Recording and Replaying System Specific, Source Code Transformations

Gustavo Santos, Anne Etien, Nicolas Anquetil, Stéphane Ducasse, Marco Tulio Valente

▶ **To cite this version:**

Gustavo Santos, Anne Etien, Nicolas Anquetil, Stéphane Ducasse, Marco Tulio Valente. Recording and Replaying System Specific, Source Code Transformations. 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Sep 2015, Bremen, Germany. pp.10. hal-01185639

HAL Id: hal-01185639
https://inria.hal.science/hal-01185639

Submitted on 20 Aug 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Recording and Replaying System Specific, Source Code Transformations

Accepted to 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015)

Gustavo Santos, Anne Etien, Nicolas Anquetil and Stéphane Ducasse
RMoD Team, INRIA Lille Nord Europe
University of Lille, CRIStAL, UMR 9189
{firstname.lastname}@inria.fr

Marco Tulio Valente
Department of Computer Science
Federal University of Minas Gerais
mtov@dcc.ufmg.br

***Abstract*—During its lifetime, a software system is under continuous maintenance to remain useful. Maintenance can be achieved in activities such as adding new features, fixing bugs, improving the system's structure, or adapting to new APIs. In such cases, developers sometimes perform sequences of code changes in a systematic way. These sequences consist of small code changes (e.g., create a class, then extract a method to this class), which are applied to groups of related code entities (e.g., some of the methods of a class). This paper presents the design and proof-of-concept implementation of a tool called MACRORECORDER. This tool records a sequence of code changes, then it allows the developer to generalize this sequence in order to apply it in other code locations. In this paper, we discuss MACRORECORDER's approach that is independent of both development and transformation tools. The evaluation is based on previous work on repetitive code changes related to rearchitecting. MACRORECORDER was able to replay 92% of the examples, which consisted in up to seven code entities modified up to 66 times. The generation of a customizable, large-scale transformation operator has the potential to efficiently assist code maintenance.**

***Index Terms*—Software Maintenance; Software Evolution; Refactoring; Programming By Demonstration; Automated Code Transformation.**

I. INTRODUCTION

Software systems must constantly evolve to remain useful in their context. Automation of code transformations is currently provided by refactoring tools, such as the Eclipse IDE. However, recent work provided discussion about the lack of trust in refactoring tools [10, 11, 16]. Developers do not understand what most of the operators actually do. Even when the refactoring tool provides an automatic operator, developers sometimes prefer to perform the same code changes manually.

On the other hand, previous work showed that developers sometimes perform sequences of code changes in a systematic way [8, 12, 14, 15]. These sequences are composed of small code transformations (e.g., create a class, implement a given interface, then override a method) which are applied to diverse but similar code entities (e.g., some classes in the same hierarchy). Such systematic behavior have been studied in the literature in the context of fixing bugs [12], and adapting a system to accomodate API updates [14].

Similarly, our recent work studied repetitive code transformations when developers improve the structure of the system [15]. When applied manually, the sequences of code changes were complex, due to the repetition of similar but not identical transformations. Developers sometimes missed opportunities to apply transformations, or did not perform all the transformations defined in the sequence.

Therefore, there is a need to specify customizable sequences of source code transformations. Such support differs from refactorings because: (i) refactorings are behavior-preserving; (ii) they are limited to a predefined set of code transformations; and (iii) refactorings are generic because they can be applied to systems from different domains. In our previous case study [15], we assumed that the behavior of the system can change. Moreover, the sequences of transformations we found are specific to the systems in which they were applied.

In this paper, we discuss the proof-of-concept implementation of MACRORECORDER. This prototype tool allows the developer to: (i) record a sequence of source code changes; (ii) store and generalize this sequence of changes, in order to afterwards (iii) apply it automatically to different code locations.

This tool presents two main contributions:

- our approach is independent of both development and transformation tools. MACRORECORDER records code edition events from the development tool that can be reproduced automatically;
- the tool allows the developer to generalize transformations in order to easily instantiate them in other code locations. This generalization is currently manual.

We evaluated the approach using real examples of code repetition in software systems. These examples (i) consist in up to nine transformations, (ii) impact up to seven code entities of different levels of abstraction (e.g., packages, classes, and methods), and (iii) they were repeated up to 66 times. MACRORECORDER applies 92% of the examples with 76% accuracy. This paper also provides discussion on the effort to generalize and automate these examples.

The paper is organized as follows: Section II provides a motivating example and the considered problem. Sections III

and IV present our solution and Section V presents the study results. Section VI presents threats to validity. Finally, Section VII presents related work on automating code transformation and Section VIII concludes.

II. Motivating Example

In recent work, we found evidences that some sequences of code changes are performed in a systematic way [15]. These sequences are composed of small code transformations which are repeatedly applied to groups of related entities. In this section, we present a motivating example of systematic code transformations.

This example is extracted from PetitSQL, a small parser in Smalltalk for static analysis of SQL commands [15]. Consider two classes, `ASTGrammar` and `ASTNodesParser`, the second inheriting from the first. Listings 1 and 2 present examples of source code modification in this system. For comprehension purposes, we illustrate the example in a Java-inspired syntax.

Listing 1: Modified code in `ASTGrammar`

```
public functionCall() {
  return { identifier,
    new Character('('),
-   argument.separatedBy(','),
+   arguments(),
    new Character(')') };
}

+ public arguments() {
+   return argument.separatedBy(',');
+ }
```

Listing 2: Modified code in `ASTNodesParser`

```
public functionCall() {
  filtered = new Collection();

  for element in super.functionCall() {
    filtered.add( new SQLFunctionCallNode()
      .setName( element.first() )
      .setArguments( element.third()
-       .filter(',') ) );
+       ) );
  }
  return filtered;
}

+ public arguments() {
+  return super.arguments()
+    .withoutSeparators();
+ }
```

The methods in `ASTGrammar` represent different elements in the SQL syntax. These methods define the corresponding grammar rule as a collection of elements. For example, a `functionCall` returns the identifier of the function and its arguments surrounded by parentheses. The subclass `ASTNodesParser` is responsible for creating an object representing the grammar node. For this purpose, the subclass filters particular nodes that the superclass returns, using the Collections API (e.g., a call to `filter`).

