ASM Sc. J., 13, Special Issue 1, 2020 for ICFICE, 92-103

# A Composability Test Method for the Composite Refactoring Assembled by Component Refactorings

Kyungmin Kim[1], Taegong Kim[1], Jaihyun Seu[1], Jaewon Oh[2] and Taewoong Kim[3*]

[1]*School of Computer Engineering, Inje University, Gimhae, Korea*

[2]*School of Computer Science and Information Engineering,*
*The Catholic University of Korea, Bucheon, Korea*

[3]*Department of Computer Education, Silla University, Busan, Korea*

Recently, much research on defining composite refactorings by composing component refactorings has been conducted. The main problem of composite refactorings is that we can't say in advance whether the composite refactoring is applicable to a program or not. The methods through which previous researches solve the problem are classified into two approaches. The first one uses an 'undo' operation, and the second one uses the joint precondition of the composite refactoring. But there are problems, in that the first one wastes resources spent on executing and undoing the composite refactoring, and the second one is useless if the joint precondition can't be computed because of its derivation complexity. In this paper, we try to decide the composability for the composite refactoring, even though the joint precondition is not explicitly specified. To do this, first, we propose a method to specify elementary refactorings. Second, we propose a method to decide the composability for the composite refactoring defined by the refactoring composition language. Based on these methods, we develop a prototype tool that can decide composability. Finally, we verify the effectiveness of this research through case studies, in which we define composite refactorings and decide the composability.

**Keywords:** composit refactoring; component refactoring; refactoring composition language; composability; JavaEAST meta model; OCL

## I. INTRODUCTION

Refactoring is actively used in the latest software development. Refactoring is the transformation that modifies program structure but keeps program behavior (Fowler, 2018). As the result of refactoring, we can improve the quality of code, such as extendibility, modularization, reusability, and maintainability. This improvement causes increase in the speed of development and decrease in code complexity (Mens & Tourwé, 2004; Eckel, 2000). Today, to verify the effectiveness of the study, the application software is actually implemented in various studies such as researches (Suryadibrata & Kim, 2017; Kim & Song, 2017; Cao *et. al.*, 2017). The use of refactoring in such software development will improve the quality of implementation.

There are popular literatures about refactorings, such as Fowler (Fowler, 2018) and Kerievsky (Kerievsky, 2004). These literatures describe a way of applying refactorings through stepwise procedures. If we look into these literatures carefully, we can find the cases that some refactorings are repeatedly used in other refactorings. It will be very efficient, if we reuse these component refactorings. Therefore, to increase the reusability of refactoring, many studies are trying to define composite refactoring by assembling component refactorings.

The main problem of composite refactorings is that we can't say, in advance before applying a composite refactoring, whether the composite refactoring is applicable to a program or not. If the composite refactoring is applied even though one of the components

*Corresponding author's e-mail: twkim@silla.ac.kr

refactorings in the composite refactoring fails, the behavior preservation of the program will not be guaranteed. There are many previous researches for addressing the problem, such as Opdyke (Opdyke, 1992) which introduces the composite refactoring concept, Roberts (Roberts, 1999), Cinnéide (Cinnéide, 2001), Kniesel (Kniesel & Koch, 2004), Huang (Huang *et. al.*, 2011), Saadeh (Saadeh & Kourie, 2009), and Li (Li & Thompson, 2012). The methods through which the previous researches solve the problem are classified into two approaches. The first one uses an 'undo' operation. If certain component refactoring fails during applying a composite refactoring, we have to roll back the entire composite refactoring through undoing the effects of component refactorings applied previously. It is a waste of resources spent on executing and undoing the composite refactoring.

The second one uses the joint precondition of composite refactoring. This approach must complete the derivation of the joint precondition of composite refactoring in advance. Since the entire composite refactoring is not applied if the joint precondition of the composite refactoring fails, the rollback problem would not arise. But the derivation of the joint precondition is very complex and difficult, since we must take into account the effects of applied component refactoring. Moreover, if the joint precondition can't be computed because of its derivation complexity, the first approach instead of the second one should be used. In this paper, the authors try to decide the composability for the composite refactoring, even though the joint precondition is not explicitly specified.

The rest of the paper is organized as follows: We introduce a refactoring composition language that can be used to define composite refactorings in Section 2. Section 3 proposes a method to specify elementary refactorings, and Section 4 proposes a method to decide the composability for the composite refactorings defined by the refactoring composition language. Based on these methods, we develop a prototype tool that can decide composability in Section 5, and Section 6 verifies the effectiveness of this research through case studies, in which we define composite refactorings by using the refactoring composition language, and decide the composability of the composite refactorings. Section 7 provides a comparison of the related works with our research. Finally, we conclude in Section 8.

## II. AN OVERVIEW OF REFACTORING COMPOSITION LANGUAGE

This section briefly introduces a refactoring composition language (RCL) for defining composite refactoring's. Because details of RCL have already been described in our earlier work, Kim (Kim & Kim., 2012), only relevant things are presented here.

### *A. Types of Component Refactoring*

Composite refactorings are assembled with component refactorings. There should be various types of component refactoring to define many composite refactorings. In Kim (Kim & Kim., 2012), they classify the types of component refactoring as elementary refactoring, creation refactoring, and defined refactoring.

For example, as shown in Figure 1, defined refactoring DR1 is assembled with elementary refactorings ER1 and ER2, and defined refactoring DR2 is assembled by elementary refactorings ER1 and ER3. The bigger defined refactoring DR3 is defined by composing the component refactorings, such as the already defined refactorings DR1, DR2 and the creation refactoring CR1. We could improve the extendibility and reusability of refactoring, if we define composite refactorings by assembling component refactorings, like this example.

