Compiling Graph Transformation Rules into a Procedural Language for Behavioral Modeling

Sabine Winetzhammer and Bernhard Westfechtel
Applied Computer Science I, University of Bayreuth, Bayreuth, Germany

Keywords: Graph Transformation Rules, Behavioral Modeling, Code Generation.

Abstract: Graph transformation rules provide an opportunity to specify model transformations in a declarative way at a high level of abstraction. So far, compilers have translated graph transformation rules into conventional programming languages such as Java, C, or C#. In contrast, we have developed a compiler which translates graph transformation rules into a procedural language for behavioral modeling (Xcore). The generated code is significantly more concise and readable than programming language code. Furthermore, the code is portable since it is completely programming language independent.

1 INTRODUCTION

Model transformation languages have been developed for specifying transformations of models at a higher level of abstraction than in conventional programming languages. Among many features (Czarnecki and Helsen, 2006), model transformation languages may be classified according to their underlying paradigm: In *procedural languages*, the transformation is described by specifying the order in which elementary transformation steps are executed. In contrast, *rule-based languages* specify transformations by a set of rules for matching and replacing patterns. Since the algorithms for pattern matching and replacement need not be provided by the user, rule-based languages are located at a higher level of abstraction than procedural languages.

A model may be considered as a *graph* whose nodes and edges correspond to the model's objects and links. *Graph transformation rules* (Ehrig et al., 1999) are ideally suited for specifying model transformations in a declarative way. Essentially, a graph transformation rule consists of a left-hand side and a right-hand side. The left-hand side describes the graph pattern to be searched, while the right-hand side defines the replacing pattern. Quite a number of graph transformation languages have been proposed, including PROGRES (Schürr et al., 1999), Fujaba (Norbisrath et al., 2013), GReAT (Agrawal et al., 2006), GrGen.NET (Jakumeit et al., 2010), Henshin (Arendt et al., 2010), MDELab (Giese et al., 2009), VIATRA2 (Varró and Balogh, 2007), eMOFLON (Anjorin et al., 2011), and ModGraph (Buchmann et al., 2011). Users of these languages specify transformations with the help of high-level graph transformation rules. Users are not concerned with the algorithms for pattern matching and replacement, which are taken care of by the underlying execution engines.

To support the execution of graph transformation rules, both *interpreters* and *compilers* have been developed. An interpreter provides excellent support for debugging, which is slowed down by a compiler. On the other hand, compiled code is more efficient. So far, compilers have translated graph transformation rules into conventional programming languages such as Java, C, or C#. This approach results in rather complicated generated code which is difficult to understand.

In contrast, we have built a *compiler* which translates *graph transformation rules* into a *procedural language* for *behavioral modeling* (Figure 1). The compiler accepts *ModGraph* rules and translates them into *Xcore* (Eclipse Foundation, 2013), a recently developed modeling language which is based on Ecore. Xcore is a textual language which covers both structural and behavioral modeling. Our compiler transforms ModGraph rules into procedural Xcore operations, specifically making use of Xcore's expression language. The Xcore environment in turn translates Xcore into Java (and prospectively into other target languages in the future). Within our work we follow our goal to provide total model driven software engineering as explained in (Winetzhammer and Westfechtel, 2013). Xcore interacts with ModGraph in or-

Winetzhammer S. and Westfechtel B..
Compiling Graph Transformation Rules into a Procedural Language for Behavioral Modeling.
DOI: 10.5220/0004723604150424
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 415-424
ISBN: 978-989-758-007-9
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

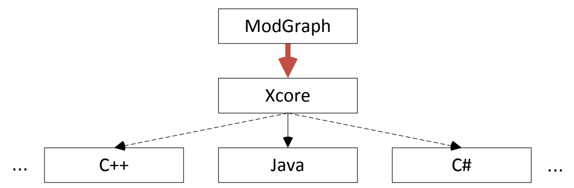

Figure 1: Staged translation.

der to provide high level control structures for rules. The translation to Xcore unifies the level of abstraction between rules and control flow.

