Template-based Reconstruction of Complex Refactorings

Kyle Prete, Napol Rachatasumrit, Nikita Sudan, Miryung Kim
Electrical and Computer Engineering
The University of Texas at Austin
Email: {kylep, nrachtasumrit, nsudan}@mail.utexas.edu, miryung@ece.utexas.edu

***Abstract*—Knowing which types of refactoring occurred between two program versions can help programmers better understand code changes. Our survey of refactoring identification techniques found that existing techniques cannot easily identify complex refactorings, such as an *replace conditional with polymorphism* refactoring, which consist of a set of atomic refactorings.**

This paper presents REF-FINDER that identifies complex refactorings between two program versions using a template-based refactoring reconstruction approach—REF-FINDER expresses each refactoring type in terms of template logic rules and uses a logic programming engine to infer concrete refactoring instances. It currently supports sixty three refactoring types from Fowler's catalog, showing the most comprehensive coverage among existing techniques. The evaluation using code examples from Fowler's catalog and open source project histories shows that REF-FINDER identifies refactorings with an overall precision of 0.79 and recall of 0.95.

I. INTRODUCTION

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the modular structure of software [1]. Automatically identifying which refactorings happened between two program versions is an important research problem because inferred refactorings can help developers understand the modifications made by other developers and can be used to update client applications that are broken due to refactorings in library components [2], [3], [4]. Furthermore, they can be used to study the effect of refactorings on software quality empirically when the documentation about past refactorings is unavailable in software project histories [5].

Our survey of refactoring identification techniques found that, even though they can handle simple refactorings, such as *rename* and *move* refactorings, they cannot easily handle complex refactorings that consist of atomic refactorings related by structural constraints. In particular, they cannot easily identify (1) refactorings that require the knowledge of changes to the control structure of a program such as *replace conditional with polymorphism* (pages. 255-259 [6]), (2) refactorings that require information about changes within method bodies such as *replace temp with query* (pages. 120-123 [6]), and (3) refactorings that themselves consist of multiple previously identified refactorings such as an *extract superclass* refactoring (pages. 336-340 [6]), which consists of *move fields/methods*.

To overcome these limitations, we developed a refactoring reconstruction technique that actively leverages the domain knowledge about well-known refactoring types. As a first step, we targeted refactoring types in Fowler's catalog [6], a comprehensive list of refactorings, well understood by software engineering practitioners. Inspired by the prior work on logic-based program representation approaches [7], [8], [9], we described the structural constraints before and after applying a refactoring to a program in terms of *template logic rules* and encoded ordering dependencies among refactoring types to define which refactoring types must be identified before finding higher-level, composite refactorings. We then developed a fact extractor that traverses the abstract syntax tree of a program and extracts facts about code elements (packages, classes and interfaces, methods, and fields), structural dependencies (containment, overriding relationships, subtyping relationships, method calls, and field accesses), and the content of code elements (e.g., if-then-else control structures in a method-body). After representing a program in terms of a database of logic facts, REF-FINDER infers concrete refactoring instances by converting a template logic rule into a logic query, and then invoking the query on the database using a Tyruba logic programming system [10].

Consider a program that has undergone a *replace conditional with polymorphism* refactoring, which replaces a conditional logic that chooses different behavior depending on the type of an object. Each leg of the conditional is extracted to an overriding method in a subclass [6]. Invoking a logic query that checks the deletion of if-statement and movement of a content from the original method to a new method that overrides the original method finds a *replace conditional with polymorphism* refactoring.

To evaluate our system, we applied REF-FINDER to code examples from Fowler's book and compared REF-FINDER's output with the known refactoring instances to assess both *recall*—how many known refactorings were indeed found by REF-FINDER and *precision*—how many of found refactorings are indeed correct. Its precision and recall were 97% and 94% respectively. We also applied REF-FINDER to release pairs of Columba and jEdit and revision pairs of Carol. Since these programs did not document refactorings, we created a set of correct refactorings by running REF-FINDER with a similarity threshold (σ=0.65) and manually verified them. We then measured a recall by comparing this set with the results found using a higher threshold (σ=0.85) and measured precision by inspecting a sampled data set. The precision and recall on open source projects were 0.74 and 0.96 respectively.

The rest of this paper is organized as follows. Section II summarizes our survey of refactoring reconstruction approaches. Section III explains our methodology in detail. Section IV discusses evaluation results on Fowler's code examples and open source projects. Section V concludes with the description of future work.

II. Related Work

Demeyer et al. [11] first proposed the idea of inferring refactoring events by comparing two program versions based on a set of ten characteristic metrics, such as LOC and the number of method calls within a method. Zou and Godfrey [16] first coined the term, origin analysis, which serves as a basis of refactoring reconstruction by matching code elements using multiple criteria (e.g., names, signatures, metric values, callers, and callees). Zou and Godfrey infer merge, split, and rename refactorings. S. Kim et al. [15] used clone detectors such as CCFinder [22] to map methods. Malpohl et al. [12] align tokens using *diff* and infers a function or variable renaming when distinct tokens are surrounded by mapped token pairs. Van Rysselberghe and Demeyer [13] use a clone detector to detect moved methods. Antoniol et al. [14] identifies class-level refactorings using a vector space information retrieval approach. Xing and Stroulia's UMLDiff [5] matches packages, classes, interfaces, fields and blocks based on their name and structural similarity in a top-down order. After matching code elements, UMLDiff infers refactorings.