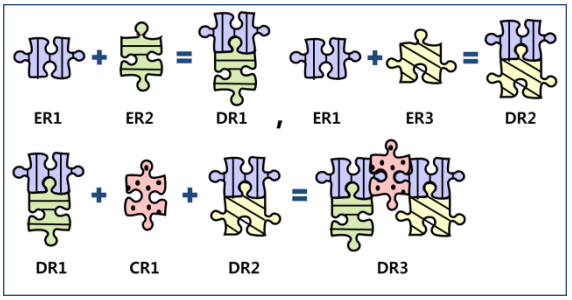



Figure 1. Examples of composite refactoring

### *B. A Meta Model of RCL*

An abstract syntax of RCL can be defined as a meta model, based on EMF (Budinsky *et. al.*, 2004). Figure 2 shows a meta model of RCL. The meta model is made up of the

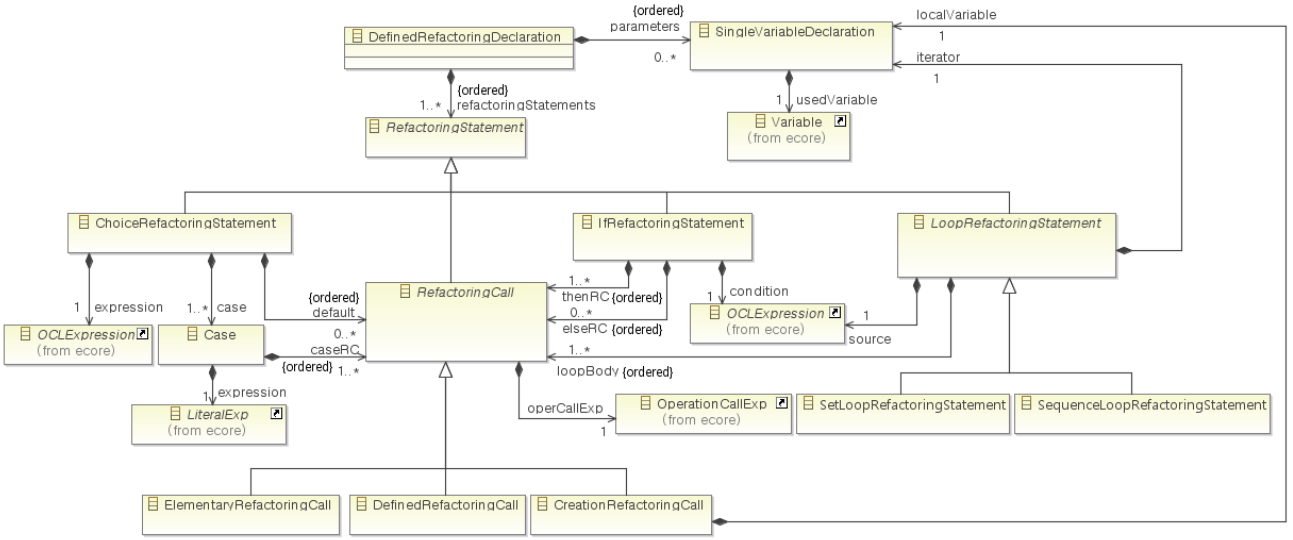



Figure 2. A meta model of RCL

name of composite refactoring, the types and names of parameters of composite refactoring, a way to compose refactoring, and the types of refactoring call.

The sequential composition method applies component refactorings sequentially. If at least one of the preconditions is not satisfied during the application of the composite refactoring, the entire composite refactoring could not be applied. This composition method is a basic one that is used in most existing researches. A meta model element that is related to sequential composition is DefinedRefactoringDeclaration. DefinedRefactoringDeclaration represents a composite refactoring to define, and refactoringStatements, a property of DefinedRefactoringDeclaration, represents component refactorings that will be composed sequentially.

Meta model elements related to selectional composition are IfRefactoringStatement, and ChoiceRefactoringStatement. The IfRefactoringStatement represents a conditional composition method. If the result of the given condition is true, it should be able to apply the thenRC part of the composite refactoring. Otherwise, the elseRC part should be able to be applied. The ChoiceRefactoringStatement represents a multiple choice composition method. The component refactorings in the case part that has the same value as the given expression, should be able to be applied. If there is no case part that has the same value as the given expression, the component refactorings in the default part should be able to be applied.

A meta model element related to iterative composition is the Loop Refactoring Statement. It should be able to apply the component refactorings of the loopbody part repeatedly, on each element included in the source. The LoopRefactoringStatement has two subclasses. One is the SequenceLoopRefactoringStatement for ordered iterative composition, and the other is the SetLoop_ RefactoringStatement for unordered iterative composition. If there is an order on elements in the source, it is necessary to use the SequenceLoopRefactoringStatement.

## III. A FORMAL SPECIFICATION OF ELEMENTARY REFACTORINGS

### *A. Program Representation*

In our research, we use the JavaEAST meta model (Kim *et. al.*, 2011) to represent Java source code. The JavaEAST meta model is one that extends the JavaAST (Java Abstract Syntax Tree) meta model proposed by MoDisco (Eclipse Modisco, 2019), including binding information. If we use this JavaEAST meta model, we can search and analyze source code efficiently.

Figure 3 (b) is a part of the XMI (W3C, 2019) document that represents a certain Java program of Figure 3 (a) based on JavaEAST meta model.

```
public class A {
    private int x;
    public int getX() {
        return x;
    }
}
```

**(a) Java sample program**

```
<?xml version="1.0" encoding="ASCII"?>
<Project xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="JavaEAbstractSyntaxtk3"
 xsi:schemaLocation="JavaEAbstractSyntaxtk3 JavaEAbstractSyntaxtk3.ecore">
 <compilationUnits container="/">
  <types xsi:type="TypeDeclaration" container="//@compilationUnits.0
   referenceNames="//@compilationUnits.1/@types.0/@superclassType/@name">
  <modifiers xsi:type="Modifier" container="//@compilationUnits.0/@types.0"
   public="true"/>
  <bodyDeclarations xsi:type="FieldDeclaration"
   container="//@compilationUnits.0/@types.0">
  <modifiers xsi:type="Modifier"
   container="//@compilationUnits.0/@types.0/@bodyDeclarations.0" private="true"/>
  <fragments container="//@compilationUnits.0/@types.0/@bodyDeclarations.0"
   referenceNames="//@compilationUnits.0/@types.0/@bodyDeclarations.1/@body/
   @statements.0/@expression//@compilationUnits.0/@types.0/@bodyDeclarations.2/@body/
   @statements.0/@expression/@leftHandSide/@name">
  <name container="//@compilationUnits.0/@types.0/@bodyDeclarations.0/@fragments.0"
   fullyQualifiedName="x" identifier="x" eclaration="true"/> </fragments>
  <type xsi:type="PrimitiveType" container="//@compilationUnits.0/@types.0/
   @bodyDeclarations.0" code="int"/> </bodyDeclarations>
  <bodyDeclarations xsi:type="MethodDeclaration"
   container="//@compilationUnits.0/@types.0">
  <modifiers xsi:type="Modifier"
   container="//@compilationUnits.0/@types.0/@bodyDeclarations.1" public="true"/>
  <body container="//@compilationUnits.0/@types.0/@bodyDeclarations.1">
   <statements xsi:type="ReturnStatement"
    container="//@compilationUnits.0/@types.0/@bodyDeclarations.1/@body">
   <expression xsi:type="SimpleName" container="//@compilationUnits.0/@types.0/
    @bodyDeclarations.1/@body/@statements.0" fullyQualifiedName="x" identifier="x"
    definition="//@compilationUnits.0/@types.0/@bodyDeclarations.0/@fragments.0"/>
   </statements> </body>
  <name container="//@compilationUnits.0/@types.0/@bodyDeclarations.1"
   fullyQualifiedName="getX" identifier="getX" declaration="true"/>
  <returnType xsi:type="PrimitiveType" container="//@compilationUnits.0/@types.0/
   @bodyDeclarations.1" code="int"/> </bodyDeclarations>
 <name container="//@compilationUnits.0/@types.0" fullyQualifiedName="A"
  identifier="A" declaration="true"/> </types> </compilationUnits> </Project>
```

