-current-es [es ast]
5       (fresh [?all-invocs ?invocs ?invoc]
6         (ast-invocs|getCodeValue target ?all-invocs)
7         (ast-invocs|getCodeValue ast ?invocs)
8         (list-list|same-length ?all-invocs ?invocs)
9         (fails
10          (membero ?invoc ?invocs)
11          (getcodevalue|wrapped-in-cast ?invoc))))))
```

Fig. 15: Evolution query for those changes in a commit that are an instance of the "Remove Cast" systematic edit.

```
1 (let [original (:original esg)
2       target (:target esg)]
3   (query-changes esg ?es
4     change->*
5     (in-current-es [es ast]
6       (fresh [?all-fields ?fields ?field ?getter ?setter]
7         (ast-ast-fields|introduced original target ?all-fields)
8         (ast-ast-fields|introduced original ast ?fields)
9         (list-list|same-length ?all-fields ?fields)
10        (fails
11          (membero ?field ?fields)
12          (child+-type curr :MethodDeclaration ?getter)
13          (field-declaration-method|getter ?field ?getter)
14          (child+-type curr :MethodDeclaration ?setter)
15          (field-declaration-method|setter ?field ?setter))))))
```

Fig. 16: Evolution query for those changes in a commit that are an instance of the "Introduce Field and Accessors" systematic edit.

until an ES is encountered that has as many introduced fields as the target AST, and in which each field has a corresponding getter and setter. Lines 7–9 ensure that the ES has the same number of introduced fields. The predicate `ast-ast-fields|introduced` unifies its last argument with a list of fields that are present in its second argument, but not in its first. Lines 10–15 use a negation-as-failure loop to ensure that every introduced field has a getter `?getter` and `?setter`. Predicates `field-declaration-method|getter` and `field-declaration-method|setter` unify their second argument with a method that adhere to the convention of getters and setters.

5.5 Query Results

After presenting the queries of our experiments, we can now discuss their results. We will first provide an overview of the results, while answering the five research questions. After this, each of the experiments' results is discussed in more detail, illustrated with visualizations of their CDGs.

RQ1 and RQ2 - The first two research questions, RQ1 and RQ2, can be answered affirmatively without showing any data. For RQ1, we have executed each evolution query and can conclude that in each case, we can successfully

use a single evolution query to identify the same pattern in different change sequences created from source code that exhibits the desired pattern. For RQ2, we manually verified that the solutions are both minimal and executable. To ensure that a solution is minimal, it is necessary to distinguish between changes that are relevant to the evolution query, and those that are irrelevant. This is explained in more detail during the discussion of RQ5.

To answer the remaining RQs, we will discuss the results presented in Table 4 and Table 5. The Id column in both of these tables also refers back to the tables describing the experiments' data sets, Table 2 and Table 3. Id 1-13 corresponds to the first experiment and Id 14-16 corresponds to the second experiment.

RQ3 - Table 4 depicts metrics about the distilled changes and the corresponding change dependency graph (CDG), and can be used to answer RQ3. Column *#Ch* shows the total number of distilled changes for the file. Next, column *LP* shows the length of the longest path through the CDG. Column *MP* shows the median length of the paths through the CDG. Both indicate, using our approach, how many changes need to be applied before a given change becomes applicable. Were the output of a change distiller used directly, this would be all of the preceding changes in the distilled sequence. The last columns show how connected the graph is: column *#Co* shows the number of connected components in the CDG. Changes from one component can only be connected with changes from the same component. Column *#Single* shows the number of components that contain only a single node. Columns *MaxIn* and *MIn* show respectively the maximum and median in-degree of the CDG (i.e., the number of changes depending on a change). Finally, columns *MaxOut* and *MOut* respectively show the maximum and median out-degree the CDG (i.e., the number of changes that a specific change depends on).

The data in Table 4 concerns RQ3, which is about the relation between a CDG's structure and the running time of an evolution query. The total running time of an evolution query can be found in the last column of Table 5. Note that the process of change distilling and constructing the CDG is also included in this running time. While not shown in the tables, change distilling and CDG construction only needs to be executed once and consumes a relatively smaller portion of the total running time, especially when a large CDG is involved. Running the evolution query itself involves applying many possible combinations of path sequences, while testing properties on evolution states. As such, our main intuition is that the running time mainly increases together with the number of changes in the CDG. This appears to be true in most cases, looking at the #Ch column. However, Id 6, 7 and 12 each have a low number of changes, but a relatively high runtime. Knowing this, we suspect the number of connected components also is an important factor that influences the runtime. The #Co column seems to confirm this, although this can only be considered a rough indication, as we did not look into this any further. The intuition behind this conclusion is that, with many connected components, there are many changes that can be performed independently. This greatly increases the number of possible change sequences, all of which are explored

Table 4: Data describing the distilled changes and their corresponding change dependency graph (CDG). The identifier (Id) column corresponds to the systematic edits in Table 2 and Table 3.

Id	#Ch	LP	MP	#Co	#Single	MaxIn	MIn	MaxOut	MOut
	Refactoring experiment (Section 5.1)								
1	202	14	8	10	4	16	1	10	1
2	74	10	3	5	4	33	1	6	2
3	1244	20	6	2	1	235	1	49	2
4	245	10	4	17	15	106	1	11	1
5	149	7	2	36	18	11	1	6	1
6	25	5	2	4	0	5	1	4	1
7	11	2	1	10	9	1	1	1	1
8	5	1	1	5	5	0	0	0	0
9	291	8	2	28	24	91	1	10	1
10	10	4	2.5	5	4	1	1	3	1
11	63	7	2	25	22	1	1	6	2
12	27	6	2	12	10	3	1	3	2
13	221	14	3	41	33	41	1	9	2
	Untangling experiment (Section 5.3)								
14	388	10	3	54	31	27	1	11	1
15	433	29	2	101	51	41	1	10	1
16	315	4	2	63	0	4	0	1	1

when running an evolution query. In short, our answer to RQ3 is that these data indicate the #Ch and #Co are the main factors that influence running time. While we cannot affect the number of changes, it may be an interesting path of future work to exploit the fact that groups of changes are independent to reduce the runtime.

RQ4 and RQ5 - Table 5 describes the solutions and its changes, and can be used to answer RQ4 and RQ5. Column *#Sol* depicts the number of changes in the minimal, executable solution returned by the query. Next, Columns *LS* and *MS* respectively depict the longest and median span, indicating the number of changes that separate two successive changes in the solution. Column *#DS* depicts the total number of changes that would need to be applied were the distilled output queried directly, before the described evolution pattern would be recognized. Thus, the columns *LS*, *MS* and *#DS* indicate how many irrelevant changes would need to be applied when **not** using our approach, while *#Sol* indicates the total number of changes that actually need to be applied using our approach.

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 (EI) 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 implementing.

Evolution Supporting (ES) 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

Table 5: Description of the evaluation's solutions. The first column shows the identifier. The next seven columns describe the minimal solution and its changes. The last column describes the running time of the computation of the result.

