

Java Refactoring Case: a VIATRA Solution*

Dániel Stein Gábor Szárnyas István Ráth

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok krt. 2, Budapest, Hungary

daniel.stein@inf.mit.bme.hu, {szarnyas, rath}@mit.bme.hu

This paper presents a solution for the Java Refactoring Case of the 2015 Transformation Tool Contest. The solution utilises Eclipse JDT for parsing the source code, and uses a visitor to build the program graph. EMF-INCQUERY, VIATRA and the Xtend programming language are used for defining and performing the model transformations.

1 Introduction

This paper describes a solution for the extended version of the TTC 2015 Java Refactoring Case. The source code of the solution is available as an open-source project.¹ There is also a SHARE image available.²

The use of automated model transformations is a key factor in modern model-driven system engineering. Model transformations allow the users to query, derive and manipulate large industrial models, including models based on existing systems, e.g. source code models created with reverse engineering techniques. Since such transformations are frequently integrated to modeling environments, they need to feature both high performance and a concise programming interface to support software engineers. EMF-INCQUERY and VIATRA aim to provide an expressive query language and a carefully designed API for defining model queries and transformations.

2 Case Description

Refactoring operations are often used in software engineering to improve the readability and maintainability of existing source code without altering the behaviour of the software. The goal of the Java Refactoring Case [10] is to use model transformation tools to refactor Java source code. We decided to solve the extended version of the case. To achieve this, the solution has to tackle the following challenges:

1. Transforming the *Java source code* to a *program graph* (PG).
2. Performing the refactoring transformation on the program graph.
3. Synchronising the source code and the program graph.

The source code is defined in a restricted sub-language of Java 1.4. The EMF metamodel of the PG is provided in the case description. The case considers two basic refactoring operations: Pull Up Method and Create Superclass. The solution is tested in an automated test framework, ARTE (Automated Refactoring Test Environment).

*This work was partially supported by the MONDO (EU ICT-611125) project.

¹<https://github.com/FTSRG/java-refactoring-ttc-viatra>

²http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64_java-refactoring-viatra.vdi

3 Technologies

Solving the case requires the integration of a model transformation tool and a Java source code parser. In this section, we introduce the technologies used in our solution.

EMF-INCQUERY. The objective of the EMF-INCQUERY [4, 6] framework is to provide a declarative way to define queries over EMF models. EMF-INCQUERY extended the pattern language of VIATRA2 with new features (including transitive closure, role navigation, match count) and tailored it to EMF models, resulting in the INCQUERY Pattern Language [5]. While EMF-INCQUERY is developed with a focus on *incremental query evaluation*, the latest version also provides a *local search-based query evaluation* algorithm.

VIATRA. The VIATRA framework supports the development of model transformations with a particular emphasis on event-driven, reactive transformations [8]. Building upon the incremental query support provided by EMF-INCQUERY, VIATRA offers a language to define transformations and a reactive transformation engine to execute certain transformations upon changes in the underlying model. The current VIATRA project is a full rewrite of the previous VIATRA2 framework, now with full compatibility and support for EMF models.

Java Development Tools. The solution requires a technology to parse the Java code into a program graph model and serialize the modified graph model back to source code. While the case description mentions the JaMoPP [1] and MoDisco [2] technologies, our solution builds on top of the Eclipse Java Development Tools (JDT) [7] used in the Eclipse Java IDE as we were already using JDT in other projects. Compared to the MoDisco framework (which uses JDT internally), we found JDT to be simpler to deploy outside the Eclipse environment, i.e. without defining an Eclipse workspace. Meanwhile, the JaMoPP project has almost completely been abandoned and therefore it is only capable of parsing Java 1.5 source files. While this would not pose a problem for this case, we think it is best to use an actively developed technology such as JDT which supports the latest (1.8) version of the Java language. As JDT is frequently used to parse large source code repositories, it is carefully optimised and supports lazy loading. Unlike JaMoPP and MoDisco, JDT does not produce an EMF model.

4 Implementation

The solution was developed partly in IntelliJ IDEA and partly in the Eclipse IDE. The projects are not tied to any development environment and can be compiled with the Apache Maven [3] build automation tool. This offers a number of benefits, including easy portability and the possibility of continuous integration.

The code is written in Java 8 and Xtend [9]. The queries and transformations were defined in EMF-INCQUERY and VIATRA, respectively. For developing the Xtend code and editing the graph patterns, it is required to use the Eclipse IDE. For setting up the development environment, please refer to the readme file.

4.1 Workflow of the Transformation

Figure 1 shows the high-level workflow of the transformation. This consists of five steps: the source code is parsed into an ASG ①; a PG is produced ②; based on the PG, the possible transformations are

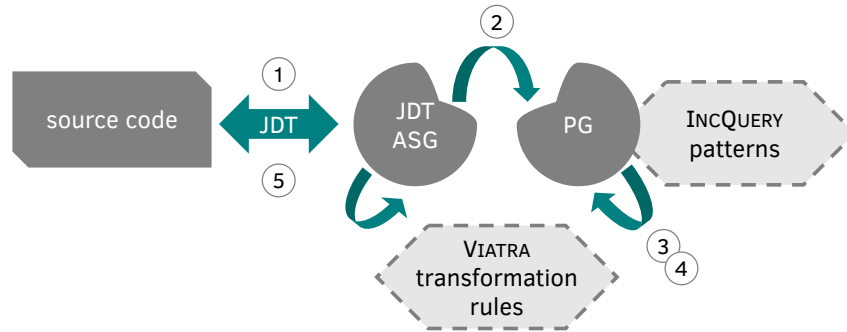


Figure 1: Workflow of the transformation.

calculated ③; if possible, these transformations are executed ④; finally, the results of the transformations are serialized ⑤. In the following, we discuss these steps in detail.