**(b) XMI document**

Figure 3. A part of the XMI document that represents a Java program based on JavaEAST meta model

### *B. A Method for Elementary Refactoring Specification*

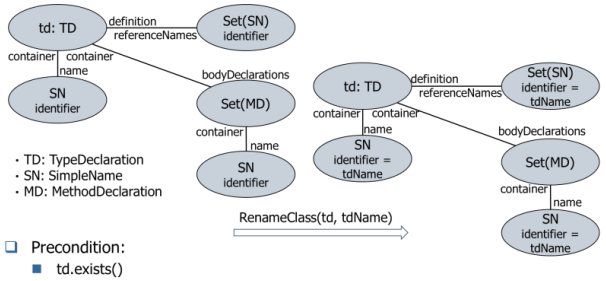
An elementary refactoring has both a precondition and a delta specification which can be specified by using OCL (OMG, 2014; Warmer & Kleppe, 2003; Ol'Khovich & Koznov, 2003; Ziemann & Gogolla, 2003; Benattou *et. al.*, 2002) and the JavaEAST meta model. A precondition is a condition that should be satisfied before applying a refactoring. Delta specification represents how the interpretation of a model property should be changed after applying a refactoring. If we use this delta specification, we can have the same effect as we obtain when we apply a refactoring to a program directly. A simple precondition can be defined by using model properties which are defined in a meta model. But for a complex precondition, we should use user-defined features which are defined using model properties and another existing user-defined features.

In this paper, we describe refactoring effects by using '@post' keyword, on the basis of the state before the application of refactorings. Refactoring effects described in this way can be easily transformed into delta specification.

We can represent the effect of a refactoring accurately, if we use delta specification. Basically, delta specification is specified by using only model properties of the meta model. Therefore, it has no impact on existing delta specifications of other refactorings, even if a new user-defined feature is added.

### *C. Examples for Elementary Refactoring Specification*

For example, we specify RenameClass elementary refactoring with the method for elementary refactoring specification previously proposed. RenameClass is the elementary refactoring that alters class name, and it has two parameters.




- Precondition:
  - td.exists()
    and not( TypeDeclaration.allInstances()->exists(t | t.name.identifier = tdName) )
    and not ( td.methods()→exists(m | m.name.identifier = tdName) )
- Refactoring effects:
  - td.name.identifier@post = tdName
  - ∀ e ∈ td.referenceNames , e.identifier@post = tdName
  - ∀ e ∈ td.bodyDeclarations->select(bd | bd.oclIsTypeOf(MethodDeclaration))
    ->collect(bd | bd.oclAsType(MethodDeclaration))
    ->select(md | md.constructor = true)->collect(md | md.name),
    e.identifier@post = tdName
- Delta specification:
  - context SimpleName def : identifier@post : String =
    let mdnames : Set(SimpleName) = td.bodyDeclarations
    ->select(bd | bd.oclIsTypeOf(MethodDeclaration))
    ->collect(bd | bd.oclAsType(MethodDeclaration))
    ->select(md | md.constructor = true)->collect(md | md.name)->asSet()
    in if (self = td.name) or ( td.referenceNames->includes(self) ) or
    (mdnames->includes(self))
    then tdname
    else self.identifier endif

Figure 4. A precondition and delta specification of RenameClass(TypeDeclaration td, String tdName)

Figure 4 is the specification of RenameClass elementary refactoring. The precondition consists of three conditions. First, the TypeDeclaration 'td', whose name is trying to be changed, should exist. Second, the TypeDeclaration, whose name is 'tdName', should not exist. Third, the method whose name is 'tdName' should not exist in TypeDeclaration 'td'. The delta specification of Figure 4 means that the name of 'td', the names that are referencing 'td', and the names of constructors of 'td', should all be changed to the new name 'tdName'.

## IV. A COMPOSABILITY TEST FOR COMPOSITE REFACTORING

### *A. Program Abstraction*

'undo' operations should be used if a program is modified directly when a composite refactoring is applied, since it is necessary to roll back already applied component refactorings if it fails to compose a certain component refactoring while applying the composite refactoring. As mentioned in Section 1, it is a waste of resources spent on executing and undoing the composite refactoring. A composability test of this paper solves the rollback problem through changing the abstraction of the program instead of modifying the program directly.

As shown Figure 5, we abstract programs with model properties and user-defined features. So an abstraction of a program will become different, if the interpretation of the model property changes. And the interpretation of the model property will be changed after refactorings are applied. We can see the changed interpretation of the model property through a delta specification of an elementary refactoring, since the delta specification represents how the interpretation of a model property should be changed after applying the elementary refactoring as mentioned earlier.