This *staged translation approach* (Figure 1) provides the following advantages over the traditional approach of compiling into a conventional programming language directly, which is followed by all competing tools:

Conciseness. The generated code is concise (but it still takes care of the details of pattern matching and replacement which should be shielded from the user).

Readability. The generated code is human readable, which facilitates e.g. code-level debugging.

Simplicity. The task of compiling is simplified significantly since Xcore provides more high-level language constructs than conventional programming languages such as Java.

Portability. With direct compilation into a programming language, one compiler is required for each target language. In our approach, the compiler does not depend on the programming language which is eventually used for execution.

2 OVERVIEW

The *Eclipse Modeling Framework (EMF)* (Steinberg et al., 2009) has been designed with the intent to improve the software process by providing lightweight support for model-driven software engineering. For this reason, EMF provides a fairly minimalistic metamodel for structural modeling (*Ecore*, an implementation of Essential MOF (EMOF) (OMG, 2011)). Using the components of the EMF core, software engineers create Ecore models as instances of the Ecore metamodel. From an Ecore model, the EMF code generator creates code for classes, including methods for creating objects, assigning attribute values, as well as creating and deleting links which implement the semantics of Ecore. However, for user-defined operations, the EMF code generator may only create empty method bodies, which have to be filled in by the user.

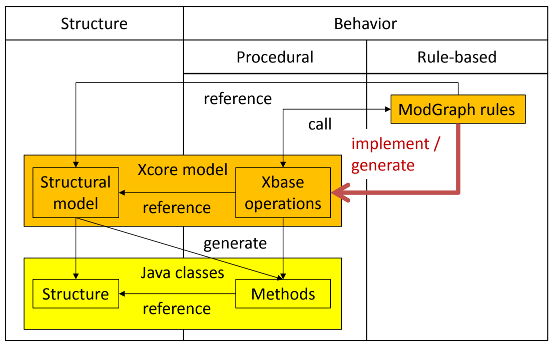

Figure 2: Interplay between ModGraph and Xcore.

Xcore (Eclipse Foundation, 2013) adds behavioral modeling to EMF. Xcore provides a single language for both structural and behavioral modeling. To this end, Xcore introduces a textual syntax for Ecore models as well as procedural behavioral models. Xcore is driven by the vision that software engineers need no longer deal with code in a programming language such as Java (as current programmers do not inspect assembly or byte code). In Xcore, the sublanguage Xbase (Efftinge et al., 2012) is used to model behavior, i.e. the bodies of operations. Xbase is an expression language that was designed to be reused in different domain-specific languages. Xbase expressions provide both control structures and program expressions in a uniform way. Its program expressions may be used e.g. for navigation in models and checking constraints. Altogether, Xbase programs specify computations in a procedural way at a higher level of abstraction than Java.

ModGraph (Buchmann et al., 2011) is an EMF-based language for specifying graph transformation rules. With ModGraph, an operation defined in an Ecore model may be realized as a *graph transformation rule* (or *rule* in short form). A graph pattern forms the core of a ModGraph rule. The graph pattern describes both the pattern to be searched and the replacing pattern in a single diagram. If no replacement is specified, the rule describes a test or a query rather than a transformation. In addition to the core, a rule may comprise optional components such as textual pre- and postconditions and graphical negative application conditions (NACs).

A graph pattern may be composed of several kinds of nodes and edges. Nodes are distinguished into a current object, named `this`, bound nodes, representing the non-primitive parameters of the operation, and unbound nodes, representing the objects to be searched in the model instance. Both may be single- (simple object and parameter) or multi-valued nodes (multi-object and multi-parameter). Nodes provide a status which may be preserved (grey, no marker), Listing 1: Definition of annotations.

```
annotation "http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot" as OCL
annotation "http://www.eclipse.org/emf/2002/Ecore" as Ecore
annotation "http://www.eclipse.org/emf/2002/GenModel" as GenModel

@GenModel(loadInitialization="true",operationReflection="true",
        modelDirectory="/[somePath]/src")
@Ecore(invocationDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
        settingDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
        validationDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot")
```