4.2 Parsing the Source Code ①

The solution receives the path to the directory containing the source files. JDT parses these files and returns each file parsed as an AST. These ASTs are interconnected, which means that using JDT’s binding resolution mechanism, the developer can navigate from one AST to another one.

4.3 Producing the Program Graph ②

Since JDT does not produce EMF models, the generated ASTs do not support complex queries and traversal operations as the ones provided by EMF and EMF-based query languages (e.g. Eclipse OCL or EMF-INCQUERY). To extract information and to build the PG, our solution applies a visitor resulting a two-pass traversal on the ASG.

1. For each object, the visitor method creates the corresponding object(s) in the PG. Since the order of these visits is non-deterministic, the visitor maintains maps to store the mapping from the objects in the ASG to the objects in the PG. These maps provide trace information between the JDT model and the partially built PG. The visitor also collects the relations between JDT nodes and caches the unique identifiers of each connected node for every relation type.
2. After every compilation unit has been parsed, the previously populated caches are used to create the cross-references between the objects in the PG (e.g. TMember.access).

4.4 Extending the PG with the Trace Model ②③④

The main patterns for both refactoring operations contain a condition that EMF-INCQUERY does not support out of the box, e.g. checking “every child” (a collection of classes) of a certain class. Passing collections as pattern parameter is only possible with a workaround. Also, the INCQUERY Pattern Language does not support universal quantifiers. To overcome these limitations, we extended the program graph metamodel with a *trace model* shown in Figure 2. The trace model defines traces for method signatures and class lists:

- MethodSignatureTrace. Java methods are uniquely identifiable by their signature. The basic PG metamodel contains a TMethodSignature class, which only identifies itself with the name of the method (using a relation to the TMethod object) and the list of its parameter types.

To support querying TMethodSignature objects with EMF-INCQUERY, we created a trace reference for each of them identified by their partial³ method signature. For example, a method method() expecting a String and an Integer will have the .method(Ljava/lang/String;I) trace signature.

- ClassListTrace. To express the collection of classes, a ClassListTrace object will identify them with their signatures joined by the # character. For example, a list of the ChildClass1 and ChildClass2 classes in the example04 package has the Lexample04/ChildClass1;#Lexample04/ChildClass2; trace signature.

After the PG is produced, it is extended with the trace model. The traces are based on EMF-INCQUERY patterns (Listing 1) and generated with a VIATRA transformation (Listing 4).

The *universal quantifier* is implemented as a double negation of the existential quantifier using the well-known identity $(\forall x)P(x) \Leftrightarrow \neg(\exists x)\neg P(x)$.

4.5 Refactoring ③④

The refactoring operations are implemented as model transformations on the JDT ASG and the PG. Each model transformation is defined in VIATRA: the LHS is defined with an EMF-INCQUERY pattern and the RHS is defined with imperative Xtend code. As VIATRA does not support bidirectional transformations, for each transformation on the PG, we also execute the corresponding actions on the ASG to keep the two graphs in sync.

4.5.1 Pull Up Method

After creating the method signature traces, the following preconditions must be satisfied before pulling up a method:

- every child class has a method with the given signature,
- the parent class does not have a method with this signature,
- the transformation will not create an unsatisfiable method or field access.

To decide whether the refactoring can be executed, every (parent class, method signature) pair satisfying the preconditions is collected by the main pattern (possiblePUM). The LHS is defined with six patterns in total. The execution is controlled by parameterising the main pattern listed in Listing 2. The RHS is defined in Listing 5 using one utility pattern.

4.5.2 Create Superclass

To create a new superclass, the parent class and the list of selected classes (connected to a class list trace) have to be passed to the pattern. The transformation can be executed if the following preconditions are satisfied:

- the target parent class does not exist,
- every selected child class has the same parent.

The LHS is defined in Listing 3 with the possibleCSC pattern using five other patterns. The RHS is defined in Listing 6, also using a utility pattern.

³The complete signature would also contain the defining type (class or interface) signature and the return type signature.

4.6 Transforming the ASG to Source Code ⑤

The changes in the ASG made by the transformations are propagated to the source code. JDT is capable of incrementally maintaining each source code file (compilation unit) based on the changes in its AST.

5 Evaluation

We executed the tests and used the log files to determine the execution times. The execution times of the test cases are listed in Table 1. The results show that all public and hidden test cases have been executed successfully. Hence, we consider the solution *complete* and *correct*. As the test cases only contained small examples, we cannot draw conclusions on the performance of the solution. Still, it is worth noting that all test cases executed in less than half a second.

The implementation of the solution required quite a lot of code. The patterns were formulated in about 150 lines of INCQUERY Pattern Language code. The transformations required 400 lines of Xtend code, while implementing the interface required by ARTE and the visitor for the transformation required more than 800 lines of Java code. However, the source code is well-structured and is easy to comprehend.

6 Summary

This paper presented a solution for the Java Refactoring case of the 2015 Transformation Tool Contest. The solution addresses both challenges (bidirectional synchronisation and program transformation) and both refactoring operations (Pull Up Method, Create Superclass) defined in the case. The framework is flexible enough to allow the user to define new refactoring operations, e.g. Extract Class or Pull Up Field.