Refactoring Crawler [2] identifies refactorings in two stages. First, it finds a list of code element pairs using *shingles* (a metric-based fingerprint) and performs a semantic analysis based on reference relationships (calls, instantiations, uses of types, import statements). The second part of the algorithm is an iterative, fix point algorithm that finds refactorings in a top-down order. Weißgerber and Diehl's technique [17] identifies and ranks refactoring candidates using names, signatures, and clone detection results. Change Distiller [19] compares two versions of abstract syntax trees; computes tree-edit operations; and maps each tree-edit to atomic AST-level change types (e.g., parameter ordering change). Though it analyzes both declaration and method-body changes derived from AST edit operations, unlike Ref-Finder, it does not relate previously identified refactorings from multiple locations to infer higher-level refactorings. M. Kim et al. [21] automatically infer systematic declaration changes as rules and determine method-level matches. We extend this prior work by actively leveraging the structural constraints of a program before and after each refactoring type.

The columns of Table I represent individual approaches and the rows indicate refactoring types. Refactoring types that are currently implemented by each approach are marked by $\surd$; the refactoring types that can be supported by simple extensions are marked by $\diamond$. (Some of these extensions were mentioned in the sources themselves.) The refactoring types not mentioned in the table require considerable extensions or are not supported by them currently. Many of these existing techniques analyze code elements at or above the level of method headers and do not analyze changes to the control structure within method bodies. Thus, detection of *decompose conditionals* requires significant extensions to these algorithms. They also cannot detect composite refactorings easily because they do not know which refactorings must be detected first and how those refactorings must be knit together to detect higher-level, composite refactorings.

The approach most similar to Ref-Finder is Xing et al.'s change-facts *queries* [18]. They first extract facts regarding design-level entities and relations. These facts are then pairwise compared to determine how the facts changed from one version to the next. Finally, queries corresponding to well-known refactoring types are applied to the database to find concrete refactoring instances. This work is similar to Ref-Finder in that it explicitly encodes the skeleton of a refactoring type as a query. Because the queries are encoded in SQL, identification of composite refactorings may require manually weaving the results. Ref-Finder also provides a more *comprehensive* coverage by supporting 63 refactoring types from Fowler's catalog [6], as opposed to 32 queries mentioned in [18].

Spyware [23] captures refactorings during development sessions in an IDE rather than trying to infer refactorings from two program versions. Refactoring reconstruction can complement Spyware by finding refactorings that are not directly supported by IDEs.

Representing a program's code elements and structural dependencies as a set of logic facts has been used for decades. Grok [7] extracts facts about code elements and structural relationships in software and supports querying the resulting relational databases. CodeQuest [9] evaluates logic queries specified by programmers to assist program investigation. Mens et al.'s intentional view [8] allows programmers to specify concerns or design patterns using logic rules. Eichberg et al. [24] use Datalog rules to continuously enforce dependency constraints as software evolves. DeMIMA [25] finds concrete instantiations of design patterns by matching the skeleton of design patterns against a program structure. Tourwé et al. [26] use logic meta-programming to detect bad code smells. Similar to the aforementioned approaches, Ref-Finder uses a logic-based representation and querying approach; while these approaches support source code navigation, detect design pattern instantiations, detect refactoring opportunities, or check structural constraints *in a single program version*, Ref-Finder focuses on identifying *refactorings that occurred between two program versions*.

III. Methodology

Ref-Finder takes two program versions as input and finds refactoring instances automatically by leveraging the catalog of template refactoring rules. Section III-A describes how Ref-Finder encodes both old and new program versions as a database of logic facts and each refactoring type as a template logic rule. Section III-B describes how Ref-Finder finds refactoring instances by converting template logic rules into logic queries and invoking the queries on the logic

TABLE I: Refactoring types currently supported by existing refactoring reconstruction approaches. The remaining 40 refactorings in Fowler's catalogue that are not mentioned in the table are not handled by any of the existing techniques.

Refactorings	Demeyer [11]	Malpohl [12]	Van Ryssel-berghe [13]	Antoniol [14]	S.Kim [15]	Zou [16]	Dig [2]	Weiβgerber [17]	Xing [5], [18]	Fluri [19]	Dagenais [20]	M.Kim [21]
Extract Method	√	√	◇	◇	◇	√	√	√	√	√	√	◇
Extract Subclass	√								√			
Move Class	√		√	√	√	√	√	√	√	√	◇	√
Move Field	√		√	√	√	√	√	√	√	√	◇	√
Move Interface	√		√	√	√	√	√	√	√	√	◇	√
Move Method	√		√	√	√	√	√	√	√	√	√	√
Rename Method	√		√	√	√	√	√	√	√	√	√	√
Replace Package	√	◇	√	◇	√	√	√	√	√	√	◇	√
Replace Class	√	√	√	√	√	√	√	√	√	√	◇	√
Replace Return	√	◇	◇	◇	◇	√	√	◇	◇	√	◇	√
Replace Input Signature	√	◇	◇	◇	◇	√	√	◇	√	√	√	◇
Add Parameter	√	◇		◇	√	◇	√	√	√	√	◇	√
Extract Superclass	√								√			
Pull Up Field	√								√			
Pull Up Method	√								√			
Push Down Field	√								√			
Push Down Method	√								√			
Remove Parameter	√	◇			√	√	√	√	√	◇	◇	√
Hide Method	◇	◇					√	√	√	◇	◇	
Unhide Method	◇	◇					√	√	√	◇	◇	
Extract subsystem/package	◇						◇	◇	√	◇	◇	◇
Inline subsystem/package	◇			◇			◇	◇	√	◇	◇	◇
Extract interface		◇	◇	◇				√	√	◇	◇	
Inline superclass									√			
Inline subclass									√			
Form Template Method									√			
Replace inheritance with delegation									√			
Replace delegation with inheritance									√			
Inline Class									√		√	√
Convert anonymous class to inner class									√			
Introduce factory method									√			
Introduce parameter object									√			
Encapsulate field									√			
Preserve whole object	◇								√	◇		◇

database. The queries are invoked in order such that prerequisite refactorings are found before high-level, composite refactorings. Section III-C illustrates the inference process using a *decompose conditional* example.