Now consider that the developer can use an already existing method to filter this collection, called `withoutSeparators`. The resulting code is easier to understand, and it calls a method that was created for this intent. The resulting change in both classes is described in Listings 1 and 2, in terms of added (+) and removed (-) lines.

This change impacts two classes and creates two methods, one in each class. However, in this small maintenance, the same sequence of code changes have to be repeated in similar but not identical methods. In PetitSQL case, this sequence of changes repeated in five other methods, and results in a tedious and error-prone task.

In order to automate similar sequences of changes, we intend to be able to express each change in terms of transformations. For example in `ASTNodesParser`, one can describe: *create a method* named "arguments" in `ASTNodesParser`, then *add a return statement* in this method (which calls the superclass and `withoutSeparators`), then *remove the method call* to `filter`. Accordingly, the transformations must be abstract in the case that the names of methods and classes might be interchanged.

Our approach proposes a mechanism to build composite, trustworthy transformations that can be performed automatically. The developer is aware of what each transformation does, because he/she performs the changes manually the first time. The resulting automation reduces the chance of error in performing manual code changes, and improves the productivity in performing such repetitive task.

III. MacroRecorder in action

In this section, we introduce MacroRecorder as the tool to specify and automate composite transformations. Using our approach in practice, the developer manually performs the changes once. The tool will collect and store these changes (see Section III-A).

The developer then specifies a different code location in which MacroRecorder must replay the recorded changes. Ideally, the tool would generalize the recorded changes into a customized transformation that would be instantiated in the specified location. However, the current prototype of MacroRecorder does not support this feature. In this paper, our goal consisted in checking whether we were able to record and replay code transformations. Therefore, the generalization stage is currently manual, i.e., the developer must specify the properties of the code entities to which the transformations must be applied (Section III-B).

Finally, the tool searches for code entities that *match* the properties specified in the previous stage. If successful, the tool instantiates the transformation into these code entities and perform the transformation automatically (Section III-C). We use the PetitSQL example above to illustrate how MacroRecorder works.

A. Recording code changes

Before performing the changes, the developer must activate MacroRecorder to start to collect code changes. Concretely, a *code change* can be (i) atomic, captured by manual code edition in the development tool (e.g., Add Method, Add Association Statement) or (ii) aggregated, captured from a

refactoring tool (e.g., Extract Method, Pull Up Method). The recording process is transparent to the developer. When recording, the developer can resume editing code until the developer disables the recording explicitly.

Figure 1 (left panel) depicts the sequence of code changes that were recorded in our example. As discussed in Section II, MACRORECORDER records three code changes in `ASTNodesParser`[1]: Add Method, Add Return Statement (which calls the superclass and two methods in cascade), and Remove Call. In `ASTGrammar` there is only one code change: Extract Method. After saving this sequence of changes, MACRORECORDER adds it to the development tool as a composite transformation.

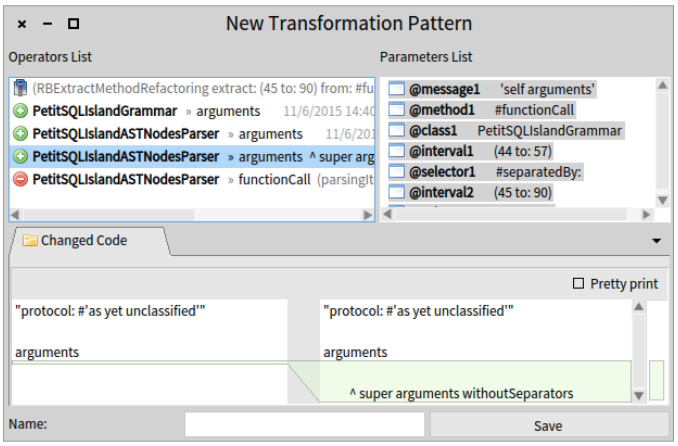

Fig. 1. Sequence of code changes recorded by MACRORECORDER in the PETITSQL example (left panel). The first code change indicates that an Extract Method refactoring was performed automatically.

Additionally, MACRORECORDER also stores the code entities that were changed by each code change (Figure 1, right panel). For example, when performing the Extract Method refactoring, the developer changed: the excerpt of code that was extracted, the new method `arguments`, the original method `functionCall`, and its containing class `ASTGrammar`.[2] We call these entities, the *parameters* of the change. They are important to generalize the sequence of changes for other examples.

B. Generalizing transformations

In this stage, the sequence of recorded code changes needs to be generalized in order to be applied in other code locations. Each code location defines a new *context*, i.e., code entities that are (i) different from the ones that were changed in the recording stage, but they also (ii) share similar properties with the recorded entities. For example, another application of these changes modifies a method named `alterTableConstraint` in `ASTGrammar`, which creates a new method called `columnReferences`. The challenge consists in expressing the context that MACRORECORDER will instantiate in each new application.

[1]Shortened from `PetitSQLIslandASTNodesParser`.
[2]Shortened from `PetitSQLIslandGrammar`.

The first generalization process is automatic and dedicated to make recorded code changes reproducible. For each event, MACRORECORDER generates a *transformation operator*. An operator is responsible for producing the code change automatically, given a list of parameters. Therefore, a code change can be the result of manual code edition, or the automated execution of a parameterized transformation operator.