Acknowledgements. The authors would like to thank Ábel Hegedüs, Oszkár Semeráth and Zoltán Ujhelyi for providing valuable insights into EMF-INCQUERY and VIATRA.

References

- [1] *JaMoPP*. <http://www.jamopp.org/index.php/JaMoPP>.
- [2] *MoDisco*. <https://eclipse.org/MoDisco/>.
- [3] Apache.org: *Maven*. <http://maven.apache.org>.
- [4] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh & András Ökrös (2010): *Incremental Evaluation of Model Queries over EMF Models*. In: *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS2010*, Springer, Springer, doi:http://dx.doi.org/10.1007/978-3-642-16145-2_6. Acceptance rate: 21%.
- [5] Gábor Bergmann, Zoltán Ujhelyi, István Ráth & Dániel Varró (2011): *A Graph Query Language for EMF models*. In: *Theory and Practice of Model Transformations, Fourth Intl. Conf., LNCS 6707*, Springer.
- [6] Eclipse.org: *EMF-IncQuery*. <http://eclipse.org/incquery/>.
- [7] Eclipse.org: *Java Development Tools (JDT)*. <https://eclipse.org/jdt/>.
- [8] Eclipse.org: *VIATRA Project*. <https://www.eclipse.org/viatra/>.
- [9] Eclipse.org: *Xtend – Modernized Java*. <https://www.eclipse.org/xtend/>.
- [10] Géza Kulcsár, Sven Peldszus & Malte Lochau (2015): *The Java Refactoring Case*. In: *8th Transformation Tool Contest (TTC 2015)*.

A Appendix

A.1 Patterns

```

1 package hu.bme.mit.ttc.refactoring.patterns
2
3 import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"
4
5 pattern methodSignature(methodSignature) {
6   TMethodSignature(methodSignature);
7 }
8
9 pattern tClassName(tClass, className) {
10  TClass(tClass);
11  TClass.tName(tClass, className);
12 }

```

Listing 1: Patterns for generating the trace model.

```

1 package hu.bme.mit.ttc.refactoring.patterns
2
3 import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"
4 import "platform:/plugin/TypeGraphBasic/model/TypeGraphTrace.ecore"
5
6 /*
7  * Main decision pattern. If the preconditions are satisfied (parentClass
8  * and methodSignatureTrace can be bound as parameters), the pattern returns
9  * its parameters, if:
10 * - every child class has a method with the given signature (N = M)
11 * - the parent class does not have it already
12 * - the transformation will not create unavailable access
13 */
14 pattern possiblePUM(parentClass : TClass, methodSignatureTrace : MethodSignatureTrace) {
15   MethodSignatureTrace.tMethodSignature(methodSignatureTrace, methodSignature);
16
17   // every child class has the method signature
18   N == count find childClassesWithSignature(parentClass, _, methodSignature);
19   M == count find childClasses(parentClass, _);
20   check(N == M && N != 0);
21
22   // parent does not already have this method
23   neg find classWithSignature(parentClass, methodSignature);
24
25   // the fields and methods will still be accessible after PUM
26   neg find childrenClassMethodDefinitionsAccessingSiblingMembers(childClass, methodSignature);
27 }
28
29 pattern childClasses(parentClass : TClass, childClass : TClass) {
30   TClass.childClasses(parentClass, childClass);
31 }
32
33 pattern childClassesWithSignature(parentClass : TClass, clazz : TClass, methodSignature : TMethodSignature)
34 {
35   TClass(parentClass);
36   TClass.childClasses(parentClass, clazz);
37   find classWithSignature(clazz, methodSignature);
38 }
39
40 pattern classWithSignature(clazz : TClass, methodSignature : TMethodSignature) {
41   TClass(clazz);
42   TMethodSignature(methodSignature);
43   TMethodSignature.definitions(methodSignature, methodDefinition);
44   TClass.defines(clazz, methodDefinition);

```

```

45 }
46
47 pattern methodsAccessingSiblingMembers(methodDefinition : TMethodDefinition) {
48   TMember.access(methodDefinition, accessedMember);
49   TClass.defines(tClass, methodDefinition);
50   TClass.defines(tClass, accessedMember);
51 } or {
52   TClass.defines(tClass, methodDefinition);
53   TMember.access(methodDefinition, accessedMember);
54   TClass.defines(otherClass, accessedMember);
55   TClass.parentClass.childClasses(tClass, otherClass);
56 }
57
58 pattern childrenClassMethodDefinitionsAccessingSiblingMembers(parentClass : TClass, methodSignature :
    TMethodSignature) {
59   TClass.childClasses(parentClass, childClass);
60   TClass.defines(childClass, methodDefinition);
61   TMethodSignature.definitions(methodSignature, methodDefinition);
62   find methodsAccessingSiblingMembers(methodDefinition);
63 }
64
65 // fire precondition pattern
66 pattern classWithName(tClass : TClass, className) {
67   TClass.tName(tClass, className);
68 }
69
70 // fire precondition pattern
71 pattern methodWithSignature(trace : MethodSignatureTrace, signature) {
72   MethodSignatureTrace.signatureString(trace, signature);
73 }
74
75 // pattern for PG refactor
76 pattern methodDefinitionInClassList(parentClass : TClass, methodSignature : TMethodSignature, clazz :
    TClass, methodDefinition : TMethodDefinition) {
77   TClass.childClasses(parentClass, clazz);
78   TMethodSignature.definitions(methodSignature, methodDefinition);
79   TClass.defines(clazz, methodDefinition);
80 }