Id	#Sol	LS	MS	#DS	#EI	#ES	#EL	Time(s)
	Refactoring experiment (Section 5.1)							
1	8	29	8	82	4	4	0	33
2	4	11	7	23	4	0	0	19
3	10	300	68	1000	4	6	0	2054
4	5	135	23	199	4	1	0	106
5	5	96	8.5	118	4	1	0	1985
6	6	4	3	14	4	1	1	285
7	2	1	0.5	1	1	0	1	748
8	1	0	0	0	1	0	0	3
9	39	22	4.5	264	2	0	37	1536
10	3	3	2	9	3	0	0	53
11	13	13	3	57	13	0	0	8842
12	15	3	1	24	10	5	0	3133
13	6	77	29	213	6	0	0	5757
	Untangling experiment (Section 5.3)							
14	51	29	2	338	49	2	0	488
15	90	38	2	385	79	9	2	2247
16	314	1	1	314	314	0	0	3708

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 (EL) 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 running time in seconds for distilling the changes, constructing the CDG, and finding a single minimal solution. This is the running time of a single execution of an evolution query, and only serves to provide the general order of magnitude of the running time.

Figure 17 illustrates this classification of the changes in the solution to the "*Replace Magic Constant*" query against commit `8275917` (Id 6). Before the commit, method `getUrl()` contained the constant `0` twice: once as a magic constant on line 8, and once as part of a check for an empty list on line 5. In the depicted solution[7], change 1 inserts a field "`private static int UNSPECIFIED_PORT;`",

[7] Keep in mind that the approach currently only looks for minimal, executable solutions. As illustrated in Figure 17, the approach disregards whether or not a distilled change sequence is realistic, i.e. a sequence that a developer might intuitively consider to transform one piece of code into another.

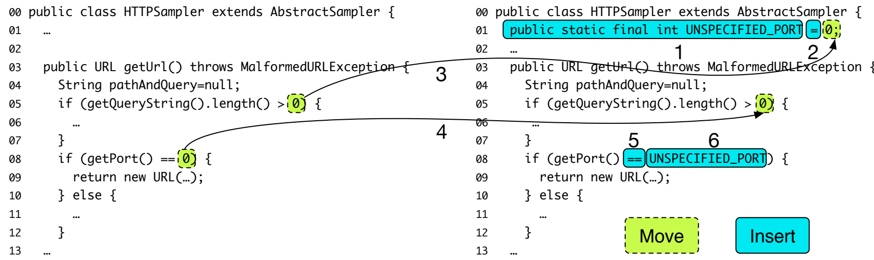

Fig. 17: Code snippet from the `HTTPSampler` class, in which a new field is introduced (1,2). This field is initialized via a move of a constant (3), which itself is replaced by a different move (4). Move 4 is an EL change as its node-to-be-moved is overwritten by a later insert (6). Insert 5, unnecessarily, overwrites the parent location of change 6, and is classified as an ES change.

change 2 inserts an initializer expression "`...= 0;`" 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 3). 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*.

After presenting Table 5 in detail, we can now answer **RQ4**, which is concerned with comparing our approach to finding minimal sequences, with a naive approach that also includes irrelevant changes in its solutions. From the LS, MS and #DS columns – 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.

Finally, the answer of **RQ5** is concerned with the ratio of EI changes compared to ES and EL changes. As Table 5 indicates, in most cases, nearly

all changes are directly related to the sought-after evolution pattern. Consequently, we conclude that most solutions would also be perceived as minimal by users of our approach. This is meant in the sense that users may not be interested in any ES or EL changes, even though these changes are necessary to obtain a solution that is executable.

Having discussed each of the different research questions, the remainder of this section discusses each of the evolution queries' results in more detail.

Results for "Replace Magic Constant": For each of the refactoring commits from projects `ant` and `JMeter` (Id 1-6), 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. We already explained the EL change in the minimal solution for commit `8275917` (Id 6) above using Figure 17.

Note that the size of the change sequence distilled for the different commits containing this refactoring varies wildly, as does their complexity. As demonstrated by columns #Sol and #DS, and Figure 18, the CDG reduces the number of changes that need to be applied for any given change significantly. Thus, the returned minimal solutions always consist out of a very small number of changes. The minimal solution for commit `a794b2b` (Id 3), for example, includes 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 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.

Each of the minimal solutions can be replayed on the source code before the commit. However, doing so might not eliminate every copy of the magic constant. This is because our specified query is somewhat too relaxed. Its minimal solution only needs to encompass the changes that eliminate a single copy of the constant, leaving the changes that eliminate the remaining copies behind. The query could be improved by, for instance, requiring that the final evolution state has as many accesses to the newly introduced field as there were copies of the constant. The danger of making queries this strict is that some instances of the evolution pattern will no longer be recognized. This would already be the case for the commit in Figure 17, where the same constant is used for two different purposes.

Figure 18 depicts the CDG created for commit `34dc512` (Id 1) of the `ant` project. Figure 19 depicts the CDG created for commit `3a53a0a` (Id 5) of the `JMeter` project. Every distilled change corresponds to a node in the graph. Dependent changes are connected through an edge. The colors of the node indicate the change type. Changes that are part of the minimal solution are depicted as a pentagon. Figure 18 demonstrates that inserting blocks of code

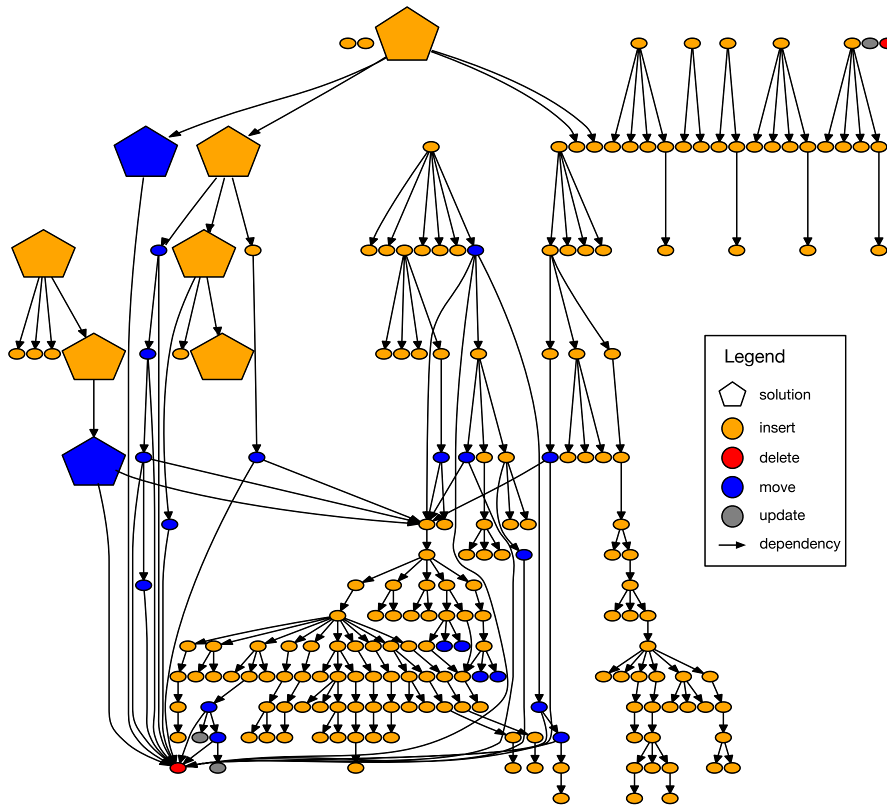

Fig. 18: CDG created for the commit with Id 1 in Table 4.

results in many parent dependencies. This is due to the minimal representation of change operations (cf., Section 3). Both figures illustrate that the CDGs have several connected components, with changes that can be applied independently from each other.

Figure 20 depicts the CDG created for commit `a794b2b` (Id 3) of the `ant` project. This figure illustrates the complexity of some of the CDGs. The high number of distilled changes results in a complex CDG, where a manual detection of the dependencies is not feasible.