Consequently, MACRORECORDER generates a *transformation pattern*, which basically consists in an ordered collection of transformation operators. Transformation patterns are system specific and they also consider aggregated operators. Therefore, transformation patterns differ from other definitions of composite transformations in the literature [4, 15]. Concretely in MACRORECORDER, a transformation pattern is responsible for instantiating a new context and executing its containing transformation operators in sequence.

The second generalization process consists in adapting the parameters to new contexts. For this prototype of MACRORECORDER, our goal is to check whether transformations can be customized and automated. Therefore, in this paper, the generalization of parameters is currently manual. However, we intend to analyze and infer the dependencies between parameters in order to automatize their instantiation (see Future Work in Section VIII).

MACRORECORDER stores the parameters for each recorded code change with unique identifiers (as introduced in Section III-A). The tool allows the developer to modify the parameter values as *expressions*. An expression is a fragment of code that will be evaluated in each application of the pattern. Figure 2 shows an example of expression in our PETITSQL pattern. The goal of this expression is to retrieve the node in `ASTNodesParser` which calls the method named `filter`. From the example in Listing 2, the retrieved node would be `element.third()`.

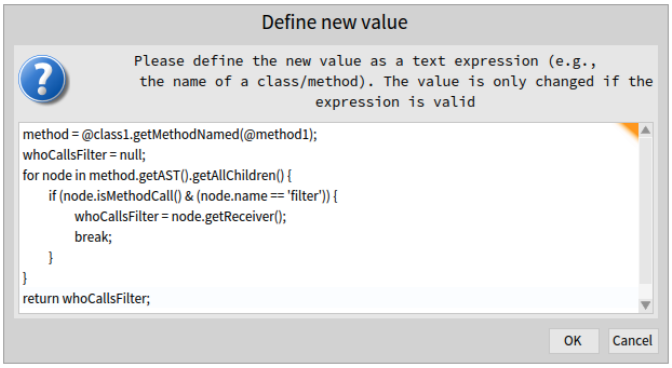

Fig. 2. Configuring parameters in MACRORECORDER (right click on a parameter, see Figure 1, right panel). Existing parameters (e.g., `@class1` and `@method1`) are referenced with their identifiers.

With this information, one can further retrieve the code in `ASTGrammar` to extract a new method from it. Therefore, given a new method (e.g., `alterTableConstraint`), the expression will be evaluated again and might return a different result.

At the end of this stage, the expected result is an abstract transformation that can be instantiated to different contexts.

C. Replaying transformations

Ultimately, after generalizing all the necessary parameters, MACRORECORDER can perform the transformation pattern in other contexts. The replaying stage is semi-automatic, and it is separated in three steps.

First, currently in MACRORECORDER, the developer must indicate explicitly the class (or method, statement, ...) where the recorded sequence of changes must be replayed. Obviously, the new code location shares properties with entities that changed in the initial recording. Otherwise, the execution of the sequence of changes would lead to no result. In PETITSQL example, the *root* parameter is the method which invokes `filter`. This fact means that the other parameters have common properties with this method. These properties are expressed in the generalization stage.

From the manual selection, the tool (i) infers which parameter may receive this code entity as a value (e.g., the first parameter defined as a method), and (ii) opens the panel depicted in Figure 1. In this example, the value of the parameter named `@method1` will change to the method which the developer selected. We discuss future work on automating this process in Section VIII.

Second, from this new context, the tool automatically computes the value for all the parameters. The parameters are necessary for the execution of the transformation operators. If the developer described an expression for any parameter, the tool will execute the expression and collect its result.

And third, MACRORECORDER performs the transformation operators and their parameters. If no parameter was generalized, MACRORECORDER will try to replicate the same operators as they were initially recorded. Figure 3 shows the result of the replication of the pattern in another method, called `alterTableConstraintFK`.

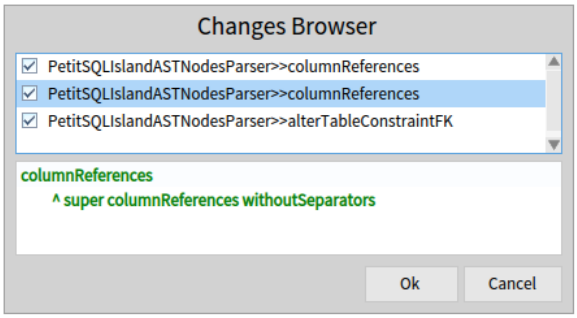

Fig. 3. Result of application of a transformation pattern in MACRORECORDER. Specially in this case, the method `columnReferences` was already extracted in `ASTGrammar`, therefore no changes were performed in this class.

In this method, the filtered element is the fifth of the collection, and it was called `columnReferences`, which was already extracted in the `ASTGrammar` class. Therefore, the resulting transformations as shown in the figure consist in: (i) creating the method in the parser, (ii) referencing the method in superclass and adding call to `withoutSeparators` and finally, (iii) removing the call to `filter` in the selected method.

At this point, the tool shows to the developer the resulting changes in the new context. The developer can check whether the changes are correct and accept them. During this process, if there is an execution error when performing at least one transformation operator, the whole transformation pattern is discarded. This fact means that MACRORECORDER rolls back changes done before the failing operator.

IV. MACRORECORDER'S APPROACH

In this section, we present MACRORECORDER's approach to record, generalize, and replay transformation patterns. The current implementation of the tool is developed in Smalltalk. However, the approach itself can be applied to other languages, such as Java. For each stage discussed in Section III, the approach has specific requirements, listed as follows:

- a code change recorder. The recorder is an extension of an IDE (e.g., ECLIPSE, PHARO) which is responsible to monitor editing activity and store code changes as events;
- an IDE that provides support to inspect source code entities and manipulate their underlying code automatically (for parameter generalization); and
- a code transformation tool (e.g., ECLIPSE's refactoring tools, REFACTORING in Pharo). The transformation tool will be extended to provide replication of each recorded code change event.