```

Listing 2: Patterns for the Pull Up Method refactoring.

```

1 package hu.bme.mit.ttc.refactoring.patterns
2
3 import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"
4 import "platform:/plugin/TypeGraphBasic/model/TypeGraphTrace.ecore"
5
6 /*
7  * Main decision pattern. If the preconditions are satisfied (the
8  * targetClass should not exist), the pattern returns its parameters, if:
9  * - every child class has the same parent
10 */
11 pattern possibleCSC(concatSignature, methodSignature : TMethodSignature) {
12   ClassListTrace.concatSignature(classListTrace, concatSignature);
13   ClassListTrace.tClasses.signature(classListTrace, methodSignature);
14
15   neg find childClassesWithDifferentParents(classListTrace, _, _);
16 }
17
18 pattern childClassesWithDifferentParents(classListTrace : ClassListTrace, classOne : TClass, classTwo :
    TClass) {
19   ClassListTrace.tClasses(classListTrace, classOne);
20   ClassListTrace.tClasses(classListTrace, classTwo);
21   find differentParents(classOne, classTwo);
22 }
23

```

```

24 pattern differentParents(classOne : TClass, classTwo : TClass) {
25   TClass.parentClass(classOne, parentClassOne);
26   TClass.parentClass(classTwo, parentClassTwo);
27   parentClassOne != parentClassTwo;
28 } or {
29   TClass(classTwo);
30   find hasParent(classOne);
31   neg find hasParent(classTwo);
32 } or {
33   TClass(classOne);
34   find hasParent(classTwo);
35   neg find hasParent(classOne);
36 }
37
38 pattern hasParent(tClass : TClass) {
39   TClass.parentClass(tClass, _);
40 }
41
42 pattern classesOfClassListTrace(concatSignature, tClass : TClass) {
43   ClassListTrace.concatSignature(classListTrace, concatSignature);
44   ClassListTrace.tClasses(classListTrace, tClass);
45 }
46
47 pattern methodSignatureAndTrace(trace : MethodSignatureTrace, methodSignature : TMethodSignature) {
48   MethodSignatureTrace.tMethodSignature(trace, methodSignature);
49 }
50
51 // pattern for PG refactor
52 pattern packageWithName(tPackage : TPackage, packageName) {
53   TPackage.tName(tPackage, packageName);
54 }
55
56 // pattern for PG refactor
57 pattern typeGraphs(typeGraph : TypeGraph) {
58   TypeGraph(typeGraph);
59 }
60
61 // fire precondition pattern
62 pattern classWithName(tClass : TClass, className) {
63   TClass.tName(tClass, className);
64 }

```

Listing 3: Patterns for the Create Superclass refactoring.

A.2 Transformations

```

1 package hu.bme.mit.ttc.refactoring.transformations
2
3 import TypeGraphBasic.TClass
4 import TypeGraphTrace.Trace
5 import TypeGraphTrace.TypeGraphTracePackage
6 import hu.bme.mit.ttc.refactoring.patterns.TraceQueries
7 import java.util.ArrayList
8 import java.util.List
9 import org.apache.log4j.Level
10 import org.eclipse.emf.ecore.resource.Resource
11 import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
12 import org.eclipse.incquery.runtime.evm.api.RuleEngine
13 import org.eclipse.incquery.runtime.evm.specific.RuleEngines
14 import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
15 import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
16 import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
17 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory

```



```

18 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
19 import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation
20
21 class TraceTransformation {
22
23     extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
24     extension BatchTransformation transformation
25     extension BatchTransformationStatements statements
26     extension IModelManipulations manipulation
27
28     extension TypeGraphTracePackage tgtPackage = TypeGraphTracePackage::eINSTANCE
29     extension TraceQueries queries = TraceQueries::instance
30     val AdvancedIncQueryEngine engine
31     Resource resource
32     val Trace trace
33
34     new(AdvancedIncQueryEngine engine, Resource resource) {
35         this(RuleEngines.createIncQueryRuleEngine(engine), resource)
36     }
37
38     new(RuleEngine ruleEngine, Resource resource) {
39         engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
40         transformation = BatchTransformation.forEngine(engine)
41         statements = new BatchTransformationStatements(transformation)
42         manipulation = new SimpleModelManipulations(iqEngine)
43         transformation.ruleEngine.logger.level = Level::OFF
44         this.resource = resource
45         this.trace = resource.contents.get(0) as Trace
46     }
47
48     val methodSignatureTraceRule = createRule.precondition(methodSignature).action [
49         val methodSignatureTrace = typeGraphTraceFactory.createMethodSignatureTrace
50         trace.methodSignatures += methodSignatureTrace
51
52         val sb = new StringBuilder(".")
53         sb.append(methodSignature.method.TName)
54         sb.append("(")
55         methodSignature.paramList.forEach[ sb.append(it.TName) ]
56         sb.append(")")
57
58         methodSignatureTrace.signatureString = sb.toString
59         methodSignatureTrace.TMethodSignature = methodSignature
60     ].build
61
62
63     def run() {
64         fireAllCurrent(methodSignatureTraceRule)
65     }
66
67     def addNewClassListTrace(List<String> classSignatures) {
68         val List<TClass> tClasses = new ArrayList
69         for (signature : classSignatures) {
70             tClasses += engine.getMatcher(TClassName).getAllValuesOfTClass(signature)
71         }
72
73         val classListTrace = typeGraphTraceFactory.createClassListTrace
74         classListTrace.concatSignature = classSignatures.join("#")
75         classListTrace.TClasses += tClasses
76
77         trace.classLists += classListTrace
78         return classListTrace
79     }
80 }