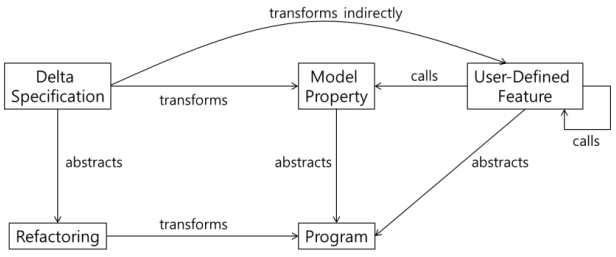



Figure 5. Associations among concepts related to program abstraction and refactoring

The interpretation of a user-defined feature is defined by using the current interpretations of model properties. Therefore, the interpretation of a user-defined feature should become different according to the interpretations of model properties. There is no description about user-defined features in the delta specification of an elementary refactoring. However, we can infer which user-defined features' interpretation should be changed by using a feature dependency graph, when the interpretation of a model property is changed. The feature dependency graph represents dependency relationships between model properties and user-defined features. Based on OCL metamodel (OMG, 2014), we construct a feature dependency graph by analyzing feature call relationships between model properties and user-defined features. If the definition of a feature is modified, the caller features which depend on the feature should be redefined using the modified feature instead of the original feature. In other words, if the interpretation of a callee feature is changed, the interpretations of caller features should be changed accordingly. Figure 6 shows a part of feature dependency graph.

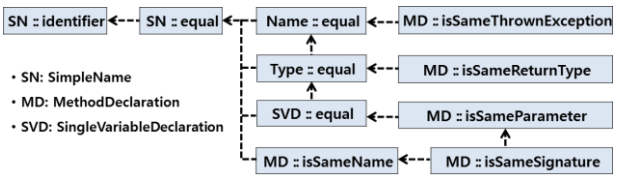



Figure 6. A part of feature dependency graph

In this way, the abstraction of a program is changed automatically, by using delta specifications and a feature dependency graph, during performing the composability test for a composite refactoring.

For example, let's see how the abstraction of a program is changed, when RenameClass(x, "Circle") elementary refactoring is applied. If RenameClass(x, "Circle") is applied, the interpretation of the model property 'identifier' in SimpleName context will be changed by using delta specifications of the elementary refactoring in Figure 4 of Section 3.3. Figure 7 (a) shows the changed interpretation of the model property. Such a change of the model property interpretation entails the changes of interpretations of the user-defined features that depend on

the model property. As shown in Figure 6, the user-defined feature 'equal' in SimpleName context directly depends on the 'identifier', and the user-defined feature 'isSameName' in MethodDeclaration context indirectly depends on it. Therefore, the change of the 'identifier' interpretation entails the changes of the interpretations of 'equal' and 'isSameName'. Figure 7 (b) shows the changed interpretations of the user-defined features. Of course, the changes of interpretations of another user-defined features are propagated along the dependency relationships in a feature dependency graph.

```
context SimpleName
def : identifier@post : String =
if (self = x.name) or ( x.referenceNames->includes(self) )
        or (x.getConstructorMethodSNs->includes(self))
then 'Circle' else self.identifier endif
```
**(a)**

```
context SimpleName
def : equal@post (sn : SimpleName) : Boolean =
if self.definition->notEmpty() and sn.definition->notEmpty()
then self.identifier@post = sn.identifier@post else false endif

context MethodDeclaration
def : isSameName@post (method : MethodDeclaration) : Boolean =
self.name.equal@post (method.name)
```
**(b)**

Figure 7. A part of redefined features after applying RenameClass(x, "Circle") elementary refactoring

To decide whether we could apply another RenameClass(y, "Rectangle") elementary refactoring after RenameClass(x, "Circle") is applied, we should reinterpret the precondition of RenameClass(y, "Rectangle") to fit in the changed abstraction of the program. Figure 8 shows the reinterpreted precondition of RenameClass(y, "Rectangle") which is originated from the precondition of RenameClass specification in Figure 4. In this figure, we can notice that the model property 'identifier' of SimpleName context is reinterpreted as new user-defined feature 'identifier@post'.

```
y.exists()
and not(TypeDeclaration.allInstances()
      ->exists(t|t.name.identifier@post ='Rectangle'))
and not(y.methods()->exists(m|m.name.identifier@post = 'Rectangle'))
```

Figure 8. The reinterpreted precondition of RenameClass(y, "Rectangle")

### *B. A Composability Test Method*

In this section, we propose a composability test method for a composite refactoring, to decide whether the composite refactoring is applicable, according to the composition method. To do this, we serialize programs to XMI documents based on the JavaEAST meta model. And we also serialize refactorings to XMI documents based on the RCL meta model.

To decide the composability of a composite refactoring, all the nodes should be visited, starting at the root node of the XMI document corresponding to the composite refactoring. A node should be handled according to the proper type of the node, when each node is visited. For example, if the ElementaryRefactoringCall node is visited, the precondition corresponding to the node should be checked after reinterpreting the precondition to fit in the current abstraction of the program. If that elementary refactoring is applied in the case that the precondition is false, the behavior preservation of a program could not be guaranteed. Therefore, we can decide that the composite refactoring, including that elementary refactoring, could not be applied. Like this, if at least one of the outcomes of visiting nodes is false, we decide that the composite refactoring, on which the composability test is being performed, could not be applied. And if all the outcomes of visiting nodes are true, we decide that the composite refactoring could be applied.

For brevity, we consider only the DefinedRefactoringCall and ElementaryRefactoringCall nodes, among the nodes that are visited when a composability test is performed. Let's look at the detailed procedures to be handled in a composability test.

Figure 9 shows the procedures to be handled, when visiting the DefinedRefactoringCall node drc. It proceeds as the following steps.

1) Load DefinedRefactoringDeclaration node drd, which corresponds to DefinedRefactoringCall node drc, among DRD XMI documents based on the RCL meta model.
2) Bind the parameters of drd to the arguments of drc.
3) Load the first RefactoringStatement node of drd.
4) Decide the type of the RefactoringStatement node.
5) Visit the node so as to handle properly according to the node type.
6) Return false as a final result if the result of visiting the node is false at step 5) Otherwise, it decides

whether there is next RefactoringStatement node.

7) If there is no next RefactoringStatement node, it returns true as a final result. Otherwise, it repeats again starting at step 4) after loading next RefactoringStatement node.