The approach is highlighted in grey in Figure 4. Specifically in Record and Replay stages, the existing tools have been extended to fit our approach requirements.

A. Recording Code Changes

MACRORECORDER relies on EPICEA [2], a tool that records developer events in the Pharo IDE, including code changes. Specifically, EPICEA listens to events concerning code changes occurring in the Pharo IDE and represents them as added, modified, and deleted nodes of the language's AST. EPICEA also records automated refactorings from the Pharo's refactoring tool. Similarly in Java environment, REPLAY is a tool that records activities from the developer, including code changes [3]. The events recorded by EPICEA are partially shown in Figure 5.

Our approach extends EPICEA's change model to record more detailed code changes. Specifically, we extended class change events to consider attributes and inheritance. And similarly, we extended method modifications to also consider arguments, temporary variables, and statements (i.e., assignments, return statements, for example). As discussed in Section III-A, the recording stage operates in the background, while the developer is editing the code manually.

B. Generalizing Transformation Operators

For each code change event extracted in EPICEA, MACRORECORDER creates a transformation operator. A transformation operator defines: (i) the type of the change; and (ii) the list of parameters that characterize the change.

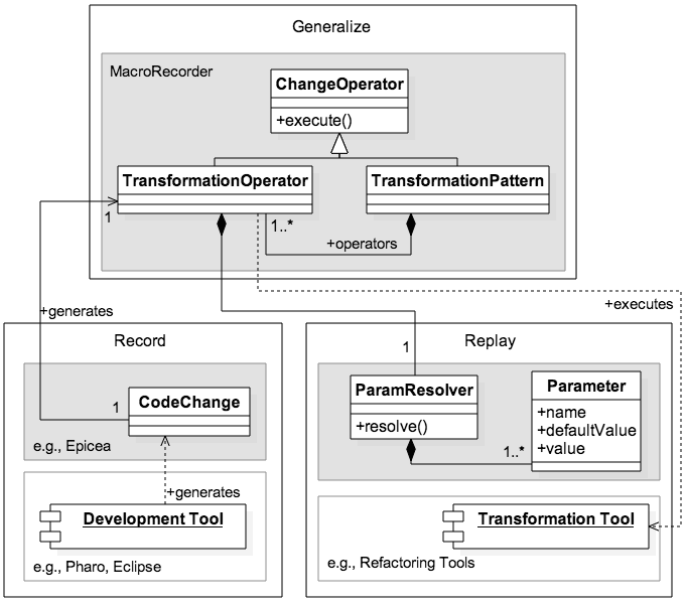

Fig. 4. Overview of MacroRecorder's approach (highlighted in grey). The transformation operator establishes the connection between recorded code change events in EPICEA and code edition algorithms in the transformation tool. A transformation pattern is a special type of operator that contains (and eventually executes) a collection of transformation operators.

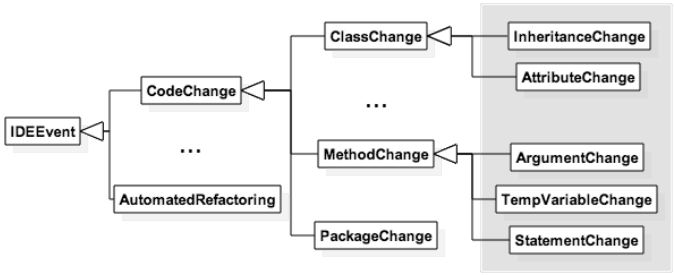

Fig. 5. Events recorded by EPICEA and MACRORECORDER from Pharo IDE. Events highlighted in grey were extended from the EPICEA's original change model. Each "Change" event defines addition, modification, and removal events. "Statement" events summarize Assignment, Expression, Method Call, and Return statements, for example.

First, MACRORECORDER generalizes the code change events. Therefore, the change type references an algorithm that performs the code change automatically. MACRORECORDER establishes the link between (i) a code change recorded in EPICEA and (ii) the algorithm that automatically performs the same code change in the transformation tool.

Second, MACRORECORDER generalizes the parameters. The list of parameters is stored in a *parameter resolver*. A parameter resolver basically describes the parameters that are necessary to replay the transformation operator. As shown in Section III-A, each parameter has a unique name and an associated code entity. The default value for this entity is extracted from the recorded events, but it can be redefined as expressions by the developer (see Section III-B).

Finally, MACRORECORDER generates a transformation pattern that represents the sequence of recorded changes. Specifically, the transformation pattern has a reference to the resolvers of its containing operators. We address the resolver in the next section to describe the instantiation of a transformation pattern in a different context.

C. Replaying Transformation Patterns

Ultimately, a transformation pattern is executed with assistance of a transformation tool. MACRORECORDER relies on the REFACTORING tool in Pharo. As discussed in Section III-C, MACRORECORDER executes the transformation operators in sequence. Each transformation operator (i) references the algorithm in the transformation tool that performs transformation automatically; and (ii) it has a parameter resolver which contains the parameters necessary to perform the transformation.

In order to obtain the new context, i.e., the entities to change in the new code location, the resolver calculates the value of all the parameters. Algorithm 1 presents the parameter resolving stage. Specifically, a parameter can be an expression described by the developer. This expression may eventually reference other parameters.[3] In the case of expression referencing parameters (lines 4 to 9), the resolver asks the transformation pattern to calculate the values of these parameters (line 7). This way, a parameter can reference parameters from other operators. At the end of this step, the resolver returns, for a given parameter name, its code entity in the current context.

```
   Data: collection of parameters P
   Result: the collection of code entities in this new context
1  values = Collection.new();
2  for p ∈ P do
3      value = p.value();
4      if value.isExpression() then
5          declarations = value.asExpression().getParameters();
6          tempParameters = pattern.valueFor(declarations);
7          value = expression.calculate(tempParameters);
8      end
9      values.add(value);
10 end
11 return values
```

Algorithm 1: Evaluating parameter values in the resolver

After instantiating the new context, the operator uses the transformation tool to perform the code change automatically, given the list of parameters values returned by its resolver. The result of this process is discussed in Section III-C.