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 (Id 7-9) we note that this only holds for a single case. The returned solution for `StubUtility2` (Id 9) 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

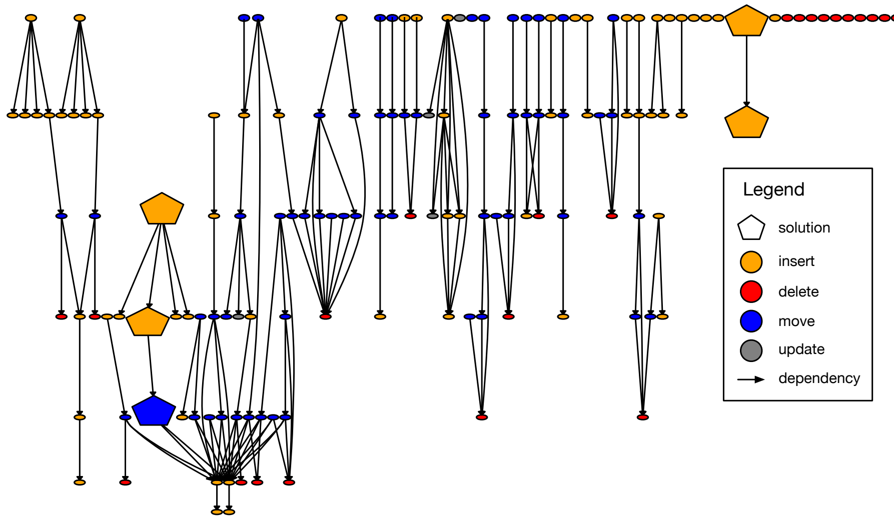

Fig. 19: CDG created for the commit with Id 5 in Table 4.

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.

Figure 21 depicts the CDG created for the `StubUtility2` class (Id 9). This figure clearly illustrates the evolution linking move operations (i.e., the blue pentagons), and the single evolution implementing delete operation (i.e., the red pentagon). The nodes-to-be-moved are all part of the subtree that will be removed by the delete. Thus, the moves must be applied before the delete. Two similar delete operations are present; one on the left side of the figure that is not part of the solution and one in the top right corner of the figure that is part of the solution. Each delete that is part of the solution removes one of the two methods that share their name.

Results for "Rename Field": The final results of our first experiment are for the "Rename Field" refactoring (Id 10-13). 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 running times 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. Detecting the absence of an element requires visiting all the nodes in the AST to ensure that the element is not present, which is a slow process.

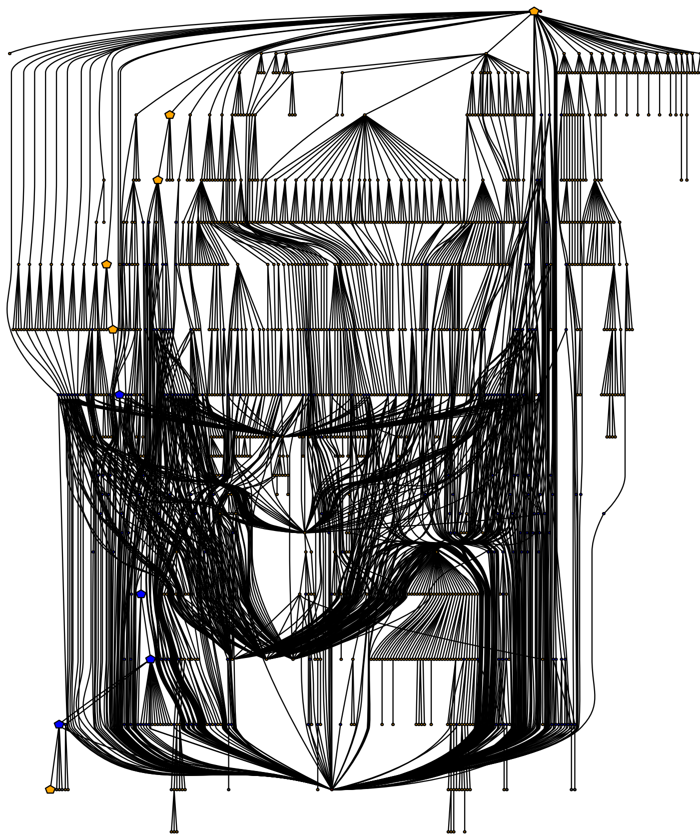

Fig. 20: CDG created for the commit with Id 3 in Table 4.

Figure 22 depicts the CDG created for the `InlineMethodRefactoring` class. This figure depicts the different evolution implementing updates that modify the accesses to the the renamed field.

Results for "Introduce Runnable": The solution for the "Introduce Runnable" systematic edit (Id 14) consists of all but two EI changes. This means that applying the solution results in a a new commit that only contains the result of the systematic edit. The two ES changes are due to a new call to `optionfocuschange` that was added in the target source code. This call is added in a new method, which is introduced by the two ES changes.

Figure 25 depicts the CDG created for these changes, and illustrates this. There are 6 groups of changes that are part of the solution on the right hand side of this figure. These represent the existing calls that are wrapped in a `Runnable`. The group of changes that are part of the solution on the left are the two ES changes that introduce a new method, and several changes that introduce the `Runnable` and call to `optionsfocuschange`.

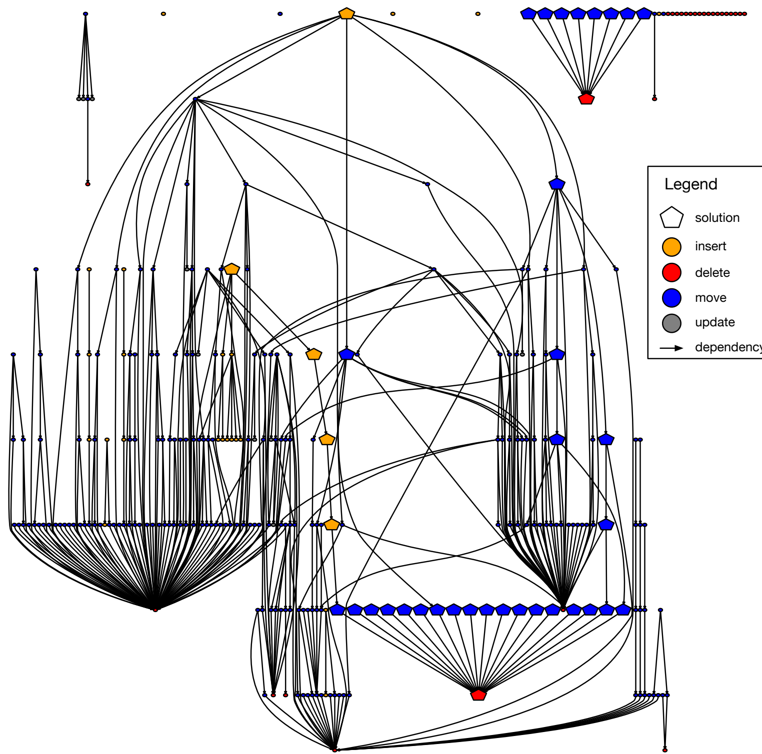

Fig. 21: CDG created for the commit with Id 9 in Table 4.