created (green, ++), or deleted (red, −−). They can be marked as return parameter (<<out>>) or as optional (<<optional>>) nodes. Nodes to preserve or to delete may be constrained, nodes to create or to preserve may be modified, for example by setting an attribute value or calling an operation (operation calls allow ModGraph rules to interact directly with each other). All nodes may be connected by two kinds of edges: links (instances of references) and paths (derived references). Analogously to nodes, links provide a status. Creation links instantiating multi-valued references may be ordered. Paths are marked with a path expression, written in OCL or Xbase. Negative application conditions describe patterns which must not occur when the main pattern has been matched. NACs are specified in a similar way as graph patterns; however, nodes and edges do not have a status and nodes may only be single-valued.

The *interplay* between *ModGraph* and *Xcore* is illustrated in Figure 2. The user defines the structural model in Xcore's textual notation. With respect to behavioral modeling, the user may choose between the procedural and the rule-based paradigm. Simple operations may be defined directly in Xbase. Complex operations may be specified in ModGraph, taking advantage of its expressiveness and its easily readable graphical notation. If a complex operation may not be coded as a single rule, the user may resort to Xbase control structures for controlling the application of multiple rules. In general, Xbase operations may call ModGraph rules and vice versa. For the purpose of execution, ModGraph rules are first compiled into Xcore operations. The second stage of compilation (currently targeting Java) is performed by the Xcore compiler. Please note that the user gets in touch only with Xcore and ModGraph (orange boxes); there is no need to inspect the generated Java code (yellow boxes).

In the following sections, we will focus on the ModGraph2Xcore compilation (see red and bold arrow in Figure 2).

3 CODE GENERATION

This section explains from scratch how ModGraph generates Xcore code and injects it into the Xcore model.

3.1 Preliminaries for Code Generation

The initial step of integrating a ModGraph rule into an Xcore model is parsing. We use a recursive, heuristic greedy algorithm to transform the rule's graph pattern into a forest of spanning trees. The forest specifies the reachability of nodes inside the graph pattern. A node is reachable if it can be accessed from a non-primitive parameter of the method or the current object using links. The forest acts as a search plan and is built in the following way:

(1) Each bound node in the pattern acts as root of a tree inside the forest.

(2) Regarding all outgoing edges of the forest's nodes, select the instance of the reference with minimum multiplicity. Consider paths as multiplicity many references. If two links instantiating references of the same multiplicity exist, choose one randomly. (3) Check if the link's target node is mandatory and not contained in any tree yet. If true, add it as a child into the tree containing the source node. (4) Repeat steps (2) and (3) for all mandatory nodes. If any mandatory node cannot be inserted into a tree it cannot be bound, is therefore unreachable, and the matching fails. (5) Repeat – without the mandatory check – steps (2) and (3) for all optional nodes. (This prevents the search of a mandatory object from an optional one.)

For more information on the pattern matching process, please refer to (Winetzhammer, 2012).

3.2 Injecting Code into the Xcore Model

Annotating the Xcore Model: Using ModGraph with Xcore means adding OCL support and some Genmodel specifications to the original model as

Listing 2: Xcore Model for Refactoring.

```
class Refactoring {
        refers EOperation[] referenceToEOperation
        refers EClass[] referenceToEClass
        refers EReference[] referenceToEReference
        refers EParameter[] referenceToEParameter
        refers EStructuralFeature[] referenceToEStructuralFeature

        op void changeUniToBidirectionalReference(
                String class1Name , String class2Name)
        op void collapseHierarchy(ClassType classType ,
                                EClass superClass , EClass subClass)
        op void removeSub(EClass superClass , EClass subClass)
        op void removeSuper(EClass superClass , EClass subClass)
}
```