V. EVALUATION

In this section, we evaluate MACRORECORDER with real cases of sequences of code changes in software systems. Our evaluation follows the methodology used in related work [7]. We evaluate MACRORECORDER's *complexity* when one must generalize transformations manually (see Section V-C). We also evaluate the tool's *accuracy* to check whether the tool is able to record and replay the transformations, in comparison to systematic manual edition (Section V-D).

[3]In Figure 2, the expression references two parameters, named `@class1` and `method1`.

A. Dataset

Our dataset is based on previous work on the existence of repetitive sequences of code changes related to rearchitecting [15]. It consists of six examples in three Smalltalk systems. Specially for one system (e.g., PACKAGEMANAGER), there are four examples of repetitive changes. These examples were described by the authors of this paper, with the assistance of the experts of each system. Table I summarizes descriptive data about the systems.

TABLE I
SIZE METRICS OF OUR DATASET. EACH LINE DESCRIBES A REARCHITECTING BETWEEN TWO VERSIONS. METRICS ARE SHOWN IN PAIRS (BEFORE AND AFTER THE REARCHITECTING).

	Packages	**Classes**	**KLOC**
PetitDelphi 0.210 / 0.214	7/7	313/296	8/9
PetitSQL 0.34 / 0.35	1/2	2/2	0.3/0.4
PackageManager 0.58 / 0.59	2/2	117/120	2.5/2.3

Each system represents a rearchitecting between two versions. We use the source code *before* the rearchitecting effort to record one sequence of code changes using MACRORECORDER. This recording will produce a transformation pattern for each of the six examples. In order to check the accuracy of the tool, we use the source code *after* the rearchitecting as our gold standard. Therefore, we calculate for each pattern occurrence[4], the similarity between (i) the actual source code after rearchitecting, and (ii) the result of automatic application of a transformation pattern by MACRORECORDER.

Table II presents descriptive data about our dataset. It describes, for each example: the number of occurrences of the pattern as originally calculated in previous work [15], the number of operators obtained after the recording stage, and the number of parameters as calculated automatically by MACRORECORDER. We reference the number of occurrences as a metric ($occurrences(P)$) in Section V-B. We discuss the number of parameters we actually configured in Section V-C.

TABLE II
DESCRIPTIVE METRICS OF TRANSFORMATION PATTERNS IN OUR DATASET. IT IS WORTH NOTING THAT NOT ALL OF THE PARAMETERS NEED TO BE CONFIGURED. WE PRESENT COMPLEXITY RESULTS IN SECTION V-C.

Transformation patterns	Pattern occurrences	Number of operators	Number of parameters
PetitDelphi	21	2	3
PetitSQL	6	3	6
PackageManager I	66	2	7
PackageManager II	19	3	5
PackageManager III	64	2	4
PackageManager IV	7	3	5
Average	30	3	5

[4]In this evaluation, we define transformation pattern occurrence as one instance of automatic transformation by MACRORECORDER.

B. Evaluation Metrics

In this section, we present the metrics we use in the evaluation of our approach. Before executing MACRORECORDER for each example in our dataset, we first need to generalize the transformation pattern. As discussed in Section IV-B, we configure the parameters in a transformation pattern with expressions. Therefore, we iteratively configure as many parameters as needed until MACRORECORDER automatically performs the transformation pattern with success.

- **Number of configured parameters** is the number of parameters the developer needs to generalize in order for MACRORECORDER to perform changes in a new context. This metric relates to the *complexity* of the approach.

After applying the transformation patterns automatically, each pattern occurrence can change the source code in different locations. In order to evaluate each pattern occurrence, we first categorize them according to their resulting code. The classification is non-exclusive and it is described as follows:

- **Matched** is an occurrence in which all the transformation operators were performed. This category relates to the ability of the approach to instantiate a new context;
- **Compilable** is an occurrence in which the resulting source code is syntactically correct. This category relates to the ability of the approach to not break the code;
- **Correct** is an occurrence in which the resulting code is behavior-equivalent to the gold standard. We make this classification by manual code inspection. This category relates to the ability of the approach to transform code that is accurate to the developer's manual edition.

Consequently, consider a transformation pattern P with $occurrences(P)$ occurrences. Therefore, $matched(P)$ is the number of occurrences in a pattern P that matched in a context. Similarly, $compilable(P)$ and $correct(P)$ calculate the number of occurrences that are compilable and correct, respectively. The following metrics are proposed in related work on automated code transformation [7].

$$coverage(P) = \frac{matched(P)}{occurrences(P)} \quad (1)$$

$$accuracy(P) = \frac{correct(P)}{occurrences(P)} \quad (2)$$

Therefore, coverage measures the percentage of the occurrences from which MACRORECORDER was able to instantiate a new context and perform the transformations. Moreover, accuracy measures the percentage of occurrences in which the modified code is equivalent to the result of manual edition.

Since a correct pattern occurrence is also compilable (but not always the opposite), we calculate for compilable occurrences, the similarity between the result of manual and automatic transformations. For each changed code entity, we calculate its AST tree c. Therefore, given the results of both manual (c_{manual}) and automatic (c_{auto}) transformations, similarity is defined as:

$$similarity(c_{manual}, c_{auto}) = \frac{|(c_{manual} \cap c_{auto})|}{|(c_{manual} \cup c_{auto})|} \quad (3)$$

Thus, similarity is also a percentage metric. Similarly to coverage and accuracy, the similarity in a transformation pattern P calculates the average similarity to all the code entities modified in all the occurrences of this pattern.