```

Listing 4: Transformation for generating the trace model.

```

1 package hu.bme.mit.ttc.refactoring.transformations
2
3 import TypeGraphBasic.TClass
4 import TypeGraphBasic.TypeGraphBasicPackage
5 import TypeGraphTrace.MethodSignatureTrace
6 import com.google.common.collect.BiMap
7 import hu.bme.mit.ttc.refactoring.patterns.PUMQueries
8 import java.io.File
9 import java.util.ArrayList
10 import java.util.List
11 import java.util.Scanner
12 import java.util.Set
13 import org.apache.log4j.Level
14 import org.eclipse.emf.ecore.util.EcoreUtil
15 import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
16 import org.eclipse.incquery.runtime.evm.api.RuleEngine
17 import org.eclipse.incquery.runtime.evm.specific.RuleEngines
18 import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
19 import org.eclipse.jdt.core.dom.ASTNode
20 import org.eclipse.jdt.core.dom.CompilationUnit
21 import org.eclipse.jdt.core.dom.MethodDeclaration
22 import org.eclipse.jdt.core.dom.TypeDeclaration
23 import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
24 import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
25 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory
26 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
27 import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation
28
29 class PUMTransformation {
30     extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
31     extension BatchTransformation transformation
32     extension BatchTransformationStatements statements
33     extension IModelManipulations manipulation
34
35     extension TypeGraphBasicPackage tgPackage = TypeGraphBasicPackage::eINSTANCE
36     extension PUMQueries queries = PUMQueries::instance
37
38     val AdvancedIncQueryEngine engine
39     val String parentSignature
40     val String methodSignature
41     val BiMap<String, CompilationUnit> compilationUnits
42
43     new(AdvancedIncQueryEngine engine, String parentSignature, String methodSignature, BiMap<String,
44         CompilationUnit> compilationUnits) {
45         this(RuleEngines.createIncQueryRuleEngine(engine), parentSignature, methodSignature, compilationUnits)
46     }
47     new(RuleEngine ruleEngine, String parentSignature, String methodSignature, BiMap<String, CompilationUnit>
48         compilationUnits) {
49         engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
50         transformation = BatchTransformation.forEngine(engine)
51         statements = new BatchTransformationStatements(transformation)
52         manipulation = new SimpleModelManipulations(iqEngine)
53         transformation.ruleEngine.logger.level = Level::OFF
54
55         this.parentSignature = parentSignature
56         this.methodSignature = methodSignature
57         this.compilationUnits = compilationUnits
58
59         compilationUnits.values.forEach[ try { it.recordModifications } catch (Exception e) {}]
60     }
61
62     val PUMRule = createRule.precondition(possiblePUM).action [
63         val parentClassKey = parentClass.TName

```

```

63     val childClasses = engine.getMatcher(childClasses).getAllValuesOfchildClass(parentClass)
64
65     var TypeDeclaration astParentClass
66     var List<TypeDeclaration> astChildClasses = new ArrayList
67     var List<MethodDeclaration> astMethodDeclarations
68
69     astParentClass = findCompilationUnits(parentClassKey, childClasses, astChildClasses)
70     astMethodDeclarations = findMethodDeclarations(astChildClasses, methodSignatureTrace)
71
72     updateASTAndSerialize(astParentClass, astChildClasses, astMethodDeclarations)
73
74
75     // ----- /\ JDT transformation ----- PG transformation \/ -----
76
77
78     val methodDefinitionsToDelete = engine.getMatcher(methodDefinitionInClassList).getAllMatches(
79         parentClass, methodSignatureTrace.TMethodSignature, null, null
80     )
81
82     val firstMethodDefinition = methodDefinitionsToDelete.get(0)
83     val savedSignature = firstMethodDefinition.methodSignature
84     val savedReturnType = firstMethodDefinition.methodDefinition.returnType
85     val savedAccess = firstMethodDefinition.methodDefinition.access
86
87     methodDefinitionsToDelete.forEach[
88         it.clazz.signature.remove(it.methodDefinition.signature); // remove signature from class
89         EcoreUtil.delete(it.methodDefinition, true) // remove the method definition
90     ]
91
92     val tMethodDefinition = tgPackage.typeGraphBasicFactory.createTMethodDefinition
93     tMethodDefinition.returnType = savedReturnType
94     tMethodDefinition.signature = savedSignature
95     tMethodDefinition.access += savedAccess
96
97     parentClass.defines += tMethodDefinition
98
99     println(tMethodDefinition)
100 ]build
101
102 protected def readFileToString(String path) {
103     new Scanner(new File(path)).useDelimiter("\\A").next
104 }
105
106 protected def TypeDeclaration findCompilationUnits(String parentClassKey, Set<TClass> childClasses, List<
107     TypeDeclaration> astChildClasses) {
108     var TypeDeclaration result
109     for (cu : compilationUnits.values) {
110         // the just created CU can not resolve
111         val firstTypeKey = "L"
112             + cu.package.name.qualifiedName.replace('.', '/')
113             + "/"
114             + ((cu.types.get(0) as TypeDeclaration).name.qualifiedName)
115             + ";"
116
117         if (parentClassKey.equals(firstTypeKey)) {
118             result = cu.types.get(0) as TypeDeclaration
119         }
120
121         for (child : childClasses) {
122             if (firstTypeKey.equals(child.TName)) {
123                 astChildClasses += cu.types.get(0) as TypeDeclaration
124             }
125         }
126     }