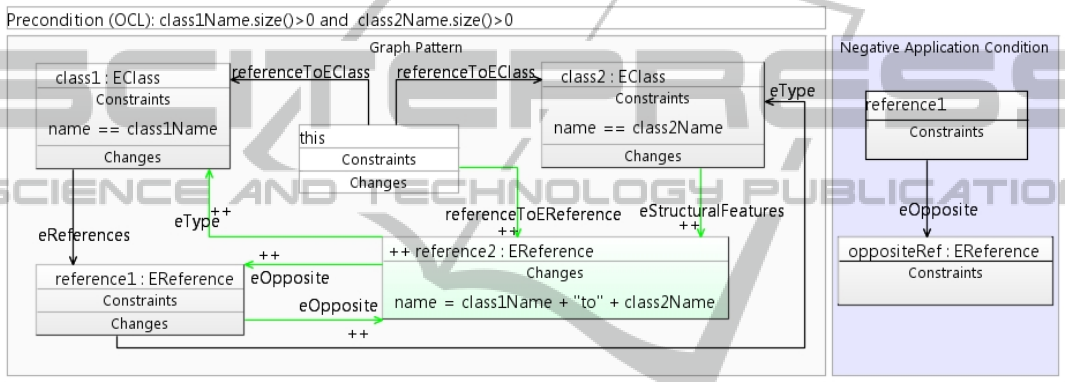

Figure 3: ModGraph rule for `changeUniToBidirectionalReference(String class1Name , String class2Name)`.

shown in Listing 1. Therefore we define annotations (including Xcore aliases) to ensure that the Xcore code generator works with correct parameters.

Concerning OCL we add EMF provided invocation, setting and validation delegates using the Eclipse OCL Pivot evaluation. The Pivot evaluator is used here because of its full OMG compliance [1].

Furthermore we add some Genmodel directives, e.g. to ensure the operation reflection is set true to make OCL work or for documentation purposes.

Please note that these changes are applied once for each Xcore model, no matter how many operations are implemented within ModGraph.

Annotating the Xcore Operation: In this step we parse the Xcore model until the rule's implemented operation is found. The generator annotates the operation implemented by the ModGraph rule depending on it's content.

In each case it creates an Xcore Genmodel documentation annotation. This annotation is used twofold: It contains a note that this Xcore code is generated by ModGraph as well as the comment on a ModGraph rule.

Using ModGraph, you may write pre- and postconditions in OCL or Xbase. If there are any OCL pre- or postconditions, they are translated into Xcore OCL annotations. (Xbase conditions are integrated into the operation's body within the subsequent step.)

Implementing the Operation's Body with Xbase: Inside the operation's body, the generated Xbase code is structured as follows: (1)Check the rule's preconditions written in Xbase. (2)Define all variables needed to perform the transformation. (3) Generate code for pattern matching using nested for-loops. (4)Within the most inner loop check the negative application condition using an additional if-condition. (5) If any object cannot be matched, an exception of type GTFailure is thrown. If all objects could be matched, go on. (6) Calculate all attribute values on the pre-state of the model. These values are stored in final variables. (7) Delete or change objects and links between them and create new ones. (8) If there is any opera-

[1]See eclipse help: http://help.eclipse.org/kepler/index.jsp

tion call inside the rule, call the operation. (9) Check Xcore postconditions. (10) Return what needs to be returned.

4 EXAMPLE

This section provides concrete examples on the code generation mechanism described above. Therefore we consider two refactoring operations on an Ecore model: changing an unidirectional reference to a bidirectional one and collapsing the hierarchy between two classes as defined by Fowler (Fowler, 1999). These examples have been selected (and slightly adapted for demonstration purposes) from a bachelor thesis (Dümmel, 2013) in which a much more comprehensive set of refactoring operations has been implemented with ModGraph transformation rules.

The structure of our simple refactoring is described textually in Xcore as shown in Listing 2. A refactoring class references elements of the Ecore model to be refactored, and defines the refactoring operations to be applied. Each refactoring operation is applied to model elements fixed by parameters. The operations are invoked through an interactive user interface. For demonstration purposes, both strings and objects are used to identify model objects. In the first example, the use of string parameters implies the insertion of nested loops into the generated code. In the second example, objects are used instead of strings to focus on other issues of code generation. In the actual implementation (Dümmel, 2013), model objects are identified consistently by strings for implementation-specific reasons.