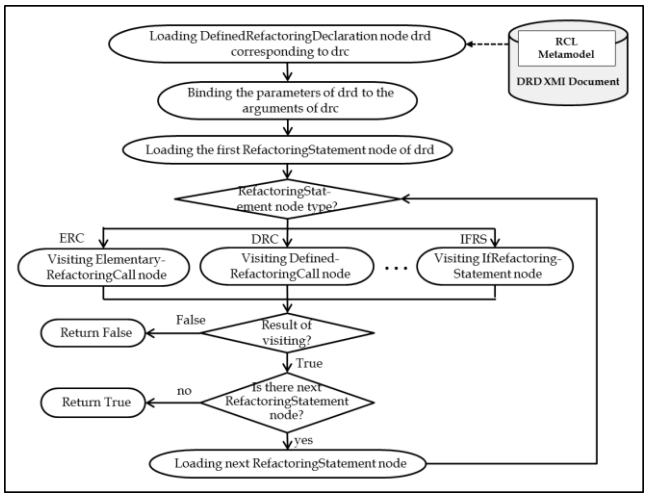



Figure 9. The handling process while visiting DefinedRefactoringCall node drc

If false is returned as the final result of visiting a DefinedRefactoringCall node, it means that at least one of the preconditions of component refactorings is not satisfied during the application of the composite refactoring. Therefore, behavior preservation could not be guaranteed, if the composite refactoring corresponding to the DefinedRefactoringCall node is applied. On the other hand, if the composite refactoring is applied when true is returned, behavior preservation could be guaranteed.

Figure 10 shows the procedures to be handled, when visiting the ElementaryRefactoringCall node erc. It loads a precondition corresponding to ElementaryRefactoringCall node erc, among Precondition XMI documents based on the 'Precondition Metamodel'. It modifies the arguments of erc to fit into the current abstraction of the program, and binds parameters used in the precondition to the arguments of erc. And it evaluates the precondition, after reinterpreting the precondition to fit into the current abstraction of the program. If the result of the evaluation is false, it returns false as a result. Otherwise, true is returned as a final result, after changing the abstraction of the program through the delta specification corresponding to erc, and a feature dependency graph.

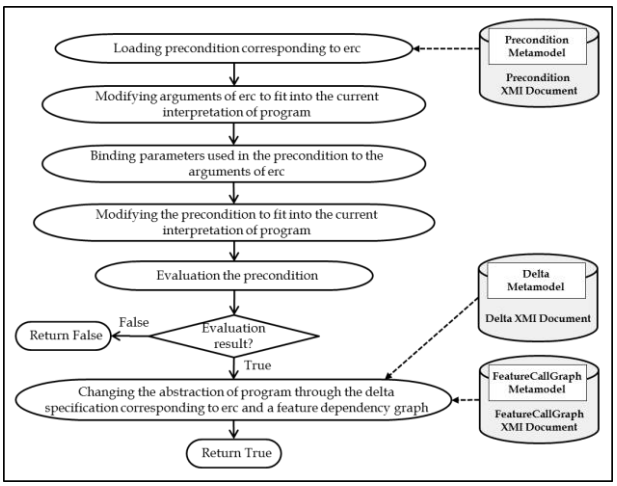



Figure 10. The handling process while visiting ElementaryRefactoringCall node erc

## V. AN IMPLEMENTATION OF THE TOOL

In this section, we describe a tool that can define a composite refactoring by using RCL, and check composability of the composite refactoring. We develop a prototype tool by using the Eclipse modeling framework (Budinsky *et. al.*, 2004; Eclipse EMF, 2019) and MDT OCL Interpreter (Eclipse OCL, 2019). And as quoted in Study (Youn *et. al.*, 2017), we have placed organized buttons so that users can navigate the tool more easily so that they have a clear and consistent conceptual structure of the layout.

Figure 11 is a screen capture that defines a composite refactoring by using the tool. If you enter the name and parameter information of the composite refactoring to define, and select the desired RefactoringStatement type, then a dialog box pops up, which helps you draw up the selected statement type to fit in the syntax. And once all the information of the corresponding composite refactoring is drawn up and stored, an XMI document, which is based on the meta model of RCL.

Figure 12 is a screen capture that performs a composability test for the composite refactoring defined by using the tool. PullUpVDF composite refactoring needs an argument of the VariableDeclarationFragment type. For example, we assign the target field in TreeViewer (rectangle part) of Figure 12 as the value of the Value column of the table by drag-and-drop. And then the

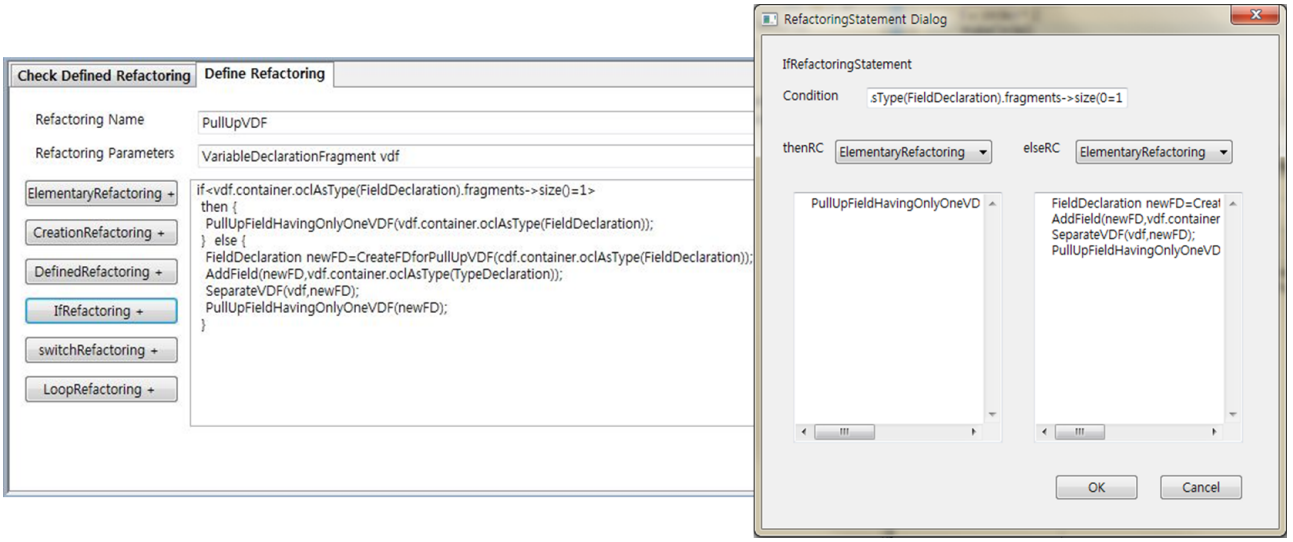



Figure 11. A definition of PullUpVDF composite refactoring by tool

composability test will start, if the 'check' button is clicked.