```

```

127     return result
128 }
129
130 protected def List<MethodDeclaration> findMethodDeclarations(List<TypeDeclaration> astChildClasses,
131     MethodSignatureTrace methodSignatureTrace) {
132     val List<MethodDeclaration> astMethodDeclarations = new ArrayList
133
134     for (childCU : astChildClasses) {
135         val methodSignature = childCU.resolveBinding.key + methodSignatureTrace.signatureString;
136         val types = (childCU.root as CompilationUnit).getStructuralProperty(CompilationUnit.TYPES_PROPERTY)
137         as List<TypeDeclaration>
138         for (type : types) {
139             for (method : (type as TypeDeclaration).methods) {
140                 if (method.resolveBinding.key.startsWith(methodSignature)) {
141                     astMethodDeclarations += method
142                 }
143             }
144         }
145     }
146     return astMethodDeclarations
147 }
148
149 protected def updateASTAndSerialize(TypeDeclaration astParentClass, List<TypeDeclaration> astChildClasses
150     , List<MethodDeclaration> astMethodDeclarations) {
151     if (astMethodDeclarations.size > 0) {
152         astParentClass.bodyDeclarations.add(ASTNode.copySubtree(astParentClass.AST, astMethodDeclarations.get
153             (0)) as MethodDeclaration)
154     }
155     for (methodDeclaration : astMethodDeclarations) {
156         methodDeclaration.delete
157     }
158 }
159
160 def fire() {
161     fireAllCurrent(
162         PUMRule,
163         "parentClass.tName" -> parentSignature,
164         "MethodSignatureTrace.signatureString" -> methodSignature
165     )
166 }
167
168 def canExecutePUM() {
169     // get the method signature by string, then get one arbitrary match with it bound
170     val parentTClass = engine.getMatcher(classWithName).getOneArbitraryMatch(null, parentSignature)
171     val trace = engine.getMatcher(methodWithSignature).getOneArbitraryMatch(null, methodSignature)
172
173     return
174     parentTClass != null &&
175     trace != null &&
176     engine.getMatcher(possiblePUM).getOneArbitraryMatch(parentTClass.TClass, trace.trace) != null
177 }
178 }

```

Listing 5: Pull Up Method transformation.

```

1 package hu.bme.mit.ttc.refactoring.transformations
2
3 import TypeGraphBasic.TClass
4 import TypeGraphBasic.TMethodSignature
5 import TypeGraphBasic.TPackage
6 import TypeGraphBasic.TypeGraph
7 import TypeGraphBasic.TypeGraphBasicPackage
8 import com.google.common.collect.BiMap

```

```

9 import hu.bme.mit.ttc.refactoring.patterns.CSCQueries
10 import java.io.File
11 import java.util.ArrayList
12 import java.util.List
13 import java.util.Scanner
14 import java.util.Set
15 import org.apache.commons.lang3.StringUtils
16 import org.apache.log4j.Level
17 import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
18 import org.eclipse.incquery.runtime.evm.api.RuleEngine
19 import org.eclipse.incquery.runtime.evm.specific.RuleEngines
20 import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
21 import org.eclipse.jdt.core.dom.ASTNode
22 import org.eclipse.jdt.core.dom.CompilationUnit
23 import org.eclipse.jdt.core.dom.MethodDeclaration
24 import org.eclipse.jdt.core.dom.Modifier.ModifierKeyword
25 import org.eclipse.jdt.core.dom.Name
26 import org.eclipse.jdt.core.dom.Type
27 import org.eclipse.jdt.core.dom.TypeDeclaration
28 import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
29 import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
30 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory
31 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
32 import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation
33
34 class CSCTransformation {
35     extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
36     extension BatchTransformation transformation
37     extension BatchTransformationStatements statements
38     extension IModelManipulations manipulation
39
40     extension TypeGraphBasicPackage tgPackage = TypeGraphBasicPackage::eINSTANCE
41     extension CSCQueries queries = CSCQueries::instance
42
43     val AdvancedIncQueryEngine engine
44     val String concatSignature
45     val String targetPackage
46     val String targetName
47     val BiMap<String, CompilationUnit> compilationUnits
48
49     var CompilationUnit targetCU
50
51     new(AdvancedIncQueryEngine engine, List<String> childClassSignatures, String targetPackage, String
        targetName, BiMap<String, CompilationUnit> compilationUnits) {
52         this(RuleEngines.createIncQueryRuleEngine(engine), childClassSignatures, targetPackage, targetName,
            compilationUnits)
53     }
54
55     new(RuleEngine ruleEngine, List<String> childClassSignatures, String targetPackage, String targetName,
        BiMap<String, CompilationUnit> compilationUnits) {
56         engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
57         transformation = BatchTransformation.forEngine(engine)
58         statements = new BatchTransformationStatements(transformation)
59         manipulation = new SimpleModelManipulations(iqEngine)
60         transformation.ruleEngine.logger.level = Level::OFF
61
62         this.concatSignature = childClassSignatures.join("#")
63         this.targetPackage = targetPackage
64         this.targetName = targetName
65         this.compilationUnits = compilationUnits
66
67         compilationUnits.values.forEach[ try { it.recordModifications } catch (Exception e) {}]
68     }
69
70     val CSCRule = createRule.precondition(possibleCSC).action [

