An NMF solution to the Java Refactoring Case at the TTC 2015

Georg Hinkel

Forschungszentrum Informatik (FZI)
Haid-und-Neu-Straße 10-14
Karlsruhe, Germany
hinkel@fzi.de

Abstract Lehman's laws state that dedicated efforts must be spent for any software artifact to prevent a loss of quality. For code, such efforts are called refactoring operations and are an important aspect of many software engineers day-to-day business. Many of these refactoring operations are specified on a much higher abstraction level than the actual source code of a given language like Java. To be able to specify these refactoring operations on a higher abstraction level as proposed in the Java Refactoring case at the Transformation Tool Contest (TTC) 2015, we propose a solution using an incremental synchronization with NMF Synchronizations of the source code regarded as a model on the one side and a simplified program graph model on the other side.

1 Introduction

This paper proposes a solution for the Java Refactoring Case[1] at the Transformation Tool Contest (TTC) 2015. Our solution is publicly available on CodePlex[2] and SHARE[3] and built upon the .NET Modeling Framework[4] (NMF) and especially on NMF Synchronizations[1].

All of the technologies used in this solution are implemented as internal languages hosted by C#. The reason for this is that we try to let developers stay with the language that they are most confident with as much as possible as recent research suggest that they will hardly change them voluntarily [2]. One of the reasons for this especially in an MDE context is that many transformation languages lack the tool support offered by mainstream languages such as Java or C# [3], [4].

As our solution is based on internal languages, we do not have this problem and thus, our solution is entirely specified in C# (besides JaMoPP). However,

[1] https://github.com/Echtzeitsysteme/java-refactoring-ttc/raw/master/Case_Description-final.pdf

[2] http://ttc2015javarefactoringnmf.codeplex.com/

[3] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64-TTC15_NMF.vdi

[4] http://nmf.codeplex.com

we were facing issues converting the JaMoPP models back to Java source files, which we were unable to resolve. As a consequence, the solution is only create the refactored JaMoPP models, but not the Java files. The solution is in detail in the next section.

2 Solution with NMF Synchronizations

We use JaMoPP [5] to translate Java files into a model representation. Since the NMF meta-metamodel NMeta is compatible with Ecore, we can easily transform the JaMoPP metamodel to an NMeta metamodel and consume the JaMoPP generated XMI representations of the input files directly. In between, we use the model synchronization language of NMF to refactor the Java files.

NMF Synchronizations is a bridge between NMF Transformations[6], [7], the model transformation language within NMF[5], and NMF Expressions[6], which is responsible for the incremental evaluation of arbitrary expressions. Basically, NMF Synchronizations uses NMF Expressions to make model transformations bidirectional and incremental, i.e. any changes of either left hand side (LHS) or right hand side (RHS) of the model can be propagated to the other. This change propagation is optional and can be chosen by the developer when running the transformation. In total, we support 18 different modes of operation, namely six directions and three different modes of change propagation (none, one way or two way).

Although NMF Synchronizations offers support for bidirectional model transformations, we do not use employ this feature. The reason is that the model transformation task of the proposed case conflicts with our definition of a model transformation being basically a function from one metamodel to another. Therefore, we classify the task of the present case study rather as a model synchronization task. We transform the Java model (we use JaMoPP) to a Program Graph model, modify this Program Graph model and propagate these changes back to the original JaMoPP model.

Currently, NMF Synchronizations only supports online synchronization. This means, any changes in the Program Graph model are immediately reflected in the JaMoPP model. In particular, the model synchronization adds hooks into the Program Graph model and reacts on changes in that it applies these changes to the JaMoPP model. As a consequence, the backward transformation from the case description and the refactoring operation on the PG get However, we do not support a one-way change propagation mode in the opposite direction of the transformation, and so we have selected the two way change propagation mode. That is, any changes in either of the JaMoPP model or the Program Graph model will be reflected in the other model.

We are aware that this causes some overhead when the Program Graph model is used only for a one-time refactoring, and we will add a change propagation mode one way to source in the future. Originally, when NMF Synchronizations

[5] `http://nmf.codeplex.com`

[6] `http://nmfexpressions.codeplex.com`

was designed, we could not think of a useful case study for this change propagation mode, but the present case study offers a good one.

2.1 Synchronization of JaMoPP and PG

Model synchronizations in NMF Synchronizations are classes and the synchronization rules are represented by public non-abstract nested classes. Listing 1 shows an excerpt of the model synchronization that synchronizes classes.

```
1  class JavaPGSynchronization : ReflectiveSynchronization {
2    public class Class2Class : SynchronizationRule<IClass, ITClass> {
3      public override void DeclareSynchronization() {
4        Synchronize(cl => cl.Name, cl => cl.TName);
5        SynchronizeMany(SyncRule<Member2Member>(),
6          cl => cl.Members.Where(m => m is ClassMethod || m is Field),
7          cl => cl.Defines);
8        Synchronize(this,
9          cl => cl.Extends as IClassifierReference != null
10           ? (cl.Extends as IClassifierReference).Target as IClass
11           : null, RegisterNewBaseClass,
12         cl => cl.ParentClass);
13     }
14   }
15 }
```

Listing 1. Synchronization of classes in JaMoPP and the PG metamodel

In line 3, we declare that `Class2Class` is a synchronization rule synchronizing JaMoPP classes with PG classes. Line 7 specifies that whenever we found such two classes that correspond (decided by another method called `ShouldCorrespond`), their names should be synchronized. Line 8-10 specify that each member of a JaMoPP class should correspond to a definition in the PG. The details for this correspondence are left to the `Member2Member` rule.