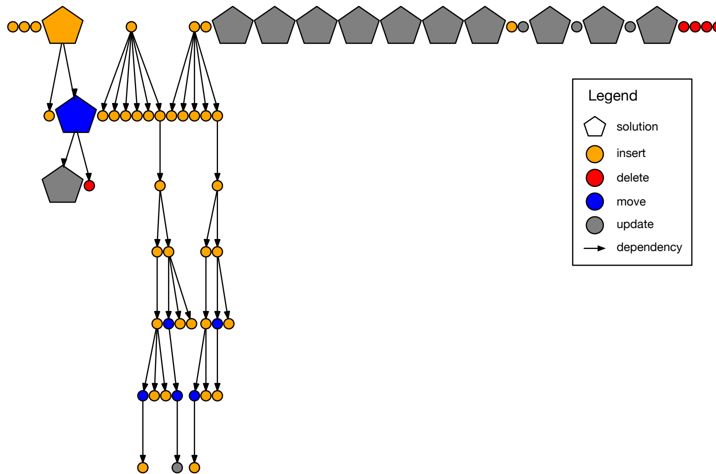

Fig. 22: Figure depicting the CDG created for the commit with Id 11 in Table 4.

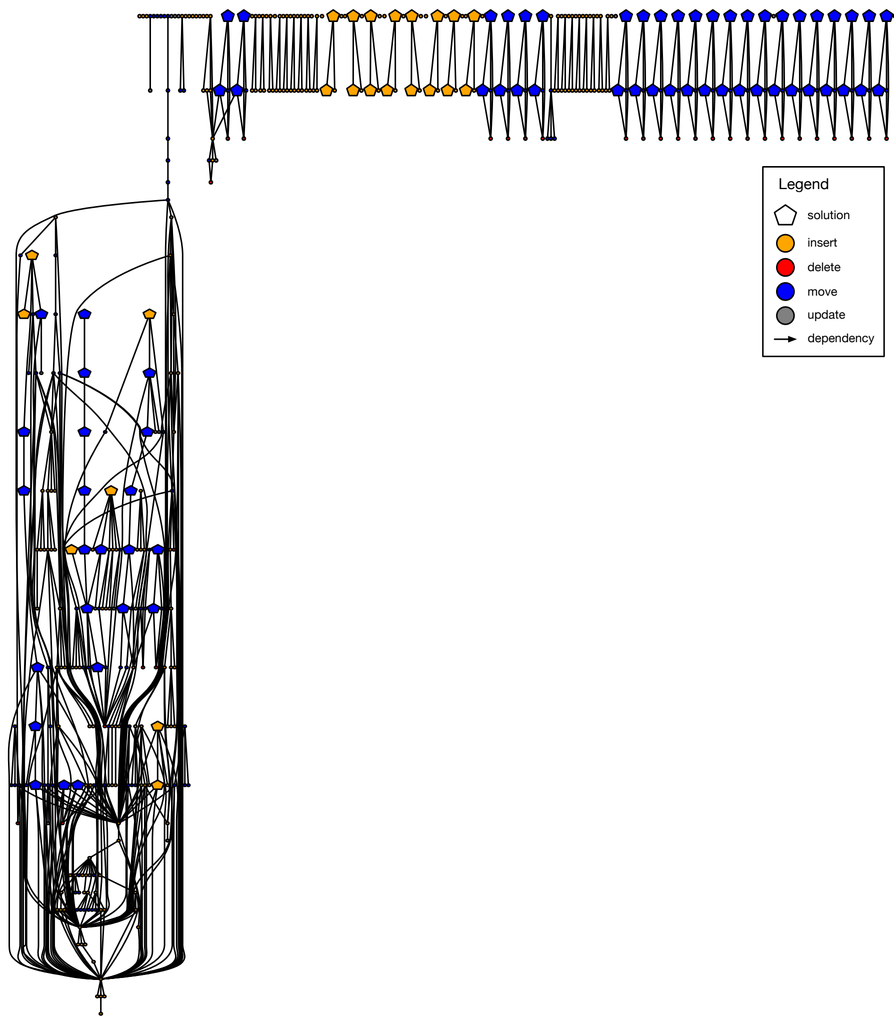

Fig. 23: CDG created for the commit with Id 15 in Table 4.

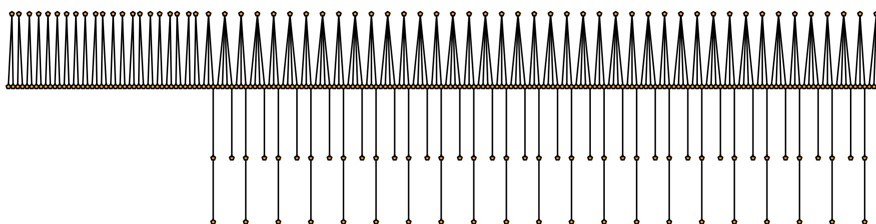

Fig. 24: CDG created for the commit with Id 16 in Table 4.

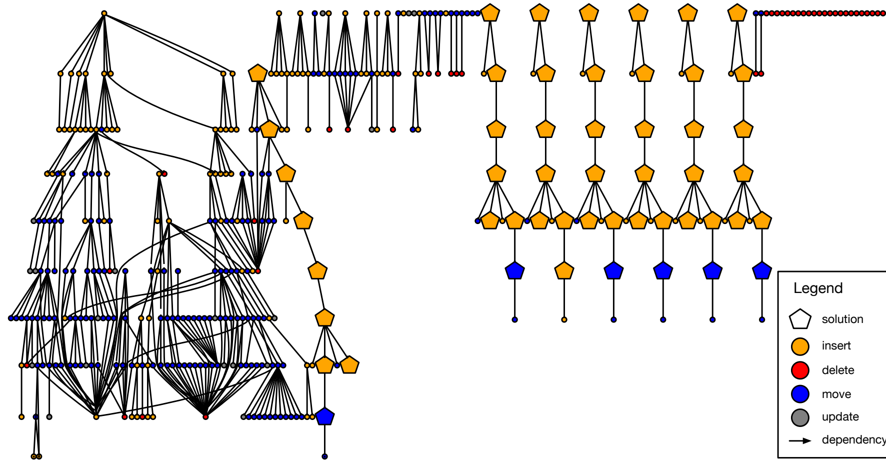

Fig. 25: CDG created for the commit with Id 14 in Table 4.

Results for "Remove Cast": The solution for the "Remove Cast" systematic edit (Id 15) consists mostly out of EI changes. Figure 23 depicts the CDG of the distilled changes. A large number of solution changes, depicted on the right hand side, remove the cast by overwriting the existing cast expression with a call to `getCodeValue`. A much more complex component is depicted on the left. Some existing code was modified by non-solution changes, hereby also affecting calls to `getCodeValue`. The extraction of these changes results in more ES and EL changes being part of the solution. The resulting untangled commit will contain some code that is not part of the systematic edit, but that is needed when the remaining changes are applied to create the final commit.

Results for "Introduce Field and Accessors": The final result is a commit (Id 16) that only contained changes that contribute to the systematic edit. Our query successfully identified that all the changes were needed to introduce the fields and corresponding accessors. Figure 24 depicts the CDG computed for these changes. The changes in the top left introduce the field, while the changes on the right introduce a getter and a setter.

6 Evaluation - User study

After having evaluated that our approach works as intended, this section focuses on evaluating the query language itself. We are interested specifically in the usability and expressiveness aspects of the language. To this end, this section addresses the following research questions:

RQ6 Is our query language sufficiently easy-to-use for researchers in the field of software engineering?

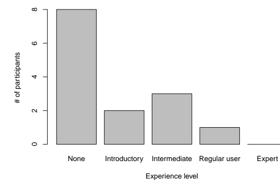

Fig. 26: Prior EKEKO experience of participants

RQ7 Is our query language sufficiently expressive to describe a wide variety of code transformations?