```

```

71     val tClasses = engine.getMatcher(classesOfClassListTrace).getAllValuesOfTClass(concatSignature)
72
73     val List<TypeDeclaration> astChildClasses = findCompilationUnits(tClasses)
74
75     val firstChild = astChildClasses.get(0)
76
77     if (targetCU == null) {
78         targetCU = createTargetClass(firstChild, firstChild.superclassType)
79     }
80
81     setParentClass(astChildClasses)
82
83     serializeCUs
84
85
86     // ----- /\ JDT transformation ----- PG transformation \/ -----
87
88     val oldParentTClass = tClasses.get(0).parentClass
89     if (oldParentTClass != null) {
90         oldParentTClass.childClasses -= tClasses
91     }
92
93     val targetSignature = "L" + targetPackage.replace('.', '/') + "/" + targetName + ";";
94     val typeGraph = engine.getMatcher(typeGraphs).oneArbitraryMatch.typeGraph
95
96     val targetTClassMatch = engine.getMatcher(classWithName).getOneArbitraryMatch(null, targetSignature)
97     var TClass targetTClass
98     if (targetTClassMatch == null) {
99         targetTClass = tgPackage.typeGraphBasicFactory.createTClass
100        targetTClass.TName = targetSignature
101
102        targetTClass.package = createPackagesFor(typeGraph, targetPackage)
103        targetTClass.parentClass = oldParentTClass
104    } else {
105        targetTClass = targetTClassMatch.TClass
106    }
107
108    (tClasses.get(0).eContainer as TypeGraph).classes += targetTClass
109    targetTClass.childClasses += tClasses
110    ].build
111
112    protected def createPackagesFor(TypeGraph typeGraph, String pkg) {
113        val String[] split = pkg.split("\\.");
114
115        var previous = "";
116        var TPackage previousTPackage
117        for (var i = 0; i < split.length; i++) {
118            var String current = previous
119            if (i != 0) {
120                current += "."
121            }
122            current += split.get(i);
123
124            var currentTPackageMatch = engine.getMatcher(packageWithName).getOneArbitraryMatch(null, current)
125            if (currentTPackageMatch != null) {
126                previousTPackage = currentTPackageMatch.TPackage
127            } else {
128                val TPackage currentTPackage = tgPackage.typeGraphBasicFactory.createTPackage
129                currentTPackage.TName = current
130                if (previousTPackage != null) {
131                    currentTPackage.parent = previousTPackage
132                } else {
133                    typeGraph.packages += currentTPackage
134                }
135            }

```

```

136     previousTPackage = currentTPackage
137   }
138 }
139
140 previousTPackage
141 }
142
143 protected def List<TypeDeclaration> findCompilationUnits(Set<TClass> childClasses) {
144   val List<TypeDeclaration> astChildClasses = new ArrayList
145
146   for (cu : compilationUnits.values) {
147     for (child : childClasses) {
148       if (cu.findDeclaringNode(child.TName) != null) {
149         astChildClasses += cu.findDeclaringNode(child.TName) as TypeDeclaration
150       }
151     }
152   }
153
154   return astChildClasses
155 }
156
157 protected def List<MethodDeclaration> findMethodDeclarations(List<TypeDeclaration> astChildClasses,
158   TMethodSignature tMethodSignature) {
159   val List<MethodDeclaration> astMethodDeclarations = new ArrayList
160   val methodSignatureTrace = engine.getMatcher(methodSignatureAndTrace).getAllValuesOfTrace(
161     tMethodSignature).get(0)
162
163   for (childCU : astChildClasses) {
164     val methodSignature = childCU.resolveBinding.key + methodSignatureTrace.signatureString;
165     val types = (childCU.root as CompilationUnit).getStructuralProperty(CompilationUnit.TYPES_PROPERTY)
166     as List<TypeDeclaration>
167     for (type : types) {
168       for (method : (type as TypeDeclaration).methods) {
169         // match
170         if (method.resolveBinding.key.startsWith(methodSignature)) {
171           astMethodDeclarations += method
172         }
173       }
174     }
175   }
176
177   return astMethodDeclarations
178 }
179
180 protected def CompilationUnit createTargetClass(TypeDeclaration childClass, Type superClassType) {
181   val ast = childClass.AST
182   val compilationUnit = ast.newCompilationUnit
183
184   if (targetPackage != null) {
185     val packageDeclaration = ast.newPackageDeclaration
186     var Name packageName
187     for (part : targetPackage.split("\\\\")) {
188       if (packageName == null) {
189         packageName = ast.newSimpleName(part)
190       } else {
191         packageName = ast.newQualifiedName(packageName, ast.newSimpleName(part))
192       }
193     }
194     packageDeclaration.name = packageName
195     compilationUnit.package = packageDeclaration
196   }
197
198   compilationUnit.imports += ASTNode.copySubtrees(ast, (childClass.root as CompilationUnit).imports)
199
200   val typeDeclaration = ast.newTypeDeclaration