A. Logic-based Representation and Refactoring Rules

Predicate definition. REF-FINDER represents each program version using a set of logic predicates (see Table II). Some of the predicates describe code elements (packages, classes, interfaces, types, methods, and fields) and their containment relationships; Some describe structural dependencies (field-access, method-call, subtyping, and overriding). These predicates about code elements and structural dependencies are adopted from our prior work on LSdiff [27]. Because our prior work only described the content of method bodies in terms of method-calls and fields-accesses, we added six predicates, conditional, cast, trycatch, throws, variabledefinition, and methodbody, to analyze *the internal content of method bodies*, control-structures and variable definitions. For example, predicate conditional was added to encode `if` statements within a method body. We also added four predicates to capture the modifier of methods/fields and getter or setter methods. These predicates were identified as we needed them to encode the skeleton of each refactoring type from Fowler's catalog as rules. REF-FINDER traverses the abstract syntax trees of a program and extracts logic facts using the Eclipse JDT analysis toolkit [30].

Some logic facts are derived from previously identified facts. For example, inheritedmethod facts are derived from both method declaration facts and subtyping relationship facts. After extracting facts from both old and new versions (denoted as before_ and after_ respectively), REF-FINDER identifies deleted_ and added_ facts by comparing the factbases. Some facts are computed on demand to assist our refactoring reconstruction process. For example, in order to determine an extract_method refactoring, REF-FINDER generates a similarbody fact if the similarity between two candidate methods is above a threshold σ.

TABLE II: Logic predicates used to model code elements, structural dependencies, and the content of code elements in Java

Basic Predicate Adopted from LSdiff [27]	**Interpretation**
package(packageFullName)	There exists a package with `packageFullName`.
type(typeFullName, typeShortName, packageFullName)	A class or an interface with name typeShortName is in `packageFullName` package.
method(methodFullName, methodShortName, typeFullName)	A method with name `methodShortName` is in `typeFullName` type.
field(fieldFullName, fieldShortName, typeFullName)	A field with name `fieldShortName` is in `typeFullName` type.
return(methodFullName, returnTypeFullName)	A method `methodFullName` returns type `returnTypeFullName`.
fieldoftype(fieldFullName, declaredTypeFullName)	A field `fieldFullName` is declared to be `declaredTypeFullName` type.
typeintype(innerTypeFullName, outerTypeFullName)	An inner class `innerTypeFullName` is declared in class `outerTypeFullName`.
accesses(fieldFullName, accessorMethodFullName)	A field `fieldFullName` is accessed by method `accessorMethodFullName`.
calls(callerMethodFullName, calleeMethodFullName)	A method `callerMethodFullName` calls a method `calleeMethodFullName`.
subtype(superTypeFullName, subTypeFullName)	A `subTypeFullName` class is a subtype of `superTypeFullName`.
inheritedfield(fieldShortName, superTypeFullName, subTypeFullName)	A `fieldShortName` field is inherited by class `subTypeFullName` from class `superTypeFullName`.
inheritedmethod(methodShortName, superTypeFullName, subTypeFullName)	A `methodShortName` method is inherited by class `subTypeFullName` from class `superTypeFullName`.
Extended Predicates (A full list of predicates is available in [28])	**Interpretation**
methodbody(methodFullName, methodBody)	A method `methodFullName` has a content `methodBody` block.
conditional(condition, thenPart, elsePart, methodFullName)	An `if-then-else` statement inside method `methodFullName` has `thenPart` block and `elsePart` block with a `condition` expression..
cast(expr, typeFullName, methodFullName)	A method `methodFullName` contains expression `expr`, which has been cast to type typeFullName.
trycatch(tryBlock, catchClauses, finallyBlock, methodFullName)	A method `methodFullName` contains a `try-catch` statement with `tryBlock` block, `catchClauses` clauses, and `finallyBlock` block.
throws(methodFullName, exceptionFullName)	A method `methodFullName` throws an exception of type `exceptionFullName`.
variabledeclaration(methodFullName, varName, varType, expr)	Within a method `methodFullName`, a variable with the identifer `varName` is declared to be type `varType` and assigned to expression `expr`.
methodmodifier(methodFullName, modifierType)	The modifier of `methodFullName` method is modifierType.
fieldmodifier(fieldFullName, modifierType)	The modifier of `fieldFullName` field is `modifierType`.
parameter(methodFullName, paramList)	A method `methodFullName` has an input signature, `paramList`.
getter(methodFullName, fieldFullName)	The method `methodFullName` returns the value of the field `fieldFullName` .
setter(methodFullName, fieldFullName)	The method `methodFullName` sets the value of the field `fieldFullName` .
similarbody(method1FullName, method1Body, method2FullName, method2Body)	The methods `method1FullName` and `method2FullName` have method bodies and their content similarity is above a threshold σ.
addedparameter (methodFullName, argName, argType)	A method `methodFullName` has a new input argument `argName` of type `argType`.
deletedparameter (methodFullName, argName, argType)	A method `methodFullName` deleted an input argument `argName` of type `argType`.

For this purpose, we have implemented a rudimentary block-level clone detection technique, which removes any beginning and trailing parenthesis, escape characters, white spaces and *return* keywords and computes word-level similarity between the two texts using the longest common subsequence algorithm [31]. This type of fact extraction can be replaced with existing clone detection techniques such as CCFinder [22]. To identify *add/remove parameter* refactoring, for each deleted method, we search for a new method with the same name, and compared its signature to identify input signature changes.