To answer these two research questions, we performed a user study[8] in which participants were asked to use our query language to both understand existing queries, and also write a few queries of their own.

6.1 User study demographics

Given that our approach is primarily meant to be used by researchers, our user study was performed with 14 participants from academia (12 people of the Software Languages Lab, Vrije Universiteit Brussel, and 2 people of the Ansymo lab, Universiteit Antwerpen). Our group of participants consists of 3 postdocs, 9 Phd students and 2 Master's students. All participants joined the study on a voluntary basis. Eleven participants have experience working in the broader field of software engineering. Six of those eleven participants specifically have experience in the area of mining software repositories, meaning they are directly representative of our approach's target audience.

None of the participants had prior experience using our approach, although some have experience with the EKEKO metaprogramming library, as shown in Fig. 26. This experience is relevant, as the desired properties of an entity state are described in terms of EKEKO predicates (i.e. the goals in the `in-current-es` construct).

6.2 User study design

At the beginning of the study, a 15 minute presentation is given to introduce the participants to the main constructs used in the query language. One simplification that we made when describing our query language is that we restricted all queries to follow the form shown in Fig. 27. That is, all queries first describe an evolution state with a "before" situation of interest, then several changes are applied using the `change->+` operator, followed by a description of the "after" situation. This simplification allows participants to focus their

[8] All material that was provided in the user study, including the study's complete results, are available online: `http://soft.vub.ac.be/~tmoldere/qwalkeko`

```
1   (query-changes esg ?es [;local variables]
2     (in-current-es [es ast]
3       ; goals before-ES)
4     change->+
5     (in-current-es [es ast]
6       ; goals after-ES ))
```

Fig. 27: Template used for all queries in the user study.

attention on the essential parts of a query, i.e. the descriptions of evolution states. We consider this a reasonable simplification, as most queries (including all queries mentioned in this paper) typically follow the same form. In addition to the introductory presentation, participants are also given a documentation page that provides a description and examples for the different predicates available in our query language.

The user study itself proceeds in four phases:

Phase 1 - The first phase consists of three reading tasks, intended to test whether the participants can understand a given query. In each task, a specific query is given (without any comments or description) and four possible matches for that query. Participants need to choose which of the given matches are correct, as well as deduce from the query what its purpose is.

Phase 2 - After the reading tasks, the participants are asked to enter their answers in an intermediate questionnaire, which also includes a few questions to gauge their overall experience with the language so far.

Phase 3 - In this phase of the study, the participants are given three writing tasks. These tasks are meant to test the ease-of-use of the language, as well as test whether the language is expressive enough to solve these tasks in a manner that is intuitive to the participants. In each writing task, participants are asked to write a query such that it fits the given description and is able to find the given match.

Phase 4 - The study concludes with a final questionnaire to measure the participants' experience regarding the learning curve, usability, expressiveness and overall experience with the language.

These four phases are explained in more detail in the following sections. The entire study, including the introductory presentation and filling in both questionnaires, takes between 2 to 3.5 hours.

6.3 Reading tasks

There are three different reading tasks in the user study. As mentioned earlier, a query and a number of possible matches are given in each task. If a possible match contains the transformation that is described in the query, it is a correct match. Each participant is then asked to select which of the potential matches are correct, and to provide a short description of the query's purpose.

Each given match is provided in the form of a before- and an after source code file. The changes between the before- and after code may or may not

contain the transformation described in the given query. If the changes do contain this transformation, the match is correct. Note that the before and after code may also contain changes that are irrelevant to the query, both to represent a realistic commit and to highlight that our approach is able to ignore these changes. To easily compare the before and after code, participants were also given a textual diff of all possible matches. In addition, because our query language examines code at the level of abstract syntax trees, participants also had access to a simple AST browser[9] to view the tree structure of all before and after code. Having access to the AST structure is especially important to easily determine the type names of AST nodes (needed in e.g. the /textttast relation) and the names of properties (needed in e.g. the `has` relation). Without such a browser, participants would have to explore the extensive reference API documentation[10] for Eclipse's Java AST structure, which would both consume a lot of time, and distract participants from working with our query language.

The queries that were given in the reading tasks do not contain any comments, and the logic variables used in the query only provide an indication regarding what type of AST node is contained in each variable, not the query's purpose. An example of one of the queries we used, and one of its correct matches, are given in Fig. 28 and Fig. 29. This particular query looks for the removal of null checks. More specifically, it looks for an `InfixExpression` that performs a null equality check on `?left` in the "before" evolution state, then it checks that `?left` still appears in the "after" evolution state (assuming it is needed elsewhere in the code), but no longer as part of a null check.

```
1 (query-changes esg ?es
2   [?method ?expression ?right ?left ?after-method ?left-after]
3   (in-current-es [es ast]
4     (child+ ast ?method)
5     (child+ ?method ?expression)
6     (ast :InfixExpression ?expression)
7     (conde
8       (has :operator ?expression "==")
9       (has :operator ?expression "!="))
10    (has :leftOperand ?expression ?left)
11    (has :rightOperand ?expression ?right)
12    (ast :NullLiteral ?right))
13  change->+
14  (in-current-es [es ast]
15    (ast-method-method|corresponding ast ?method ?after-method)
16    (child+ ?after-method ?left-after)
17    (ast-equals ?left ?left-after)
18    (fails
19      (fresh [?expression-after ?right-after]
20        (parent ?left-after ?expression-after)
21        (has :rightOperand ?expression-after ?right-after)
22        (ast :NullLiteral ?right-after)))))
```

Fig. 28: Query that looks for removed null checks.

[9] Based on the `org.eclipse.jdt.astview` plugin

[10] This refers to the API documentation for all classes in the `org.eclipse.jdt.core.dom` package.

```
// Before
public class TreeNode {
  public void addChild(TreeNode n) throws Exception {
    if (n!= null) {
      children.add(n);
    } else {
      throw new Exception("Cannot add null children");}}
  ...
}

// After
public class TreeNode {
  public void addChild(TreeNode n) throws Exception {
    children.add(n);
  }
  ...
}
```

Fig. 29: Example of a correct match for Fig. 28.

The three reading tasks consist of the following three queries:

T1.1 Removal/rename of a formal parameter in an interface method
T1.2 Removal of a null check (shown in Fig. 28)
T1.3 Magic constant refactoring (also used in the previous evaluation, Fig. 8)

Based on the query description that the participants provided for each task, we examined how many people found a correct answer, a partially correct answer or an incorrect answer. A partially correct answer would be, for instance in T1.1, only mentioning that a parameter is renamed in the query (even though it can also be a removal). An answer that is either too abstract or unrelated to the correct answer is deemed incorrect. For T1.1, there are 7 correct answers and 7 partial answers. For T1.2, there are 10 correct answers, 2 partial answers and 2 incorrect answers. For T1.3, there are 10 correct answers, 1 partial answer and 3 incorrect ones.

However, aside from the query description, if we consider which matches were selected in each task, there are few fully correct answers. In T1.1, 4 participants selected *only* the correct matches. For T1.2, there are 2 participants. For T1.3, there are 4 participants. On the other hand, almost everyone did select some (or too many) of the correct matches in all tasks.