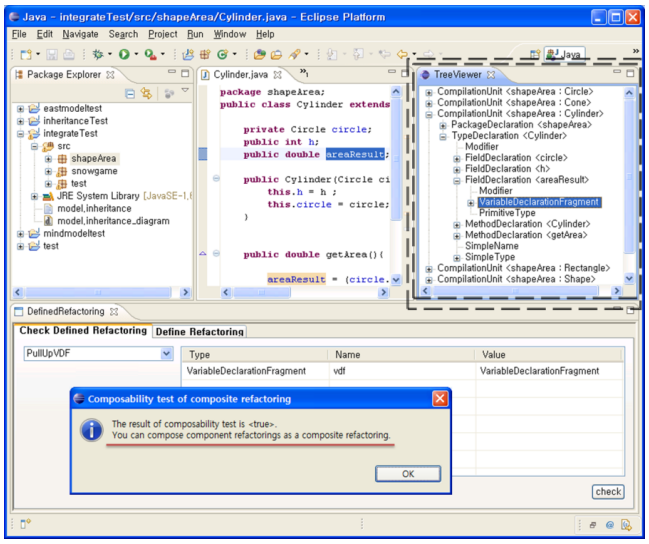



Figure 12. Composability test of PullUpVDF composite refactoring by the tool

## VI. CASE STUDY

In this section, we define IPCwFM composite refactoring, and then we decide whether or not the composite refactoring is applicable to an example Java program.

### *A. Definition*

IPCwFM that is introduced as 'Introduce Polymorphic Creation with Factory Method' in Kerievsky (Kerievsky, 2004), is the refactoring that targets the 'Factory Method' design pattern. This refactoring can be applicable, when sibling subclasses implement a method similarly, except for an object creation step. For example, IPCwFM could be applied to the Java code of Figure 13 (a). This code introduced in Kerievsky (Kerievsky, 2004) is a part of 'XML Builder', which makes it easy to create an XML document.

| | |
|---|---|
| public class DOMBuilderTest extends TestCase{<br>private OutputBuilder builder;<br>public void testAddAboveRoot(){<br>String invalidResult = "<orders><br><order>" + "</order></orders>"<br>+ "<customer></customer>";<br>builder=new DOMBuilder("orders")<br>builder.addBelow("order");<br>try{<br>…builder.addAbove("customer");<br>…<br>} catch (RuntimeException<br>ignored) { } } } | public abstract class AbstractBuilder_<br>Test extends TestCase{<br>protected OutputBuilder builder;<br>protected abstract OutputBuilder<br>createBuilder(String rootName);<br>public void testAddAboveRoot(){<br>String invalidResult = "<orders><br><order>" + "</order> </orders>"<br>+ "<customer></customer>";<br>builder = createBuilder("orders");<br>try{<br>…builder.addAbove("customer");<br>…<br>} catch (RuntimeException<br>ignored) { } } } |
| public class XMLBuilderTest extends<br>TestCase{<br>private OutputBuilder builder;<br>public void testAddAboveRoot(){<br>String invalidResult = "<orders><br><order>" + "</order></orders>"<br>+ "<customer></customer>";<br>builder=new XMLBuilder("orders")<br>builder.addBelow("order");<br>try{<br>…builder.addAbove("customer");<br>…<br>} catch (RuntimeException<br>ignored) { } } } | public class DOMBuilderTest extends<br>AbstractBuilderTest{<br>private OutputBuilder<br>createBuilder(String rootName){<br>return new DOMBuilder<br>(rootName); } }<br><br>public class XMLBuilderTest extends<br>AbstractBuilderTest{<br>private OutputBuilder<br>createBuilder(String rootName){<br>return new XMLBuilder<br>(rootName); } } |
| **(a) original Java code** | **(b) refactored Java code** |

Figure 13. Application of IPCwFM composite refactoring

Figure 14 (a) shows the steps of IPCwFM refactoring. After making an abstract class, it sets up an inheritance hierarchy between the abstract class and the classes including an object creation step. It makes methods that create the same object as the object creation step on each

element of sequence 'cics', and adds these methods as program elements. It changes each object creation to the object creation method call. And then it pulls up the methods to superclass, if the bodies of the methods become the same. If we represent the procedure of IPCwFM by using a concrete syntax of RCL, it is like Figure 14 (b). We can define the procedure as a defined refactoring, by using the prototype tool.

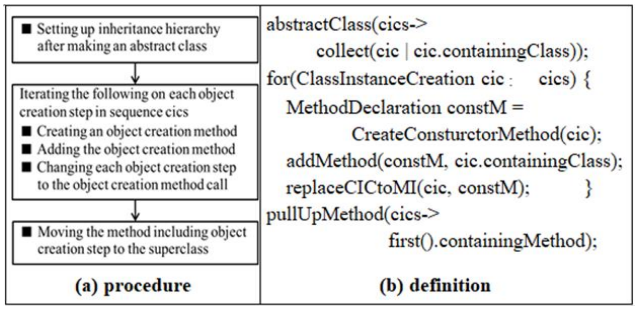



Figure 14. A definition of IPCwFM(Sequence(ClassInstanceCreation) cics)

IPCwFM can be defined by composing a composite refactoring, which is assembled by one iterative composition, and another two composite refactorings, sequentially. Table 1 explains about the component refactorings that IPCwFM consists of.