Coding refactoring types as template logic rules. We encode each refactoring type as a logic rule. A consequent predicate represents a target refactoring type to be inferred and the predicates in the antecedent represent pre-requisite refactorings or change-facts. For example, the *extract method* rule, added_method(m2, n2, t2) $\wedge$ after_method(m1, n1, t1) $\wedge$ similar_body(m2, b2, m1, b1) $\wedge$ after_calls(m1, m2) $\rightarrow$ extract_method(m1, m2, b2, t1), represents that an *extract_method* refactoring requires that a new method m2's body content b2 is extracted from method m1 in the old version and that m1 now calls m2.

We manually encoded 63 refactoring types in Fowler's catalog as rules. Complex refactorings are described using other refactorings as pre-requisites. Table III shows the logic-rule based representations for six refactorings out of 63 refactorings in the technical report [28].

We excluded *convert procedural design to objects*, *substitute algorithm*, *duplicate observed data*, *introduce foreign method*, *replace record with data class*, and *separate domain from presentation* because they either are too ambiguous or require significant knowledge about mapping from design to code. We excluded *split temporary variable* and *remove assignments to parameter* refactorings because they require accurate alias analysis. We also excluded *consolidate duplicate conditional fragments* because this requires detecting clones at an arbitrary block granularity.

B. Refactoring Identification via Logic Queries

Topological sort. Because the description of a composite refactoring consists of a set of low-level refactorings, we explicitly define partial ordering relationships among refactorings. These ordering relationships are derived from template logic-rule descriptions. For example, based on an *extract superclass* refactoring description, added_subtype(...) $\wedge$ $\neg$ before_subtype(...) $\wedge$ (move_field(...) $\vee$ move_method(...)) $\rightarrow$ extract_superclass(...), the ordering relationship between *move method/field* and *extract superclass* is defined.

We used a topological sort algorithm [32] to identify which refactorings need to be inferred first. Each refactoring type

TABLE III: Refactoring types expressed in terms of template logic rules. This table includes six refactoring types from Fowler's catalogue and the remaining template rule-based description of sixty five refactorings is described in our technical report [28].

Refactoring type	Corresponding template logic rule	Description of each refactoring type
replace conditional with polymorphism	deleted_conditional(condition, thenPart, elsePart, mFullName) ∧ before_method(mFullName, mShortName, tFullName) ∧ after_subtype(tFullName, subtFullName) ∧ added_method(submFullName, mShortName, subtFullName) ∧ similar_body(submFullName, mFullName) → replace_conditional_with_polymorphism(mFullName)	You have a conditional that chooses different behavior depending on the type of an object. Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.
collapse hierarchy	(deleted_subtype(tParentFullName, tChildFullName) ∧ (pull_up_field(fShortName, tChildFullName, tParentFullName) ∨ pull_up_method(mShortName, tChildFullName, tParentFullName))) ∨ (before_subtype(tParentFullName, tChildFullName) ∨ deleted_type(tParentFullName, tParentShortName, package) ∧ (push_down_field(fShortName, tParentFullName, tChildFullName) ∨ push_down_method(mShortName, tParentFullName, tChildFullName))) → collapse_hierarchy(tParentFullName, tChildFullname)	A superclass and subclass are not very different. Merge them together.
pull up method	move_method(fShortName, tChildFullName, tParentFullName) ∧ before_subtype(tParentFullName, tChildFullName) → pull_up_method(fShortName, tChildFullName, tParentFullName)	A method is moved from a class to its superclass.
extract superclass	added_subtype(superClassFullName, classFullName) ∧ NOT(before_type(superClassFullName, X, X)) ∧ (move_field(fieldShortName, classFullName, superClassFullName) ∨ move_method(methodShortName, classFullName, superClassFullName)) →extract_superclass(superClassFullName, classFullName)	You have two classes with similar features. Create a superclass and move the common features to the superclass.
extract method	added_method(toMethodFullName, toMethodShortName, toClassFullName) ∧ after_method(fromMethodFullName, fromMethodShortName, fromClassFullName) ∧ similar_body(toMethodFullName, toMethodBody, fromMethodFullName, fromMethodBody) ∧ after_calls(fromMethodFullName, toMethodFullName) → extract_method(fromMethodFullName, toMethodFullName, toMethodBody, toClassFullName)	You have a code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method.
encapsulate downcast	added_cast(X, tFullName, mFullName) ∧ added_return(mFullName, tFullName) ∧ deleted_return(mFullName, oldtFullName) ∧ (after_subtype(oldtFullName, tFullName) ∨ (after_subtype(oldtFullName, othertFullName) ∧ after_subtype(othertFullName, tFullName))) → encapsulate_downcast(mFullName, tFullName)	A method returns an object that needs to be downcasted by its callers. Move the downcast to within the method.

can be considered as a node in a refactoring hierarchy graph, where an edge from node x to node y exists if a refactoring type x contains a refactoring type y in its antecedent. This sorting result determines the invocation order of logic queries.

Finding concrete refactoring instances. We find concrete refactoring instances by converting the antecedent of each rule into a logic query and invoking the query on the database of logic facts using the Tyruba logic programming engine [10]. For example, to find *pull up method* refactoring instances, the antecedent of the rule, move_method(m1,n1,p1) ∧ before_subtype(p1,c1) is invoked on the factbase to find constant bindings for the logic variables, m1,n1,p1 and c1. Our algorithm then creates a new fact with the consequent predicate, pull_up_method, by substituting its variables with the constant bindings. This new fact is written to the factbase so that other refactorings that require this fact as a pre-requisite can be found later (see Algorithm 1).