C. Complexity Evaluation

In this evaluation, we investigate the complexity to generalize a transformation pattern with MACRORECORDER. We perform this generalization by manual definition of expressions for the parameters in each pattern. In this case, Table III describes the number of parameters we had to modify explicitly to make the pattern applicable for all of its occurrences.

TABLE III
NUMBER OF CONFIGURED PARAMETERS FOR EACH TRANSF. PATTERN

Transformation patterns	Configured parameters	Total Number of parameters
PetitDelphi	1	3
PetitSQL	1	6
PackageManager I	2	7
PackageManager II	2	5
PackageManager III	1	4
PackageManager IV	2	5

In general, we had to configure up to two parameters per pattern, even for the first pattern in PACKAGEMANAGER which has seven parameters (see Table II). Although a few parameters must be configured, some of them might be more complex to express. We discuss an example of expression we wrote for PETITSQL in Section III-B. In order to retrieve the right entity to change, the expression had to iterate over all of the AST nodes in a method.

We found one case in which MACRORECORDER is limited in the creation of more complex patterns. This limitation is also found in related work [7], and we found this case in the first pattern in PACKAGEMANAGER. Listing 3 present the modified code in one occurrence of this pattern.

Listing 3: Modified code in PACKAGEMANAGER

```
public dependencies() {
-   package.addDependency('Seaside-Core')
-     .addVersion(package.getVersion());
+   dependencies = new Collection();
+   dependencies.add( new Pair('Seaside-Core',
      '=3.1.0') );
+   return dependencies;
}
```

In this example, packages are represented as data objects. The method `dependencies()` defines the packages on which a package depends. The developers decided that packages should not be modified with setter methods. Therefore, instead of creating instances for each depending package (using `addDependency` and `addVersion`), the developer must define a simple pair of strings. One pair defines the name of the package and the version on which the modified package depends. After generalizing the pattern with MACRORECORDER, we applied it automatically to all the data objects that implements a method called `dependencies()`.

However, some methods instantiate more than one dependency. In MACRORECORDER, the number of operators in pattern is limited to the ones the developer records. In this example, when replaying the changes with MACRORECORDER, the tool only changed the first occurrence of the pattern. Therefore, the resulting transformation is incorrect (see Section V-D). In practice, we could apply the pattern repetitively in the same method until all instantiations of dependencies are removed. However, we come back to the same problem of performing tedious, repetitive tasks. MACRORECORDER must also support the repetition of a subset of the pattern.

Summary: Up to two parameters were necessary to generalize the patterns. However, their corresponding expressions might be complex to define. The results show that, with few limitations, MACRORECORDER *successfully creates parameterizable transformation patterns.*

D. Accuracy Evaluation

We now investigate whether MACRORECORDER's automated code transformation is close to repetitive manual transformations performed by the developer. Table IV summarizes the results, using the metrics we defined in Section V-B.

TABLE IV
ACCURACY RESULTS. WE DESCRIBED THE METRICS IN SECTION V-B

	occurrences	*matched*	*compilable*	*correct*	*coverage (%)*	*accuracy (%)*	*similarity (%)*
PetitDelphi	21	21	21	21	100	100	100
PetitSQL	6	6	6	4	100	66	85
PackageManager I	66	50	50	11	76	17	20
PackageManager II	19	14	14	14	74	74	100
PackageManager III	64	64	64	64	100	100	100
PackageManager IV	7	7	7	7	100	100	68
Average					92	76	79

In general, 92% of the pattern occurrences matched. We discussed in Section V-C one of the examples in which part of the pattern is not executed. We observed the same outcome in the second pattern in PACKAGEMANAGER. For such cases, we might increase the coverage of the results by (i) adding an additional condition or (ii) repeating a subset of operators in the same code location.

For accuracy, 76% of the automatic transformations are behaviour-equivalent to developer's manual edition. Similar to coverage results, the accuracy is lower in PACKAGEMANAGER and PETITSQL patterns because of their small variations. Consequently, in these cases the similarity is also low.

Finally, the result of automatic transformation is 79% similar to developer's manual edition. The patterns in PETITDELPHI and PACKAGEMANAGER (the second and third ones) had the best similarity in the study. In PETITDELPHI, the pattern consists in removing methods and classes from the system. Our tool covered all of these occurrences. In PACKAGEMANAGER, the pattern creates methods with only one statement. Therefore, from the occurrences that matched the context, MACRORECORDER replays them exactly like the developer.

Specially in the fourth pattern in PACKAGEMANAGER, even though MACRORECORDER produced correct transformations, the output is not completely similar to the version which was manually edited by the developer. Specifically in this case, the similarity is lower because of an Add Statement transformation. This operator needs to calculate the position in the method's AST where the statement will be added. This calculation is necessary because the operator assumes that other methods can be very different from the recorded change.

Currently, the operator uses data analysis to collect all the variables that the statement uses. This operator puts the statement after the declarations of its dependent variables. If no variable declarations are found, the operator puts the statement as the first one of the method. Listings 4 and 5 present an example of modified code in this case.

The first modification was performed by the developer, retrieved from source code history; the second one is result of automatic transformation using MACRORECORDER. In this example, the correct statement was automatically removed from code. However, to add the new statement, the transformation operator calculated a different location. In the transformation tool, the transformation operators are independent, i.e., they do not store information about the operators that were performed before. However, we consider that the code is still correct although it is slightly different from the developer's code.