Lines 11-15 specify that the base classes should be synchronized. The current rule (`Class2Class` should be used to identify corresponding base classes as well, explaining the `this` parameter in Line 11. However, whereas the base class of a Java class in the PG metamodel is available directly as a reference, the base class in JaMoPP is encoded in a classifier reference, making the expression to obtain the base class slightly more complex. As a consequence, NMF Synchronizations is not able to infer how to revert the expression and we have to specify this (how a JaMoPP class is assigned another class as a base class) through another method, namely `RegisterNewBaseClass`. With this method, the behavior how to assign a JaMoPP class a new base class is implemented in regular imperative code.

The implementation of `Member2Member` in case of methods is presented in Listing 2.

```
public class Method2MethodDefinition : SynchronizationRule<IMethod, ITMethodDefinition> {
  public override bool ShouldCorrespond(IMethod left, ITMethodDefinition right, ISynchronizationContext
      context) {
    var sig = right.Signature;
    if (sig == null) return false;
    var meth = sig.Method;
    if (meth == null) return false;
    return left.Name == meth.TName;
```

```
8   }
9   public override void DeclareSynchronization() {
10     MarkInstantiatingFor(SyncRule<Member2MemberDefinition>());
11     Synchronize(meth => meth.Name, meth => meth.Signature.Method.TName);
12     LeftToRight.Require(Rule<Method2MethodSignature>(), meth => meth.Name,
13        meth => meth.Parameters.Select(p => GetBaseClass(p.TypeReference)).AsItemEqual(),
14        (meth, signature) => meth.Signature = signature);
15   }
16 }
```

Listing 2. The synchronization rule for method definitions

In this listing, again Line 1 declares `Method2MethodDefinition` as a synchronization rule from JaMoPP methods to PG method definitions. A JaMoPP method should correspond to a PG method definition in a given scope if the methods have the same name here. We specify the exact behavior in lines 3-10. Since the structure of the PG metamodel is very different to JaMoPP in this regard, the method is a few lines long.

Line 13 marks the synchronization rule instantiating for the `Member2Member`rule. That is, if a member is a method, then the rule `Method2MethodDefinition` should be used to synchronize members, regardless of the transformation direction.

Line 14 denotes that the name of a method in JaMoPP should be kept consistent with the name of the method in the PG model. If we changed the name of a method in JaMoPP, the change is propagated to the PG `TMethod` element. However, this change is propagated back to the JaMoPP model causing all methods that are connected to this PG method to change their name accordingly, regardless of their declaration scope or signature. So we have specified a very powerful rename refactoring in just a single line of code.

NMF Synchronizations under the hood uses the transformation engine of NMF Transformations. In particular, every synchronization rule is mapped to a pair of transformation rules, one for either direction of the synchronizations. Since these transformation rules are still accessible, we can add a dependency to the `Method2MethodSignature` that creates a `TMethodSignature` element for the given name and parameter list. For a given name and parameter list, the transformation engine ensures that only one method signature element is created. This transformation rule calls another rule `Method2Method` that creates a method element for each string that appears as a method name in the JaMoPP model.

These transformation rules `Method2Method` and `Method2MethodSignature` are called any time the *LeftToRight* rule of the synchronization rule `Method2MethodDefinition` are called. That is done either initially for each method in the JaMoPP model (restricted to at most once per input names and parameter lists) and as well for any new JaMoPP method that is added to the JaMoPP model afterwards.

2.2 Refactoring of the PG Graph

The refactoring part of our solution uses straight forward imperative code to achieve the refactoring operations. As the *Create Superclass* is straight forward to implement in classic C# code, we omit a description. The implementation of the *Pull Up Method* refactoring is shown in Listing 3.

```
public bool PullUpMethod(TypeGraph typeGraph) {
 foreach (var method in typeGraph.Methods) {
  foreach (var signature in method.Signatures) {
   var methodsGroupsToPull = from def in signature.Definitions
                        where def.Overriding == null
                        group def by (def.Parent as TClass).ParentClass into methods
                        select methods;
   foreach (var methodGroup in methodsGroupsToPull.Where(group => group.Count() >= 2)) {
    if (methodGroup.Key != null) {
     var first = methodGroup.First();
     var firstParent = first.Parent as ITClass;
     methodGroup.Key.Defines.Add(first);
     firstParent.Defines.Remove(first);
     foreach (var m in methodGroup.Skip(1)) {
      (m.Parent as ITClass).Defines.Remove(m);
     }}}}}}
```

Listing 3. The implementation of *Pull Up Method*

The solution utilizes the Language Integrated Query (LINQ) that is around for almost ten years now and used by thousands of developers. Given the conciseness of this specification based on the `TypeGraph` metamodel, we see no reason to use a specialized language for the refactoring. However, due to the online synchronization, we have to be careful to always keep the model in a consistent state, we must not discard the method that should stay as otherwise the connected implementation in the JaMoPP model would be lost.

References

[1] G. Hinkel, "Change propagation in an internal model transformation language," in *Theory and Practice of Model Transformations*, accepted, to appear, Springer, 2015.

[2] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, ACM, 2013, pp. 1–18.

[3] M. Staron, "Adopting model driven software development in industry–a case study at two companies," in *Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 57–72.

[4] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases," *Empirical Software Engineering*, vol. 18, no. 1, pp. 89–116, 2013.

[5] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, *Jamopp: The java model parser and printer.* Techn. Univ., Fakultät Informatik, 2009.

[6] G. Hinkel, "An approach to maintainable model transformations using an internal DSL," Master's thesis, Karlsruhe Institute of Technology, 2013.

[7] G. Hinkel and L. Happe, "Using component frameworks for model transformations by an internal DSL," in *1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp 2014)*, ser. CEUR, 2014.