C. Example of Refactoring Reconstruction Process.

Consider a *decompose conditional* refactoring, which simplifies a complicated conditional by extracting methods from an `if` statement's condition expression, and *then* and *else* blocks. Table IV shows programs before and after a *decompose conditional* refactoring, its template logic rule, and the snapshots of the factbase during the refactoring identification process. Since the *decompose conditional* refactoring is framed in terms of the *extract method* refactoring, the latter precedes it in the order of identification. Invoking the query results in the three *extract method* instances, which are then written to the factbase file. Querying the antecedent of the *decompose conditional* refactoring returns a non empty result set, implying that the target refactoring did occur.

IV. Evaluation

This section presents evaluation concerning two aspects: How many known refactorings were accurately detected, and how correct were the identified refactorings? Let R represent

Input:
FB, factbase
S, a refactoring hierarchy graph that encodes ordering dependencies among refactorings
R, a hash map of (key:refactoring type, value:rule)
t, a target refactoring type to be inferred

```
while n in topologicalSort (S,t) do
    if n has not been visited yet then
        mark n as visited;
        foreach node m with an edge from n to m do
            inferRefactoring(FB, S, R, m);
        end
        rule:= getValue (R, n) ;
        a := rule.antecedent;
        c := rule.consequent;
        resultSet := executeTyrubaQuery (FB, a) ;
        foreach e in resultSet do
            create a new fact f with predicate c using constant
            bindings e;
            FB := FB ∪ {f};
        end
    end
end
```

Algorithm 1: inferRefactoring

the total number of refactorings identified and E represent the total number of expected refactorings. The precision and recall are defined as follows:

- *Precision* = $|E \bigcap R| \ / \ |R|$
- *Recall* = $|E \bigcap R| \ / \ |E|$

We performed two case studies—one on manually created code examples from Fowler [6] and the other on version pairs of Java open source projects *Carol*, *Columba*, and *jEdit*. For the first type of evaluation, we measured both precision and recall. For the second study, we randomly sampled at most 50 refactorings per version and measured the precision. Since it is hard to find known refactorings, we ran REF-FINDER using similarity threshold σ= 0.65 and manually inspected randomly chosen refactorings until we found 10 correct refactorings. We then measured a recall against this data set at a more reasonable threshold, σ= 0.85.

A. Evaluation using Fowler's Code Examples

Almost all of the related work in refactoring reconstruction cites Fowler's Refactoring book [6]. Code examples from the same book were used to do a preliminary evaluation on 63 refactorings types. The example programs were intended to have only one refactoring in each. After applying REF-FINDER to these programs, we manually inspected the results to confirm whether the expected refactoring instances were found. REF-FINDER found 59 of 63 different types correctly, resulting in 93.7% recall and 97.0% precision. It even found additional refactorings that resulted from combining Fowler's examples in addition to constituent refactorings in addition to constituent refactorings.

Fowler's Refactoring book [6] describes 72 refactorings, which we numbered alphabetically. To display them more easily, we broke up the refactorings into seven groups, as

TABLE IV: Reconstruction of *decompose conditional*

Program differences

```
public class Foo{
  public void main(){
-    // START OF BLOCK 0
-    if (date.before(SUMMER_START)
-        ||date.after(SUMMER_END)//EXPR 1
-        charge = quantity * winterRate
-           + winterServiceCharge;//BLOCK 2
-    else
-        charge = quantity * summerRate;//BLOCK 3
-    //END OF BLOCK 0
+     // START OF BLOCK 4
+     if (notSummer(date))
+        charge = winterCharge(quantity);
+     else
+        charge = summerCharge(quantity);
+     // END OF BLOCK 4
  }
+  boolean notSummer(Date date){
+    return date.before(SUMMER_START)
+    || date.after(SUMMER_END);//BLOCK 5
+  }
+  int winterCharge(int quantity){
+    return quantity * winterRate
+    * winterServiceCharge;//BLOCK 6
+  }
+   int summerCharge (int quantity){
+     return quantity * summerRate;//BLOCK 7
+  }
}
```

decompose conditional **refactoring rule**

deleted_conditional(condition, thenPart, elsePart, origM)
∧ extract_method(origM, newM1, condition, t1)
∧ extract_method(origM, newM2, thenPart, t2)
∧ extract_method(origM, newM3, elsePart, t3)
→ decompose_conditional(condition, thenPart, elsePart, origM)

Initial factbase

added_method("Foo.summerCharge","summerCharge","Foo").
added_method("Foo.notSummer","notSummer","Foo").
added_method("Foo.winterCharge","winterCharge","Foo").
added_methodbody("Foo.main", BLOCK 4).
added_methodbody("Foo.winterCharge", BLOCK 6).
added_methodbody("Foo.notSummer", BLOCK 5).
added_methodbody("Foo.summerCharge", BLOCK 7).
deleted_methodbody("Foo.Foo.main", BLOCK 0).
deleted_conditional(EXPR 1, BLOCK 2, BLOCK 3, "Foo.main").
similar_body("Foo.winterCharge", BLOCK 6,"Foo.main", BLOCK 0).
similar_body("Foo.notSummer", BLOCK 5,"Foo.main", BLOCK 0).
similar_body("Foo.summerCharge", BLOCK 7,"Foo.main", BLOCK 0).

Additional facts written to the factbase, after identifying *extract method* refactorings

extract_method("Foo.main","Foo.summerCharge", BLOCK 7,"Foo").
extract_method("Foo.main","Foo.notSummer", BLOCK 5,"Foo").
extract_method("Foo.main","Foo.winterCharge", BLOCK 6,"Foo").