```

```

198     typeDeclaration.modifiers().add(ast.newModifier(ModifierKeyword.PUBLIC_KEYWORD))
199     typeDeclaration.name = ast.newSimpleName(targetName)
200
201     if (superClassType != null) {
202         typeDeclaration.superclassType = ASTNode.copySubtree(ast, superClassType) as Type
203     }
204
205     compilationUnit.types += typeDeclaration
206
207     compilationUnit
208 }
209
210 protected def insertMethodDeclaration(MethodDeclaration declaration) {
211     val typeDeclaration = targetCU.types.get(0) as TypeDeclaration
212     typeDeclaration.bodyDeclarations.add(ASTNode.copySubtree(targetCU.AST, declaration) as
        MethodDeclaration)
213 }
214
215 protected def setParentClass(List<TypeDeclaration> typeDeclarations) {
216     val ast = targetCU.AST
217
218     var Type fqn
219     if (targetPackage != null) {
220         for (part : targetPackage.split("\\\\")) {
221             if (fqn == null) {
222                 fqn = ast.newSimpleType(ast.newSimpleName(part))
223             } else {
224                 fqn = ast.newQualifiedType(fqn, ast.newSimpleName(part))
225             }
226         }
227
228         fqn = ast.newQualifiedType(fqn, ast.newSimpleName(targetName))
229     } else {
230         fqn = ast.newSimpleType(ast.newSimpleName(targetName))
231     }
232
233     for (declaration : typeDeclarations) {
234         declaration.superclassType = ASTNode.copySubtree(declaration.AST, fqn) as Type
235     }
236 }
237
238 protected def removeChildMethodDeclarations(List<MethodDeclaration> methodDeclarations) {
239     for (declaration : methodDeclarations) {
240         declaration.delete
241     }
242 }
243
244 def serializeCUs() {
245     val targetDir = StringUtils.substringBefore(
246         compilationUnits.keySet.get(0),
247         "/src/"
248     ) + "/src/" + targetPackage.replace('.', '/')
249     val targetPath = targetDir + "/" + targetName + ".java"
250
251     val targetFile = new File(targetPath)
252     targetFile.parentFile.mkdirs
253
254     compilationUnits.put(targetPath, targetCU)
255 }
256
257 protected def readFileToString(String path) {
258     new Scanner(new File(path)).useDelimiter("\\A").next
259 }
260
261

```



```

262 def fire() {
263     fireAllCurrent(
264         CSCRule,
265         "concatSignature" -> concatSignature
266     )
267 }
268
269 def canExecuteCSC() {
270     val targetSignature = "L" + targetPackage.replace('.', '/') + "/" + targetName + ";"
271     val targetTClass = engine.getMatcher(classWithName).getOneArbitraryMatch(null, targetSignature)
272
273     if (targetTClass != null) {
274         return false
275     }
276
277     engine.getMatcher(possibleCSC).countMatches > 0
278 }
279
280 }

```

Listing 6: Create Superclass transformation.

A.3 Metamodel of the Trace Model

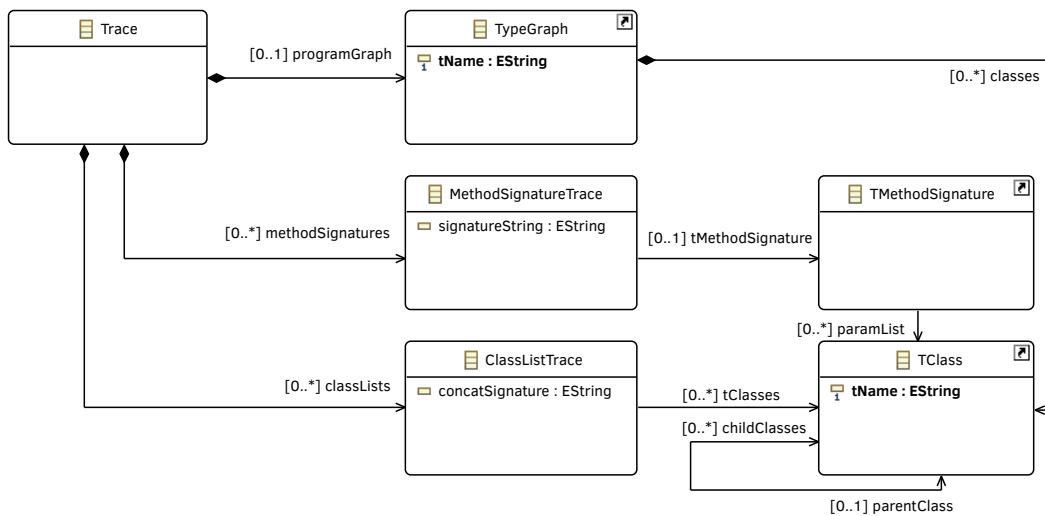


Figure 2: Metamodel of the trace model.

A.4 Benchmark Results

The benchmarks were conducted on a 64-bit Arch Linux virtual machine running in SHARE. The machine utilized a single core of a 2.00 GHz Xeon E5-2650 CPU and 1 GB of RAM. We used OpenJDK 8 to run the ARTE framework and the solution.

test case	time [s]
pub_pum1_1_paper1	0.463
pub_pum1_2	0.013
pub_pum2_1	0.333
pub_pum3_1	0.094
pub_csc1_1	0.189
pub_csc1_2	0.093
hidden_pum1_1	0.063
hidden_pum1_2	0.013
hidden_pum2_1	0.114
hidden_pum2_2	0.082
hidden_csc1_1	0.081
hidden_csc1_2	0.007
hidden_csc2_1	0.058
hidden_csc3_1a	0.189
hidden_csc3_1	0.179

Table 1: Execution times for the test cases.