Table 1. Component refactorings that IPCwFM consists of

| Component refactoring | Explanation |
|---|---|
| createSetterMethod (VDF vdf) | creation refactoring to create a new MD for the setter method that assigns the value of vdf |
| createGetterMethod (VDF vdf) | creation refactoring to create a new MD for the getter method that returns the value of vdf |
| addMethod (MD md, TD td) | elementary refactoring that adds md to td |
| replaceFieldRead (SimpleName ref, MD getterM) | composite refactoring that changes ref, which is a read access of a field, into invocation to the getterM method that returns the value of the field |
| replaceFieldWrite (SimpleName ref, MD setterM) | composite refactoring that changes ref, which is a write access of a field, into invocation to the setterM method that assigns the value of the field |
| replaceModifierTo Private(VDF vdf) | elementary refactoring that changes the access modifier of vdf into private |

MD: MethodDeclaration, VDF: VariableDeclarationFragment, TD: TypeDeclaration

### *B. Composability Test*

Let's decide whether IPCwFM is applicable to the Java code of Figure 13 (a). IPCwFM needs a sequence of ClassInstanceCreation type as an argument. Therefore if the composability test starts after selecting 'new DOMBuilder("orders")' and 'new XMLBuilder("order")' object creation steps as the argument values, true is returned as the result. This means that behavior preservation could be guaranteed, if we apply the IPCwFM to the example program.

Table 2. Chain of RefactoringCalls occurred during the composability test for IPCwFM

| # | Name | Type | Time(sec) |
|---|---|---|---|
| 1 | abstractClass | DRC | 0.1958 |
| 1.1 | createAbstractClass | CRC | 0.0009 |
| 1.2 | addClass | ERC | 0.0217 |
| 1.3 | addExtendsLinkHasSuperclass | DRC | 0.1598 |
| 1.3.1 | createSuperclassType | CRC | 0.0007 |
| 1.3.2 | addExtends | ERC | 0.0213 |
| 1.3.3 | replaceExtends | ERC | 0.0498 |
| 1.3.4 | replaceExtends | ERC | 0.0491 |
| 2 | createConstructorMethod | CRC | 0.0008 |
| 3 | addMethod | ERC | 0.0246 |
| 4 | replaceCICtoMI | DRC | 0.1376 |
| 4.1 | replaceCICinRHSofASMTtoMI | DRC | 0.1103 |
| 4.1.1 | createStaticMI | CRC | 0.0008 |
| 4.1.2 | replaceCICinRHSofASMTtoMI | ERC | 0.0622 |
| 5 | createConstructorMethod | CRC | 0.0007 |
| 6 | addMethod | ERC | 0.0278 |
| 7 | replaceCICtoMI | DRC | 0.1342 |
| 7.1 | replaceCICinRHSofASMTtoMI | DRC | 0.0967 |
| 7.1.1 | createStaticMI | CRC | 0.0007 |
| 7.1.2 | replaceCICinRHSofASMTtoMI | ERC | 0.0571 |
| 8 | pullUpMethod | DRC | 0.2803 |
| 8.1 | abstractMethodinSuperclass | DRC | 0.0617 |
| 8.1.1 | createAbstractMethod | CRC | 0.0009 |
| 8.1.2 | addMethod | ERC | 0.0229 |
| 8.2 | pullUpVDF | DRC | 0.0857 |
| 8.2.1 | pullUpFieldHavingOnlyOneVDF | ERC | 0.0565 |
| 8.3 | pullUpMethodNotUsingMember | ERC | 0.0591 |

ERC: ElementaryRefactoringCall, CRC: CreationRefactoringCall, DRC: DefinedRefactoringCall

Table 2 represents the detailed component refactoring calls occurred during the composability test for IPCwFM and the running time. The results were obtained on a Pentium Dual-Core desktop with 4 gigabytes of memory running Windows 7. As expected, the average time spent on processing ERCs is less than that of DRCs. The average time of ERCs, CRCs and DRCs are 0.0411 seconds, 0.0008 seconds and 0.1402 seconds respectively. The total running time and the memory usage for the composability test are 0.8852 seconds, 6.01 megabytes respectively. Unfortunately, there are no similar approaches available to our knowledge to evaluate the results against.

## VII. COMPARATIVE RESEARCH

Roberts (Roberts, 1999) specifies a refactoring by defining both a precondition and a postcondition. He defines primitive analysis functions, which are the base of program analysis, and derived analysis functions, which are derived from the primitive analysis functions. He uses those functions in specifying refactorings. A precondition is described based on first order predicate logic, by using primitive analysis functions and derived analysis functions. A postcondition specifies the interpretation of a primitive analysis function that should be changed after applying a refactoring. Roberts' primitive analysis function, derived analysis function, and postcondition are similar to our model property, user-defined feature, and delta specification, respectively.

In the study of Roberts, the relationship between primitive analysis function and derived analysis function is implicit. But in our study, the relationship between model property and user-defined feature is explicit. Therefore our composability test method, as described in Section 4, can infer which user-defined features' interpretation should be changed, when the interpretation of a model property is changed.

And in the study of Roberts, only primitive analysis functions become the target of change, because derived analysis functions can be updated from the changes of primitive analysis functions. The fact that only model properties are used in our delta specification, is similar to Roberts' study. In the study of Roberts, entire primitive analysis functions become the target to be changed by a refactoring. But in our study, the target of change is localized on the properties of model elements related to a refactoring, because the structure between properties is fixed by a JavaEAST meta model.

The study of Cinnéide (Cinnéide, 2001) is similar to the study of Roberts. He specifies a refactoring by defining both a precondition and a postcondition. He specifies the precondition and postcondition of a refactoring based on first order predicate logic, by defining analysis functions. And for the operations related to creation, which do not impact on the meaning of program, he defines helper functions. The analysis functions of Cinnéide are similar to our model properties and user-defined features, and the helper functions are similar to our creation refactoring. The helper functions of Cinnéide are defined as both a precondition and a postcondition, but our creation refactorings are hard-coded as the Java operations to create model elements.

In the study of Cinnéide, if a new analysis function is added, it should update the postconditions of refactorings that impact on the new function. But in our research, even if a new user-defined feature is added, there is no impact on any delta specification. And in Cinnéide's study, entire analysis functions become the target to be changed by a refactoring, because analysis functions are not layered.

Kniesel and Koch (Kniesel & Koch, 2004) specify a refactoring by defining both a precondition and a backward transformation description. They define conditions that are the base of program analysis based on first order predicate logic, and then they use them to specify preconditions. A backward transformation description specifies the interpretation of a condition that should be changed after applying a refactoring. This backward transformation description is similar to the delta specification of our study.