To get a clear impression of the pattern matching in Xcore, we consider the refactoring rule to change a unidirectional reference into a bidirectional one as shown in Figure 3. Since only the classes' names are given as parameters of the operation, a precondition, written in OCL ensures them not to be empty or null. The graph in the graph pattern shows the pattern to be matched: starting at the current object named `this`, two classes with the given names need to be found. The class we call `class1` needs to contain a reference typed over `class2`. In that case the negative application condition (NAC) ensures that the reference does not have an opposite set yet. If all these conditions are fulfilled, an opposite reference is created and embedded into the model by setting the links marked with ++ and colored green.

The Xcore code generated for this rule is shown in Listing 3. In line 1 a Genmodel annotation is used to mark the operation as generated by ModGraph. Line 2 checks the OCL precondition (in Figure 3 at the top), using the EMF OCL Pivot evaluator via an annotation. Line 3 & 4 show the Xcore generated operation head. For each bound object in the rule's graph pattern, the generator declares variables as shown in lines 5-7. Lines 8-19 show the matching, which is implicitly given in the ModGraph rule. Nested for-loops are built up according to the spanning forest described in 3.1. These for-loops also use the Xbase λ-expression language to filter the collections they iterate by the constraints given to the objects in the rule, e.g. `name == class1Name`. The most inner loop contains an if condition, that checks the NAC. If matching succeeds, the variables defined above the loops are initialized. Unfortunately Xcore does not support break-commands. Therefore, the variables are initialized with the last match found[2]. Line 19 checks if matching has succeeded; otherwise, an exception is raised.

Line 20 shows the calculation of the name for the new reference depending on the pre-state of the model. The reference itself is created in line 21 and its name is set to the calculated one in line 22. Lines 23 to 27 put the new reference into its context executing the following expressions: The new reference's opposite is set to the existing one and vice versa. `class2` is set as a container for the new reference by adding it to its structural features. The refactoring class adds the new reference and the reference's type is set to `class1`.

The second refactoring rule shown here is collapse hierarchy. It shows the interplay of procedural and rule-based operations. For simplification we assume that there is only one subclass to a superclass[3]. Collapsing a hierarchy means eliminating either the sub- or the superclass. Both possibilities may be modeled in separate ModGraph rules implementing the methods `removeSub` and `removeSuper`. The rule to remove the subclass is shown in Figure 4. An Xcore precondition ensures the parameters not to be null[4]. The graph pattern shows the superclass and the subclass as well as all its operations and structural features, which need to be shifted to the superclass. Additionally all references typed over the subclass need to be retyped by the superclass. All features are marked optional because if there is none, the rule may be executed anyway. The ModGraph rule for removing the superclass works analogously.

[2]Since we expect that break-commands will be added soon to Xcore, we refrain from rewriting the generated code with more awkward while-loops returning the first match.

[3]The presence of another subclass may be excluded by a negative application condition in a similar way as already demonstrated in Figure 3.

[4]This precondition is redundant and has been added to demonstrate that Xcore expressions may be used alternatively to OCL expressions.

Listing 3: Generated Xcore implementation of method `changeUniToBidirectionalReference`.

```
@GenModel(documentation="Generated by ModGraph. ")
@OCL(pre_pre1="class1Name.size()>0 and  class2Name.size()>0 ")
op void changeUniToBidirectionalReference(
                                String class1Name , String class2Name) {
        var EClass class1 = null
        var EClass class2 = null
        var EReference reference1 = null
        for (_class1 : referenceToEClass.filter(e|e.name == class1Name)) {
          for (_class2 : referenceToEClass.filter(e|e.name == class2Name)){
            for (_reference1 : class1.EReferences.filter(e|e.EType.equals(_class2))) {
                  if (! ( reference1.EOpposite != null )) {
                                class1 = _class1
                                class2 = _class2
                                reference1 = _reference1
                  }
            }
          }
        }
        if(class1 == null) throw new GTFailure
        val reference2NameValue = class1Name + "to" + class2Name
        var reference2 = EcoreFactory::eINSTANCE.createEReference()
        reference2.name = reference2NameValue
        reference2.EOpposite = reference1
        reference1.EOpposite = reference2
        class2.EStructuralFeatures.add(reference2)
        referenceToEReference.add(reference2)
        reference2.EType = class1
}
```