Summary: Most of the automated transformation patterns are correct, they cover most of the pattern occurrences, and the result of automatic transformation is often similar to manual edition. With the assistance of MACRORECORDER*, it is possible to replicate transformation patterns in different code locations.*

E. Discussion

In comparison to MACRORECORDER, SYDIT [7] is the most similar tool in related work. Both tools rely on one example of change to describe a transformation pattern, and the destination of automatic transformations must be defined manually by the user. We discuss more about SYDIT in Section VII. SYDIT performs transformations automatically with 82% coverage and 70% accuracy. In our evaluation, we showed that MACRORECORDER has 92% coverage and 76% accuracy.

However, we cannot directly compare these tools for two reasons. First, SYDIT's examples were applied to bug fixes. In fact, the tool is limited to in-method editions. Although we support in-method transformation operators (e.g., Add Statement), our transformations have a higher level of abstraction, i.e., we can modify methods, classes, and the dependencies between them in a single pattern.

Listing 4: Result of manual edition

```
public testResolvedDependency() {
  // ...
  solver = new Solver().add(repository);
  dependency = new Dependency()
    .setPackage(package)
-     ;
+     .setVersion(new VersionConstraint('3.1');
  // ... }
```

Listing 5: Result of automatic transformation

```
public testResolvedDependency() {
  // ...
+  dependency = new Dependency()
+     .setPackage(package)
+     .setVersion(new VersionConstraint('3.1');
  solver = new Solver().add(repository);
-  dependency = new Dependency()
-     .setPackage(package);
  // ... }
```

VI. THREATS TO VALIDITY

Construct Validity: The construct validity is related to how well the evaluation measures up to its claims. Specifically in the complexity evaluation, we do not rely on real developers using the tool. The conclusions in this evaluation are based on our use of the prototype. However, we previously discussed that the implementation of the tool is a proof-of-concept one. The evaluation concerns the limitations of an automated support when applying abstract code transformations. We intend to evaluate the usability of the tool in future work.

Internal Validity: The internal validity is related to uncontrolled factors that might impact the experimental results. In our study, we discussed examples in which the code generated by automatic transformation is not similar to manual edition, but it is considered correct (see Listings 4 and 5). In the case of method transformations, we do not consider side effects caused by replacing code in different locations compared to the developer's edition. For example, the added assignment statement in Listing 5 might execute code that will impact the instantiation of a `Solver`. Therefore, the code resulting from the automated transformation might also be incorrect.

Three factors alleviate this threat. First, this case only happened in this PACKAGEMANAGER's example. Other examples are more straightforward, e.g., the transformation creates a method and adds some statements in it. The resulting code in these cases are very similar to the gold standard. Second, the statement is an assignment, followed by the instantiation of an object and a couple of calls to setter methods. Therefore we assume that, at best, the behavior was not affected. And third, in a practical setting, MACRORECORDER shows the result of the automated transformations before actually changing the code (see Section III-C). The developer can discard the changes if they are not correct, and the code is not compromised.

However, we have two suggestions to avoid this threat. First, we can use tests to check whether the behavior changed, if available in the target system. And second, we might also

analyze dependencies between transformation operators in order to perform them in cascade. For example in Listing 5, the Add Statement operator could consider the position in which the previous Remove Statement operator was applied.

External Validity: The external validity is related to the possibility to generalize our results. In our study, the main threat consists in whether repetitive sequences of code changes exist in real software systems. In previous work [15], we found eleven sequences in small and large systems. Due to language constraints, we study six of these sequences in this paper. Moreover, most of the systems under analysis, as well as most of their sequences of code changes were very small. We acknowledge this issue.

First, we do not claim that these were the only repetitive sequences of changes in these systems. In fact, to identify sequences of similar code changes and to express their properties is a very challenging task. However, even in a small system such as PACKAGEMANAGER there were sequences with up to 60 occurrences. Despite the limitations of our tool, we obtained interesting results in these systems. We intend to develop a MACRORECORDER version for Java, to which we can replicate our study with more transformations.

Second, the sequences of code changes are small. However, in order to generalize their corresponding transformations in MACRORECORDER, some parameters were very complex to define in a programatic way. In most of the examples, we needed to iterate over a method's AST to retrieve a specific code entity and change its code. In general, our results show that the size of the system is not an issue in terms of complexity. We could identify and replay both repetitive and complex sequences of code changes with success.

VII. RELATED WORK

Demonstrational Programming is a term mostly used in robotics to comprise approaches that identify and automate repetitive sequences of operations in a given context. These approaches do not depend on a formal definition to describe and/or automate these sequences of operations.

According to this paradigm, the user provides concrete examples of how to perform the operations; then the computer must repeat the operations afterwards, in different contexts. As concrete examples, the feature *Search and Replace* in most text editors, the multiple selections feature in SUBLIMETEXT, and the macro feature in MICROSOFT OFFICE programs are examples of programming by demonstration approaches.

In order to compare Demonstrational Programming approaches with MACRORECORDER's, we separated them in two categories: approaches that identify transformation patterns from source code, and approaches that automate transformation patterns in different code locations.

A. Transformation Pattern Identification

In this section, we introduce approaches that mine source code repositories to identify patterns of code changes. Therefore, these approaches rely on the fact that repetitive changes were already applied in the past. Most of these approaches do not provide automations for the patterns they identify. Some examples are related to bug fixes [13], API evolution [1], and well-known refactorings applied to methods [9].

Jiang et al. [5] identify similar code changes that may involve different releases in the code repository. They identify the operators involved but they do not generalize them. On the other hand, Kim et al. [6] generalize repetitive changes in *change rules*, which are composed of an application condition, a set of exceptions, and the sequence of change operators in terms of added and removed lines. Their approach has a limited set of operators. Moreover, their condition is only based on structural dependencies (e.g., all classes in a given package). However, we intend to extend the generalization in MACRORECORDER to provide the automatic application of a pattern into an entire system, given an application condition.