If a new condition is added, backward transformation descriptions of the already specified refactorings that impact on the new condition should be updated. And entire conditions become the target to be changed by a refactoring, because the conditions are not layered as in the study of Cinnéide.

Li and Thompson (Li & Thompson, 2012) specify a refactoring by defining a precondition. The composability can not be decided in advance before applying a composite refactoring, since there is no a notion of postcondition in Li's study. But our study can decide the composability of a composite refactoring in advance, before applying a composite refactoring, through the composability test using a precondition and a delta specification.

## VIII. CONCLUSION

In this paper, we have defined composite refactorings by assembling component refactorings. And we also have tried to decide composability for the defined composite refactorings. To do this, we have proposed a method to specify elementary refactorings, through both a

precondition and a delta specification. Then, we have proposed a method to decide composability for composite refactorings defined by RCL. Based on these methods, we have developed a prototype tool. And then we have tried to define composite refactorings by using the tool, and to decide composability for the defined composite refactorings through a composability test.

Using a refactoring composition language, we can define new composite refactorings by assembling existing component refactorings. Through this, the reusability of refactorings can be improved. And we can decide the composability of composite refactorings in advance, before applying composite refactorings, through the composability test. It is also easy to draw delta specifications up and maintain them, since we specify elementary refactorings by using OCL and the JavaEAST meta model. If elementary refactorings were specified to guarantee behavior preservation, the behavior preservation of the composite refactoring, which has been defined with these elementary refactorings, could also be guaranteed by using the composability test.

Currently, the tool developed in our study has been designed so as to make the composability test and definition of composite refactorings possible. However, the tool is not able to transform programs as a result of applying refactorings. We plan to automate the transformation of programs using the refactoring effects of elementary refactorings.

## IX. ACKNOWLEDGEMENT

This work was supported by the 2019 Silla University research grant.

## X. REFEREENCES

Benattou, M., Bruel, J.M. & Hameurlain, N. 2002, 'Generating test data from OCL specification', in *Proceedings of the ECOOP'2002 Workshop on Integration and Transformation of UML Models*.

Budinsky, F., Steinberg, D., Ellersick, R., Grose, T.J. & Merks, E. 2004, *Eclipse modeling framework: a developer's guide*, Addison-Wesley Professional.

Cao, K., Kang, I., Choi, H. & Jung, H. 2017, 'Reagent Cabinet Management System Using Danger Priority', *Journal of information and communication convergence engineering*, vol. 15, no. 4, pp. 227-231.

Cinnéide, M.O. 2001, 'Automated application of design patterns: a refactoring approach', PhD thesis, Trinity College Dublin, Dublin, Ireland.

Eckel, B. 2000, *Thinking in Patterns with JAVA Revision 0.5a*, MindView, Inc.

Eclipse EMF 2019, *The Eclipse EMF website*, <https://www.eclipse.org/modeling/emf/>.

Eclipse Modisco 2019, *The Eclipse MoDisco website*, <https://projects.eclipse.org/projects/modeling.mdt.modisco/>.

Eclipse OCL 2019, *The Eclipse OCL website*, <https://projects.eclipse.org/projects/modeling.mdt.ocl>.

Fowler, M. 2018, *Refactoring: improving the design of existing code*, Addison-Wesley Professional.

Huang, J., Carminati, F., Betev, L., Luzzi, C., Lu, Y. & Zhou, D. 2011, 'Identifying composite refactorings with a scripting language', in *2011 IEEE 3rd International Conference on Communication Software and Networks*, pp. 267-271.

Kerievsky, J. 2004, *Refactoring to patterns*, Addison-Wesley Professional.

Kim, K.B. & Song, D.H. 2017, 'Real Time Road Lane Detection with RANSAC and HSV Color Transformation', *Journal of information and communication convergence engineering*, vol. 15, no. 3, pp. 187-192.

Kim, K.M. & Kim, T.G. 2012, 'A Refactoring Composition Language for Composite Refactorings', *Journal of KIISE: Software and Applications*, vol. 39, no. 7, pp. 523-536.

Kim, K.M., Jang, P.J. & Kim, T.G. 2011, 'A Composition Check of Composite Refactorings Not Having a Specification of Precondition', *The KIPS Transactions: PartD*, vol. 18, no. 1, pp. 23-34.

Kniesel, G. & Koch, H. 2004, 'Static composition of refactorings', *Science of Computer Programming*, vol. 52, no. 1-3, pp. 9-51.

Li, H. & Thompson, S. 2012, 'A domain-specific language

for scripting refactorings in erlang', in *International Conference on Fundamental Approaches to Software Engineering*, Springer-Verlag Berlin Heidelberg 2012, pp. 501-515.

Mens, T. & Tourwé, T. 2004, 'A survey of software refactoring', *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126-139.

Ol'Khovich, L. & Koznov, D.V. 2003, 'OCL-based automated validation method for UML specifications', *Programming and Computer Software*, vol. 29, no. 6, pp. 323-327.

OMG, 2014, Object Constraint Language Version 2.4.

Opdyke, W.F. 1992, 'Refactoring object-oriented frameworks', PhD thesis, University of Illinois, Chicago, CA.

Roberts, D.B. 1999, 'Practical analysis for refactoring', PhD thesis, University of Illinois, Chicago, CA.

Saadeh, E. & Kourie, D.G. 2009, 'Composite refactoring using fine-grained transformations', in *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pp. 22-29.

Suryadibrata, A. & Kim, K.B. 2017, 'Ganglion Cyst Region Extraction from Ultrasound Images Using Possibilistic C-Means Clustering Method', *Journal of information and communication convergence engineering*, vol. 15, no. 1, pp. 49-52.

W3C 2019, *The W3C website*, <http://www.w3.org/XML/Schema.html>.

Warmer, JB & Kleppe, AG 2003, *The object constraint language: getting your models ready for MDA*, Addison-Wesley Professional.

Youn, J.H., Seo, Y.H. & Oh, M.S. 2017, 'A study on UI design of social networking service messenger by using case analysis model', *Journal of information and communication convergence engineering*, vol. 15, no. 2, pp. 104-111.

Ziemann, P. & Gogolla, M 2003, 'Validating ocl specifications with the use tool: An example based on the bart case study', *Electronic Notes in Theoretical Computer Science*, vol. 80, pp. 157-169.