These results seem to indicate that our participants do have a general understanding of the query language's concepts, but the exact semantics of some predicates may be more complex. In particular, several participants mentioned in the intermediate questionnaire they found the `fails` predicate to be confusing, which implements the "negation as failure" concept of logic programming. It may be possible to prevent such confusion by providing additional/improved documentation and examples. It also is worth noting that, during these tasks, participants were not allowed to execute the queries. This complicates understanding the queries' semantics, but this was necessary as executing the queries would make it trivial to select which of the given matches are correct. These reading tasks have primarily measured the understandability of our query lan-

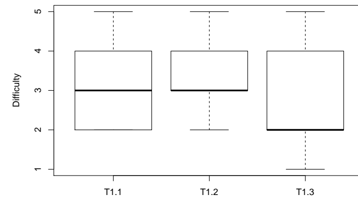

Fig. 30: Difficulty rating of the reading tasks

guage, which is an important factor of the language's ease-of-use, and partially addresses RQ6.

Participants were also asked to rate the difficulty of each task using a Likert scale (Oppenheim 2000), as shown in Fig. 30. The difficulty ratings are rather varied, but none of the reading exercises are considered trivial. This corresponds to our expectations, as we chose our tasks to represent realistic uses of our language, rather than design more artificial tasks that focus on testing one specific predicate of the language at a time. While the latter approach would be more useful to pinpoint which specific predicates are more difficult or easier to understand, this would require a large amount of tasks and would be less representative for testing the understandability of real-world queries.

6.4 Writing tasks

After the reading tasks and the intermediate questionnaire, the participants are asked to perform three writing tasks. In each task, a query should be written (from scratch) such that it produces the given match and corresponds to the given description. The given match of each task is provided in the same form as the reading tasks: a before- and after source code file, with access to a textual diff and an AST browser.

The given descriptions for the three writing tasks are the following:

T2.1 Write a query that looks for methods that were private, but they are now changed to public.
T2.2 Write a query that looks for the removal of an unused method. That is, private methods that are never called. (This is a query also used in a previous experiment, shown in Fig. 9.)
T2.3 Write a query that looks for migrations of Junit 3 tests to Junit 4 tests. Main changes between Junit 3 and 4:
  – Test classes no longer need to extend `TestCase`
  – The `setUp` method can now have any name, but it should get a `@Before` annotation. Likewise, the `tearDown` method can now have any name, but it should get an `@After` annotation

– Test methods no longer need the have a ”test” prefix in their name, but should get a `@Test` annotation.

For each task, participants had to keep track of time and were given a 30 minute time limit. The time taken for T2.1 and T2.2 is given in the box plots[11] of Fig. 31. The time for T2.3 was not included because, as can be inferred from the task's description, this task is significantly more complex than the other two, and was not intended to be finished in 30 minutes. The idea for this task was to gauge how much of the query could be completed within 30 minutes.

In case of T2.1, 10 (out of 14) participants were able to complete the task successfully. For T2.2, 9 participants finished the task. After the study we found that most of the remaining participants got stuck due to a minor bug in our AST browser that does not show the (only) child of a `Modifier` AST node. Visiting this child is one way to determine whether this AST node represents a `public` or `private` keyword, which is part of T2.1 and T2.2. There are other ways (as used by the other participants) in which visiting the children is not necessary, e.g. by examining the source code representation of an AST node. If anything, this issue shows that participants do rely on the AST browser as a companion tool to help with writing queries.

While T2.3 was not intended to be completed within the allotted 30 minutes time, 2 participants did complete this task. Five people indicated they were close to completion. In the questionnaire, several participants indicated a need for some additional higher-level predicates for more commonly used operations, such as retrieving a class's superclass. It should be straightforward to implement such predicates on top of the core Ekeko predicates.

We also manually examined the queries that the participants wrote to examine the degree of variation between the different solutions. For the most part, the main form of variation consists of some participants (typically those with prior Ekeko experience) opting to make more use of higher-level/compound predicates compared to sticking a more limited collection of lower-level/core predicates.

Finally, the difficulty rating given per writing task is given in Fig. 32. As expected, T2.3 is considered the most difficult one. While the difficulty ratings are not significantly different from those in the reading tasks, the query language was sufficiently usable and expressive for most participants to solve T2.1, T2.2 and (partially) T2.3.

6.5 User study results

After finishing the writing tasks, the user study is concluded with a final questionnaire, which asks the participants to directly rate the language's usability, expressiveness, its learning curve, its usefulness for the MSR community, and

[11] This paper uses Tukey box plots (Frigge et al 1989), the default type of box plots produced by the R language.

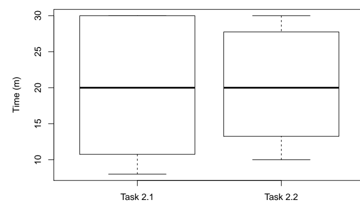

Fig. 31: Time taken for the writing tasks

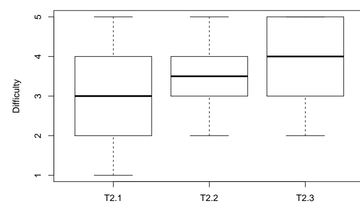

Fig. 32: Difficulty rating of the writing tasks

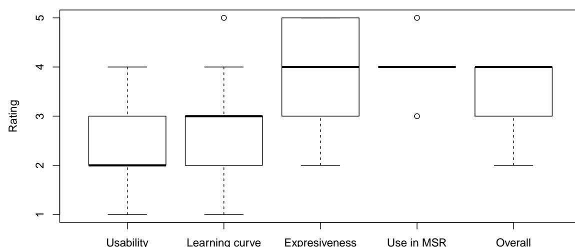

Fig. 33: Final questionnaire ratings

to provide an overall rating. These results are given in Fig. 33. The y-axis for each plot can be interpreted as "higher is better".

In case of usability, the rating ranges from 1 (difficult-to-use) to 5 (easy-to-use). It is clear from the box plot that our language is generally considered more difficult to use (with an average usability rating of 2.5). This result may be partially attributed to the previously mentioned issue that some participants had during the writing tasks. However, it can also be attributed to a significant extent that some of the difficulty is inherent, as our approach requires familiarity with both logic programming and the AST structure of

Java source code. This is also reflected in the rating for the learning curve (2.9 on average) , which ranges from 1 (steep) to 5 (smooth). In this category, most people opted for 3. To rate the learning curve, each of the options had specific labels; the label of option 3 is "I understand the basics, and I am confident I will become more proficient with some more practice.". This label also corresponds to our expectations of the learning curve, given the 2-3 hours of experience that our participants received in this study. Nonetheless, as a few participants have mentioned, the usability of the approach can also be improved with additional tool support, such as an autocompletion feature while writing queries. Overall, considering the results of all tasks, the usability rating and the learning curve rating, we can provide an answer to **RQ6**: while there is a learning curve to become fluent with our approach, and there is room for improvement in terms of tool support and predicate documentation, the approach is sufficiently usable to write useful queries.

In terms of expressiveness, our approach generally has received high ratings in Fig. 33 (4 on average). Expressiveness was also mentioned by several participants when asked what they liked about the approach. Considering that the tasks were also chosen to represent a wide variety of practical uses of our approach, **RQ7** can be answered: our query language is sufficiently expressive to express a wide variety of code transformations.

While the last two ratings in Fig. 33 are more informal, we considered it worth mentioning that the approach is perceived as useful to the MSR research community by all participants, and that the approach received an average overall rating of 3.5.

7 Limitations and threats to validity

This section focuses on limitations of the approach itself, and threats to validity regarding the experiments that were performed.