Additional facts written to the factbase, after identifying *decompose conditional* refactoring

decompose_conditional(EXPR 1, BLOCK 2, BLOCK 3, "Foo.main").

shown in Table V. Each refactoring is represented by its number followed by the quantity of that refactoring we found. For example, 18(4) means type *extract method* was found four times. $|E|$ lists the number of refactorings we expected to find, and $|R|$ the number of found refactorings. When counting the total number of refactorings, we counted both composite and constituent refactorings. For example, the *decompose conditional* refactoring in Table IV is counted as 4 not 1.

The two false positive *extract method*s resulted from short methods with similar but unrelated content, as the content

TABLE V: Results: Fowler's code examples

	Excluded refactorings	Expected refactorings (E)	Identified refactorings (R)	$\|E\|$	$\|R\|$	Prec.	Rec.	False negatives	False positives
1 to 10	8 (*consolidate duplicate conditionals*), 9 (*convert procedural design to objects*)	1-7,10	1-7,10, 25, 18(4), 23, 50, 33, 34, 38, 39	8	19	1.00	1.00		
11 to 20	11 (*duplicate observed data*)	12-20	12-17, 18(2) 19-20, 33(3), 34(3), 1(2), 45(2)	9	20	0.95	1.00		18 (*extract method*)
21 to 30	29 (*introduce foreign method*)	21-28, 30	21-28, 34(3), 33,30	9	12	1.00	1.00		
31 to 40		31-40	31-32, 1(2), 45(2), 33-35, 37-40	10	13	1.00	0.90	36 (*preserve whole object*)	
41 to 50	42 (*remove assignments to parameter*)	41, 43-50	41, 43-48, 50, 28, 18, 23	9	11	1.00	0.89	49 (*replace conditional with polymorphism*)	
51 to 60		51-60	51-58, 18, 45, 60	10	11	1.00	0.90	59 (*replace parameter with explicit methods*	
61 to 72	61 (*replace record with data class*), 68 (*separate domain from presentation*), 70 (*split temporary variable*), 71 (*substitute algorithm*)	62-67, 69, 72	62-64, 66-67, 69, 72, 56, 50, 1, 45, 51, 16, 18	8	14	0.86	0.88	65 (*replace type code with state/strategy*)	56 (*replace magic number with symbolic constants*), 18 (*extract method*)
Total	**9**			**63**	**100**	**0.97**	**0.94**		

similarity is a measure relative to the method size. Another false positive, *replace magic number with constant*, was found on an example where a superclass adds constant fields to replace methods from its subclasses. Most false negatives resulted from not being able to find similarbody facts, which indicates that the similarity threshold σ must be carefully tuned for different types of refactorings.

B. A Case Study using Open Source Projects

We selected *Carol* because the authors knew of existence of complex refactorings in the data set from Kim and Notkin's prior study [27]. The other two subject programs *jEdit* and *Columba* were selected because they were used by many mining software repository projects. The inspected version pairs were randomly selected. The size of input factbase ranged from 12869 to 39353 in *Carol*, from 110151 to 121931 in *jEdit*, and from 374026 to 381893 in *Columba*. The time taken to run the refactoring tool was the lowest for *Carol* revision pair 548-576 (with input factbase size of 31628) at 0.091 minutes and the highest for *Columba* revision pair 352-449 (with input factbase size of 381893) which took 64 minutes for refactoring identification. We deactivated identification of four refactoring types, *replace temp with query*, *inline temp*, *introduce explaining variable*, and *remove control flag* because this requires local variable level analysis and thus slows down REF-FINDER's performance. The evaluation of *Carol* was done on a 2.4GHz Core 2 Duo Windows Vista machine with 2GB RAM. The evaluations of *jEdit* and *Columba* was done on a 3.06GHz Core 2 Duo MacOS machine with 8GB of RAM.

Two of the authors manually inspected refactoring instances reported by REF-FINDER to confirm their correctness. Table VI summarizes the number of refactoring instances found by REF-FINDER, the precision measures, and the types of refactorings found by REF-FINDER. In total, REF-FINDER found 774 refactoring instances. We sampled at most 50 from each version pair and inspected 344 refactorings in total. We determined that 254 of the 344 are correct through manual inspection, reporting 0.738 precision. In the table, the types of refactorings that are found by REF-FINDER but are not handled by existing approaches are in bold font.

TABLE VII: Example of *hide delegate*

Program differences

```
public class TextUtilities{
  public static int findMatchingBracket(Buffer buffer,
      int line, int offset, int startLine, int endLine)
      throws BadLocationException{
    ...
-    TokenMarker tokenMarker = buffer.getTokenMarker();
-    TokenMarker.LineInfo lineInfo = tokenMarker
-        .markTokens(buffer,line);
-    Token lineTokens = lineInfo.firstToken;
+    Buffer.LineInfo lineInfo = buffer.markTokens(line);
+    Token lineTokens = lineInfo.getFirstToken();
    ...
}
```

Most incorrectly identified instances occurred due to argument renamings in method input signatures. For example, in *jEdit* releases 3.0.2-3.1, 17 of the 21 false positives were *add parameter* and *remove parameter* refactorings caused by this.