B. Transformation Pattern Application

In this section, we selected tools that generalize and automate the application of transformation patterns. Therefore, the approaches we present rely on the fact that the developer is aware of a repetitive code transformation task. Consequently, a transformation pattern must be applied, either automatically or with some assistance from the developer.

The first tool that applies programming by demonstration in the context of code transformation is SYDIT [7]. SYDIT relies on one example (e.g., one method before and after a code change). The tool generates an *edit script* in terms of added, removed, and modified AST nodes. Moreover, the tool uses control and data dependence analysis to calculate the *context* of the change. The context extracts the properties of the changed entities to assist the application of the script in new locations. Finally, the tool depends on the developer to explicitly indicate where the script will be applied.

The LASE tool is an extension of SYDIT [8]. As opposed to SYDIT, LASE relies on two or more code change examples from the developer. The tool generalizes or specifies the edit script depending on the differences between these examples. For example, if the same transformation is applied to two different AST nodes, the approach generalizes the node as a parameter. However, the accuracy of the resulting edit script depends on the quality of the dependence analysis (for SYDIT) or the given examples (in LASE). The script will not be able to find small variations of candidate cases (over specification), or the script will be too general and it might be applied to undesired candidates (over generalization). MACRORECORDER's contribution consists on the parameterization of code entities, which is customizable by the developer.

CRITICS is a tool which relies on one change example [17]. Similarly to MACRORECORDER, CRITICS is based on generalization of the example by the developer. In practice, the developer incrementally generalizes the example by parameterizing types, variables, and method names. This parameterization is similar to our approach, but it is restricted in the set of entities that can be generalized.

The three approaches differ from MACRORECORDER in two more aspects. First, their evaluations relied on changes related to bug patches, which generally comprise very localized operators. Our case study included operators with higher level of granularity, which included from update statements to change the hierarchy of classes [15]. And second, LASE and CRITICS use the context of the change to find new change opportunities. In previous work, we concluded that inferring an application condition from changes is very complex. MACRORECORDER allows the developer to indicate where the changes will be applied, either directly (similar to SYDIT) or in the future by using a condition which will select the code entities automatically.

VIII. CONCLUSIONS

During a large maintenance effort, developers sometimes perform repetitive sequences of changes on similar code entities. Due to this repetition, these sequences are tedious to perform. Moreover, these sequences are error-prone due to their manual application.

In this paper, we presented MACRORECORDER, a proof-of-concept tool to record and replay sequences of source code changes. In MACRORECORDER, one can record the sequence of changes once, generalize the transformations, and then perform the sequences automatically in other code locations. We discussed our approach which extends development and transformation tools, in order to record code change events that are parametrizable and replicable.

Our case study consisted in real cases of repetitive code changes. The sequences of changes in our study have five parameters in average. To generalize the transformations, we had to generalize up to two parameters. However, the generalization was complex. However, MACRORECORDER was able to perform 92% of the examples with 76% accuracy. The source code resulting from automatic transformation is 79% similar to manual code edition. We discussed specific features in our case study that MACRORECORDER can overcome in order to improve its accuracy. The evaluation leads to the conclusion that customizable, system specific, source code transformations can be automated.

For future work, we propose the automation of the definition of transformation patterns. First, we propose to automate the generalization of the parameters. Currently, MACRORECORDER relies on the manual definition of expressions by the developer. In this context, we propose to analyze the dependencies between parameters in order to infer abstract expressions automatically (e.g., extract the AST of a method and filter some of its nodes).

Second, we propose to automate the application of the transformations. Such support would be provided by an application condition. This condition will check, out of all the code entities in the system, which entities are candidate to be changed. This application condition would assist the application of transformations in all of the opportunities at once.

REFERENCES

[1] J. Andersen and J. L. Lawall. Generic patch inference. In *23rd International Conference on Automated Software Engineering*, pages 337–346, 2008.

[2] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stephane Ducasse. Untangling fine-grained code changes. In *22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 341–350, 2015.

[3] Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. Software evolution comprehension: Replay to the rescue. In *19th International Conference on Program Comprehension*, pages 161–170, 2011.

[4] Muhammad Javed, Yalemisew Abgaz, and Claus Pahl. Composite ontology change operators and their customizable evolution strategies. In *Workshop on Knowledge Evolution and Ontology Dynamics, collocated at 11th International Semantic Web Conference*, pages 1–12, 2012.

[5] Qingtao Jiang, Xin Peng, Hai Wang, Zhenchang Xing, and Wenyun Zhao. Summarizing evolutionary trajectory by grouping and aggregating relevant code changes. In *22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 1–10, 2015.

[6] Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson Jr. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, 2013.

[7] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *32nd Conference on Programming Language Design and Implementation*, pages 329–342, 2011.

[8] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering*, pages 502–511, 2013.

[9] Narcisa Andreea Milea, Lingxiao Jiang, and Siau-Cheng Khoo. Vector abstraction and concretization for scalable detection of refactorings. In *22nd International Symposium on Foundations of Software Engineering*, pages 86–97, 2014.

[10] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *31st International Conference on Software Engineering*, pages 287–297, 2009.

[11] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *27th European Conference on Object-Oriented Programming*, pages 552–576, 2013.

[12] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *32nd International Conference on Software Engineering*, pages 315–324, 2010.

[13] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[14] Baishakhi Ray and Miryung Kim. A case study of cross-system porting in forked projects. In *20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[15] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Tulio Valente. System specific, source code transformations. In *31st International Conference on Software Maintenance and Evolution*, pages 1–10, 2015.

[16] Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara, and Ralph E. Johnson. A compositional paradigm of automating refactorings. In *27th European Conference on Object-Oriented Programming*, pages 527–551, 2013.

[17] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Interactive code review for systematic changes. In *37th Intl. Conference on Software Engineering*, pages 1–12, 2015.