7.1 Limitations

Performance - Runtime performance and memory usage are currently not a focus of the approach. However, as the experiments have shown, history queries can take a long time to complete. This may be acceptable for empirical studies, but less so for developers who want to obtain near-instant answers to their queries. The main parts of the approach that affect performance are the following: first, importing the source of a revision from a version control system can take several seconds due to data being written to disk, and Eclipse updating its internal models. Second, distilling changes between two versions has a time complexity of $O(n^2)$, where n is the maximum number of nodes in either the source or target AST (Falleri et al 2014). Finally, the main factor affecting performance is the structure of the CDG, as discussed in Section 5. Some of these performance issues are mitigated: the execution of evolution

queries is implemented such that solutions can be computed in an on-demand manner. This makes it possible to write queries incrementally, which can be tested quickly. As discussed in Section 4.1, there also are coarse-grained language constructs, `change==>` and `change==>*`, which can be used to reduce the search space of a query.

If the user needs to apply the approach to a range of revisions, further optimizations are possible: while not discussed in this paper, we have also implemented predicates to access version control information. (e.g. which files are modified in a commit, who made the modifications, ..) This can be obtained relatively quickly, compared to obtaining fine-grained change information. As such, it may be possible to write an initial query in terms of version control information to find any relevant revisions, and to write a subsequent query that uses the fine-grained representation for only those relevant revisions. Finally, to reduce memory usage, an incremental representation of the source code could be used (Alexandru and Gall 2015). In such a representation, an initial AST is created for the first revision containing that AST. Later revisions are represented by only storing the modifications to that initial AST.

Multi-file support - Our approach currently receives two revisions of one file as input. Consequently, transformations involving multiple files cannot be easily expressed. For instance, an evolution query that can recognize modifications to an API, in which a method declaration and all calls to that method are modified. A naive approach to supporting multiple files is to simply merge the contents of all files into one large file. However, this creates an unfeasibly large CDG with many connected components, which was discussed in RQ3 of Section 5. To address this problem, partial order reduction techniques (Peled 1998), typically used to reduce the state space of concurrent systems, could be repurposed to reduce the search space of a CDG.

7.2 Threats to validity

False positives - During the experiments, we only ran the evolution queries on revision pairs that are known to contain the sought-after patterns. This was done to reduce the total running time of the experiment. As such, we cannot claim that these patterns were *not* found in any of the other revision pairs. However, if additional matches would be found, this either means that the oracle needs to be adjusted, or it means that the evolution query needs to be modified. In either way, the evolution query itself cannot be blamed. It can only produce solutions that match its specifications, which is why we mentioned in Section 5 that a notion of false positives does not apply to our approach.

Generalizability - The experiments focused on two types of transformations, refactoring and systematic edits. There may be other important types of transformations that do not fit in one of these two categories. More importantly, as the approach is currently restricted to examining the evolution of

one file at a time, our approach currently does not scale yet to more complex transformations that span multiple files.

Intermediate evolution states - We have currently used our approach to detect source code transformations where it is sufficient to identify only the source and the target AST. For instance, the refactoring experiment looks for an ES where the sought-after refactoring is not present yet, and an ES where the refactoring has been applied. We have not thoroughly tested evolution queries that also specify properties for any intermediate evolution states.

User study size - The user study of Sec.6 was performed with 14 participants, of which 6 have experience in the MSR field. This may not be statistically representative of the MSR community.

8 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 (Hajiyev et al 2006), PQL (Martin et al 2005) and SOUL (De Roover et al 2011). 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 (Hindle and German 2005) and V-Praxis (Mougenot et al 2009), extended a PQL by adding a revision argument to each predicate. More recent tools feature dedicated specification languages. The Boa platform (Dyer et al 2013) 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 (Fluri et al 2007) 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 (1996). gumtree (Falleri et al 2014) 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.

Changes and their dependencies have been used in various different contexts: The work of Martinez et al (2013) is most closely related to ours; it makes use of ChangeDistiller to look for specific code change patterns related to bug fixing. These patterns are specified directly in terms of fine-grained

changes and their relations, whereas our evolution queries describe patterns at a higher level, in terms of properties of the source code. Ebraert et al (2007) have detected dependencies between recorded changes, while Uquillas Gómez et al (2014) help integrators navigate changes and their dependencies. To this end, they both build a FAMIX model and a corresponding change model that models changes made to the FAMIX model. The level of changes they work with are modifications to the FAMIX model, and are more coarse-grained than AST nodes. They do provide some semantic dependencies while we limit ourselves to syntactic dependencies.

The work of Yoon and Myers (2015) presents the AZURITE tool, which extends the Eclipse IDE to selectively undo fine-grained code changes. To undo a selection of previous code changes, the tool needs to detect any conflicts that can occur between changes, which closely corresponds to the notion of dependencies between changes. AZURITE operates using textual changes, which are logged while the developer is typing code. Our approach makes use of more fine-grained AST-level changes, which could be useful to detect conflicts at a semantic level next to structural conflicts. Hayashi et al (2012) also use textual changes, with the aim of restructuring changes to improve understandability, which is closely related to the problem of commit untangling. Using AST-level changes can be helpful to restructure according to semantic criteria.

The work of Servant and Jones (2012) presents a technique to obtain the history of a set of lines of code, to determine when/who inserted/removed each line in a project. The changes in this work are represented at a line-based level. Our AST-based approach could supplement their work, in the sense that evolution queries can be written to track the evolution of methods, statements, variables, etc.

Finally, OperationSliceReplayer (Maruyama et al 2016) is an approach that enables skipping uninteresting changes from a sequence of *logged* changes, in order to construct a particular class member of a Java class. To this end, they construct an "operation history graph", which models changes and the different class members present in the source code they affect. Their main goal is to help developers understand how code has evolved by only replaying changes that are of interest to the developer. The main difference with our approach is that we focus on replaying distilled changes in different orderings to find a minimal set that implements an evolution pattern, while they focus on skipping changes in a change sequence in order to efficiently build a particular class member. As such, their graph is based around storing dependencies between class members and their affecting changes, while our graph focuses on dependencies between changes.

On a related note, Li et al. present an approach to slice software history such that the changes that implement a particular feature/bug fix are grouped together, with the aim of simplifying porting/transplanting certain functionality to different branches. To allow establishing semantically related changes, this work represents changes at the AST-level.

Similar to one of our experiments, Weissgerber and Diehl (2006) present a technique to identify refactorings from source code changes. Their model

of changes is specifically targeted at this purpose, considering adding/removing/moving classes, parameters , fields, etc. The model used in our work is general-purpose, as it could be used to describe changes in any (abstract syntax) tree structure. Consequently, our approach requires some additional work in describing the desired abstractions needed for an evolution query that detects a particular refactoring. On the other hand, our approach can be applied in a variety of different applications and contexts.

In summary, existing approaches either do not include low-level source code changes, or do not feature a dedicated query language. In this paper, we have presented such a query language and have integrated it in the history querying tool QwalKeko by Stevens and De Roover (2014).

9 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. Another potential path of future work is to lower the learning curve of our approach such that evolution queries can be specified without any knowledge about Java AST structures. One option is to provide integration with the Ekeko/X tool by De Roover and Inoue (2014), a tool built on top of Ekeko where source code can be queried and transformed in terms templates written in Java.

We want to investigate further applications of our approach. For instance, we want to investigate whether we can cherry-pick changes from a commit, e.g. to extract a single feature. This feature can be expressed as an evolution query, and our approach returns a minimal executable edit script. We want to see whether such an 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.