TABLE VI: Results from jEdit, Columba, and Carol (σ = 0.85)

jEdit						
Versions	$\|R\|$	$\|E\|$	Prec.	Recall	Types of identified refactorings	Time (min)
3.0-3.0.1	10	9	0.75	0.78	remove parameter(1), add parameter(2), **replace magic number with constant**(4), extract method(3)	0.87
3.0.1-3.0.2	1	1	1.00	1.00	remove parameter(1)	0.67
3.0.2-3.1	214	10	0.45	1.00	**change unidirectional to bidirectional association** (3), **replace magic number with constant**(7), **replace parameter with method**(4), **replace nested conditional with guard clauses**(3), **hide delegate**(4), **introduce null object**(4), **change bidirectional to unidirectional association**(1), **separate query from modifier**(1), **consolidate conditional expression**(7), **remove middle man**(4), **replace exception with test**(1), **inline method**(5), remove parameter(89), add parameter(73), extract method(4), move method(4)	12.37
Columba						
300-352	43	10	0.52	0.90	add parameter(5), **change bidirectional to unidirectional association**(1), **remove middle man**(1), push down method(1), move field(2), move method(2), extract method(10), **consolidate conditional expression**(2), **introduce assertion**(3), push down field(1), remove parameter(3), pull up constructor body(1), rename method(7), **replace nested conditional guard clauses**(3),**inline method**(1)	20.86
352-449	209	10	0.91	1.00	**replace nested conditional guard clauses**(2), add parameter(43), remove parameter(79), **inline method**(1), extract method(1), rename method(2), hide method(1), move field(13), move method(11), **consolidate conditional expression**(4), **introduce null object**(2), **replace magic number with constant**(2), **replace constructor with factory method**(1), pull up method(3)	63.60
Carol						
62-63	12	10	1.00	1.00	move method(3), move field(6), add parameter(1), remove parameter(2)	0.15
389-421	8	5	0.63	1.00	**replace exception with test**(1), **replace magic number with constant**(1), add parameter(4), extract method(2)	0.15
421-422	147	10	0.64	0.90	form template method(5), move field(8), **remove assignment to parameters**(12), **replace exception with test**(1), move method(102), add parameter(5), extract superclass(7), extract method(2), remove parameter(5)	0.74
429-430	48	10	0.85	1.00	**introduce local extension**(1), **inline method**(2), **replace exception with test**(4), **replace magic number with constant**(3), move method(20), extract superclass(5), move field(13)	0.12
430-480	37	10	0.81	1.00	**replace magic number with constant**(2), **replace exception with test**(2), **inline method**(2), **consolidate conditional expression**(2), move method(5), move field(5), rename method(1), add parameter(7), extract method(3), remove parameter(8)	1.01
480-481	11	10	0.91	0.90	move field(6), move method(5)	0.61
548-576	20	10	1.00	1.00	move method(15), extract interface(2), move field(3)	0.09
576-764	14	10	0.85	1.00	add parameter(3), **introduce local extension**(1), move method(7), move field(3)	0.69
Total	**774**	**115**	**0.74**	**0.96**		**101**

We now discuss several refactoring instances that are hard to detect using existing methods but were found by REF-FINDER.

(A) *extract superclass* was found previously between *Carol* revisions 429 and 430 when analyzing the LSDiff output [27]. The class `NameSVC` had five subclasses—`CmiRegistry`, `JacORBCosNaming`, `JeremieRegistry`, `IIOPCosNaming` and `LmiRegistry`. An *extract superclass* refactoring occurred in revision 430. Table VIII contrasts the LSdiff results with REF-FINDER results side by side. While LSdiff leaves it to a developer to knit together related changes (Fact 1, Rule 1 to 3), REF-FINDER reports an *extract_superclass* refactoring explicitly by knitting constituent *move* refactorings.

(B) *hide delegate* prevents clients of a class from knowing about the class's delegates. Instead of allowing a client to retrieve a delegate and call methods on it, methods are added to a class to provide access to the delegate's methods. The client no longer accesses the delegate directly, instead making calls through the intermediate class. REF-FINDER found one such instance in *jEdit* versions 3.0.2 to 3.1 (see Table VII). `TextUtilities` called `Buffer.getTokenMarker()`, then called `TokenMarker.markTokens()`. In the new version, this delegation is hidden from `TextUtilities` who merely calls `Buffer.markTokens()`. REF-FINDER produced hide_delegate(-"org.gjt.sp.jedit.syntax.TokenMarker", "org.gjt.sp.jedit.Buffer", "org.gjt.-sp.jedit.TextUtilities").

(C) *inline method* is applied to move the content of a method doing little work to its caller. On *Columba* revisions 352 and 449, the class `MessageController` had a method `chooseBodyPart()` that is called once only from `showMessage()`. The ten lines of code are moved from `chooseBodyPart()` to its caller and the original method is deleted in revision 449.

To determine the effect of threshold σ on the time taken to infer refactorings and the number of found refactorings, we ran REF-FINDER on *jEdit* pair 3.0.2-3.1 and increased σ in 0.05 increments from 0.5 to 0.9. Figure 1 shows that the lower the threshold σ, the longer the running time and the more refactorings found. With a higher threshold σ, REF-FINDER finds fewer refactorings. Relaxing the criteria determining method content similarity may increase the number of false positives and thus could lead to a lower precision. While using a more strict similarity criterion could increase precision, it may decrease recall.

TABLE VIII: Comparison between our prior work (LSdiff[27]) and REF-FINDER results.

LSdiff	REF-FINDER
Fact 1. `AbsRegistry` is a new class. added_type("AbsRegistry","org.objectweb.carol.jndi.ns") Rule 1. The `port` field was deleted in `NameSvc`'s subtype. before_subtype("NameSvc", t) ∧ before_field(f, "port", t) ⇒ deleted_field(f,"port", t) Rule 2. The `setPort` method was deleted from `NameSvc`'s subtype. before_subtype("NameSvc", t) ∧ before_method(m,"setPort(int)", t) ⇒ deleted_method(m, "setPort(int)", t) Rule 3. The `getPort` method was deleted from `NameSvc`'s subtype. before_subtype("NameSvc", t) ∧ before_method(m,"getPort()", t) ⇒ deleted_method(m, "getPort()", t) ...	Refactoring 1. `AbsRegistry` was extracted from `CmiRegistry` extract_superclass("AbsRegistry", "CmiRegistry") Refactoring 2. The port field was moved from `CmiRegistry` class to `AbsRegistry` class. move_field("port", CmiRegistryFullName, AbsRegistryFullName) Refactoring 3. The `getPort` method was moved from `CmiRegistry` class to `AbsRegistry` class. move_method("getPort()", CmiRegistryFullName, AbsRegistry-FullName) Refactoring 4. The `getPort` method in `AbsRegistry` was extracted from the `getPort` method in `CmiRegistry`. extract_method("CmiRegistry.getPort()","-AbsRegistry.getPort()","port","AbsRegistry")...