10 Conclusion

We have presented an approach to extracting a minimal executable edit script from distilled change sequences. The change equivalence and change representation problems render specifying the sought-after transformation in terms of the properties of the changes difficult. Instead, an evolution query describes the source code prior and after the sought-after code transformation. Thus, the transformation is specified in terms of the source code, solving the change equivalence and change representation problems. 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.

Acknowledgements

We would like to thank all participants of our user study. We would also like to thank the anonymous reviewers for their detailed reading of this manuscript and their high-quality feedback.

References

Alexandru CV, Gall HC (2015) Rapid multi-purpose, multi-commit code analysis. In: Proc. of the 37th Int. Conf. on Software Engineering (ICSE15)

Chawathe SS, Rajaraman A, Garcia-Molina H, Widom J (1996) Change detection in hierarchically structured information. In: Proc. of the Int. Conf. on Management of Data (SIGMOD96)

Christophe L, Stevens R, De Roover C (2014) Prevalence and maintenance of automated functional tests for web applications. In: Proc. of the Int. Conf. on Software Maintenance and Evolution (ICSME14)

De Roover C, Inoue K (2014) The ekeko/x program transformation tool. In: Proc. of 14th Int. Working Conf. on Source Code Analysis and Manipulation (SCAM14), Tool Demo Track

De Roover C, Stevens R (2014) Building development tools interactively using the ekeko meta-programming library. In: Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR14)

De Roover C, Noguera C, Kellens A, Jonckers V (2011) 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)

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: Proc. of the Int. Conf. on Software Engineering (ICSE13)

Ebraert P, Vallejos J, Costanza P, Paesschen EV, D'Hondt T (2007) Change-oriented software engineering. In: Proc. of the 2007 Int. Conf. on Dynamic languages (ICDL07)

Falleri JR, Morandat F, Blanc X, Martinez M, Montperrus M (2014) Fine-grained and accurate source code differencing. In: Proc. of the 29th Int. Conf. on Automated Software Engineering (ASE14

Fluri B, Würsch M, Pinzger M, Gall HC (2007) Change distilling: Tree differencing for fine-grained source code change extraction. Transactions on Software Engineering 33(11)

Frigge M, Hoaglin DC, Iglewicz B (1989) Some implementations of the boxplot. The American Statistician 43(1):50–54

Hajiyev E, Verbaere M, Moor OD (2006) Codequest: Scalable source code queries with datalog. In: Proc. of the 20th European conference on Object-Oriented Programming (ECOOP06)

Hayashi S, Omori T, Zenmyo T, Maruyama K, Saeki M (2012) Refactoring edit history of source code. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp 617–620, DOI 10.1109/ICSM.2012.6405336

Hindle A, German DM (2005) SCQL: A formal model and a query language for source control repositories. In: Proc. of the 2005 Working Conf. on Mining Software Repositories (MSR05)

Kamiya T, Kusumoto S, Inoue K (2002) Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering

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

Liu YA, Rothamel T, Yu F, Stoller SD, Hu N (2004) Parametric regular path queries. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI04)

Martin M, Livshits B, Lam MS (2005) 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)

Martinez M, Duchien L, Monperrus M (2013) Automatically extracting instances of code change patterns with ast analysis. In: 2013 IEEE International Conference on Software Maintenance, pp 388–391, DOI 10.1109/ICSM.2013.54

Maruyama K, Omori T, Hayashi S (2016) Slicing fine-grained code change history. IEICE Transactions 99-D(3):671–687

Meng N, Kim M, McKinley KS (2013) Lase: Locating and applying systematic edits by learning from examples. In: Proc. of the 35th Int. Conf. on Software Engineering (ICSE13)

Molderez T, Stevens R, De Roover C (2017) Mining change histories for unknown systematic edits. In: Proc. of the 14th Int. Conf. on Mining Software Repositories (MSR17)

de Moor O, Lacey D, Wyk EV (2002) Universal regular path queries. Higher-Order and Symbolic Computation pp 15–35

Mougenot A, Blanc X, Gervais MP (2009) D-Praxis: A peer-to-peer collaborative model editing framework. In: Proc. of the 9th Int. Conf. on Distributed Applications and Interoperable Systems (DAIS09)

Murphy-Hill E, Parnin C, Black AP (2012) How we refactor, and how we know it. Transactions on Software Engineering 38:5–18

Negara S, Codoban M, Dig D, Johnson RE (2014) Mining fine-grained code changes to detect unknown change patterns. In: Proc. of the 36th Int. Conf. on Software Engineering (ICSE14)

Oppenheim AN (2000) Questionnaire design, interviewing and attitude measurement. Bloomsbury Publishing

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

Peled D (1998) Ten years of partial order reduction. In: Hu AJ, Vardi MY (eds) Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 17–28

Pérez J (2013) Refactoring planning for design smell correction: Summary, opportunities and lessons learned. In: 2013 IEEE International Conference on Software Maintenance, pp 572–577, DOI 10.1109/ICSM.2013.98

Prete K, Rachatasumrit N, Sudan N, Kim M (2010) Template-based reconstruction of complex refactorings. In: Proc. of the 2010 Int. Conf. on Software Maintenance (ICSM10)

Servant F, Jones JA (2012) History slicing: Assisting code-evolution tasks. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, FSE '12, pp 43:1–43:11, DOI 10.1145/2393596.2393646, URL http://doi.acm.org/10.1145/2393596.2393646

Stevens R (2015) A declarative foundation for comprehensive history querying. In: Proc. of the 37th Int. Conf. on Software Engineering, Doctoral Symposium Track (ICSE15)

Stevens R, De Roover C (2014) Querying the history of software projects using QWALKEKO. In: Proc. of the 30th Int. Conf. on Software Maintenance and Evolution

Stevens R, De Roover C (2017) Extracting executable transformations from distilled code changes. In: Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER17)

Uquillas Gómez V, Ducasse S, Kellens A (2014) Supporting streams of changes during branch integration. Science of Computer Programming 96

Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V (1999) Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, CASCON '99, pp 13–, URL http://dl.acm.org/citation.cfm?id=781995.782008

Weissgerber P, Diehl S (2006) Identifying refactorings from source-code changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp 231–240, DOI 10.1109/ASE.2006.41

Yoon YS, Myers BA (2015) Supporting selective undo in a code editor. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 223–233, URL http://dl.acm.org/citation.cfm?id=2818754.2818784

Reinout Stevens is currently employed as a full stack developer and data scientist at Maxflow BVBA. In 2017, he obtained his PhD in Computer Science at the Software Languages Lab of the Vrije Universiteit Brussel. His research focused on the use of declarative programming to reason about the history of software projects.

Tim Molderez is a postdoctoral researcher at the Software Languages Lab in Brussels. He is currently active in the area of machine learning techniques applied to distributed systems. In 2014 he obtained his PhD on modular reasoning in aspect-oriented languages at the Universiteit Antwerpen. His research interests include static program analysis, program transformation, distributed systems and data mining.

Coen De Roover is an assistant professor at the Software Languages Lab of the Vrije Universiteit Brussel in Belgium. The central theme of his research is the design of program analysis and transformation techniques, and their application in software engineering tools for quality assurance. Example analysis techniques include abstract interpretation of dynamically-typed programs in general, and of JavaScript programs in particular. Example tools include tools for detecting user-specified bug patterns in an implementation, or for validating an implementation with respect to a user-specified design. Here, an executable logic often serves as the tools specification language.