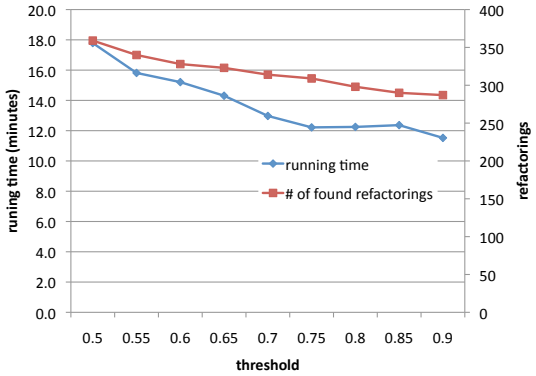

Fig. 1: REF-FINDER Performance while varying threshold (σ)

C. Limitations

Since REF-FINDER queries the facts extracted using LSDiff, the performance of LSDiff's fact extraction directly affects REF-FINDER's capability in identifying different types of refactorings. Complex refactorings themselves consist of a set of atomic refactorings and thus an incorrect identification of atomic refactorings will lead to an incorrect or missed identification of complex refactorings. Our interpretation of the definitions for the different types of refactorings mentioned in Fowler might be subject to bias.

The evaluation on Fowler's code examples [6] suffers from an additional validity concern since the code examples and the rules used to encode refactorings are from the same source. The version histories of the open source projects may not be representative for other kinds of software projects.

D. Future Work

We plan to investigate the robustness of our refactoring reconstruction tool when refactorings overlap with non-refactorings, (i.e., floss refactorings, coined by Murphy-Hill et al. [33]). In addition, to better measure REF-FINDER's recall, we plan to seed refactorings using Eclipse's refactoring features and compare a set of refactorings found by REF-FINDER with a set of seeded refactorings. We also plan to compare reconstructed refactorings with a set of recorded refactorings in IDE, such as Spyware's refactoring logs [23].

V. CONCLUSION

Existing refactoring reconstruction approaches handle simple refactoring types such as *renames*, *moves*, and basic *extracts* but steer away from complex refactorings which themselves consist of atomic refactorings and require analysis of the internal method body content. In this paper, we used a logic meta-programming approach to identify complex refactorings from two program versions. Different types of refactorings were expressed as template logic rules and a logic programming engine was used to infer concrete refactoring instances. Our evaluation shows that REF-FINDER's overall precision is 0.79.

ACKNOWLEDGMENT

This research is in part supported by IBM Jazz Innovation Award. The authors thank anonymous reviewers for their thorough comments.

REFERENCES

[1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.
[2] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP*, 2006, pp. 404–428.
[3] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," in *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM, 2005, pp. 274–283.
[4] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818–836, 2007.
[5] Xing and Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *ASE '05*. New York, NY, USA: ACM, 2005, pp. 54–65.
[6] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
[7] R. C. Holt, "Structural manipulations of software architecture using tarski relational algebra," in *WCRE '98: Proceedings of the Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1998, p. 210.
[8] K. Mens, T. Mens, and M. Wermelinger, "Maintaining software through intentional source-code views," in *SEKE '02*. ACM, 2002, pp. 289–296.
[9] E. Hajiyev, M. Verbaere, and O. de Moor, "Codequest: Scalable source code queries with datalog," in *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, vol. 4067. Berlin, Germany: Springer, 2006, pp. 2–27.
[10] K. D. Volder, "Type Oriented Logic Meta Programming," Ph.D. dissertation, The University of British Columbia, 1998.
[11] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," *SIGPLAN Not.*, vol. 35, no. 10, pp. 166–177, 2000.

[12] G. Malpohl, J. J. Hunt, and W. F. Tichy, "Renaming detection," *Automated Software Engg.*, vol. 10, no. 2, pp. 183–202, 2003.

[13] F. Van Rysselberghe and S. Demeyer, "Reconstruction of successful software evolution using clone detection," in *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003, p. 126.

[14] G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 31–40.

[15] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When functions change their names: Automatic detection of origin relationships," in *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 143–152.

[16] L. Zou and M. Godfrey, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, 2005.

[17] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 231–240.

[18] Z. Xing and E. Stroulia, "Refactoring detection based on umldiff change-facts queries," in *WCRE '06*. Washington, DC, USA: IEEE, 2006, pp. 263–274.

[19] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[20] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 481–490.

[21] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 333–343.

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

[23] R. Robbes and M. Lanza, "Spyware: a change-aware development toolset," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 847–850.

[24] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 391–400.

[25] Y.-G. Guéhéneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 667–684, 2008.

[26] T. Tourwé and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 91.

[27] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319.

[28] K. Prete, N. Rachatasumrit, and M. Kim, "Catalogue of template refactoring rules," The University of Texas at Austin, Tech. Rep. UTAUSTIN-ECE-TR-041610, April 2010.

[29] A. Loh and M. Kim, "A program differencing tool to identify systematic structural differences," in *ICSE '10 Research Demo*, 2010, p. 4.

[30] *Eclipse, http://www.eclipse.org*.

[31] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2001.

[33] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 287–297.