Listing 4: Xcore implementation of method `collapseHierarchy`.

```
1  op void collapseHierarchy(ClassType classType, EClass superClass, EClass subClass){
2                 if(classType == ClassType::SUPER_CLASS)
3                  removeSuper(superClass, subClass)
4                 if(classType == ClassType::SUB_CLASS)
5                  removeSub(superClass, subClass)
6          }
```

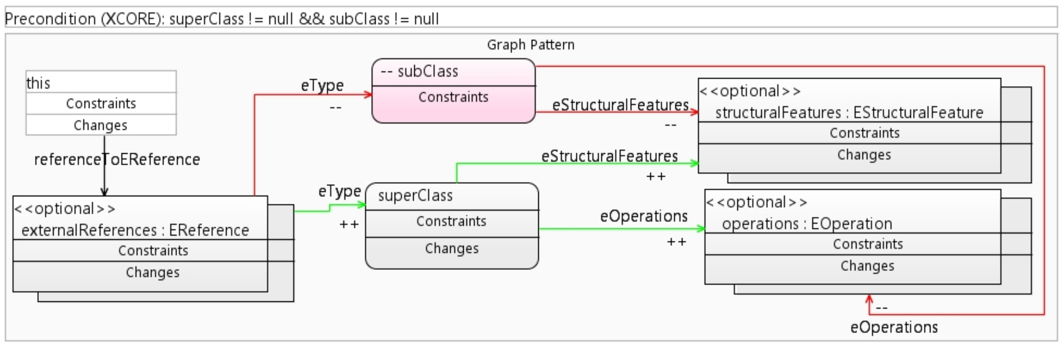

Figure 4: ModGraph rule for `removeSub(EClass superClass , EClass subClass)`.

As we do not want to split up this refactoring, we use Xcore's procedural capabilities and write a third method encapsulating the other two. This method is called `collapseHierarchy`. It acts as a control flow determining which of the two generated methods will be called. Its Xcore implementation is shown in lines 1-6 in Listing 4.

Listing 5 shows the generated Xcore code for the ModGraph rule `removeSub`. Lines 1 and 2 show the comment complementing the rule. Line 3 contains the head of the method. Line 4 checks the Xcore precondition. Lines 5–7 declare variables for storing matches of multi-objects. In lines 8–11, values for these variables are retrieved which are assigned in lines 12–14. Line 15 removes the subclass. Lines 16 and 17 assign the structural features and operations

Listing 5: Generated Xcore implementation of method `removeSub`.

```
1  @GenModel(documentation="Generated by ModGraph: Removes the
2           subclass. Part of the collapse hierarchy refactoring. ")
3  op void removeSub(EClass superClass , EClass subClass) {
4           if(! ( superClass != null && subClass != null )) throw new GTFailure
5           var EList<EStructuralFeature> structuralFeatures = null
6           var EList<EOperation> operations = null
7           var EList<EReference> externalReferences = null
8           val _operations = subClass.EOperations
9           val _structuralFeatures = subClass.EStructuralFeatures
10          val _externalReferences = referenceToEReference
11                  .filter(e|e.EType.equals(subClass)).asEList
12          structuralFeatures = _structuralFeatures
13          operations = _operations
14          externalReferences = _externalReferences
15          org::eclipse::emf::ecore::util::EcoreUtil::remove(subClass)
16          superClass.EStructuralFeatures.addAll(structuralFeatures)
17          superClass.EOperations.addAll(operations)
18          externalReferences.forEach(e|e.EType = superClass)
19          }
20
21 op void removeSuper(EClass superClass , EClass subClass) {
22          /*analogously to removeSub*/
23          }
```

to the superclass, respectively, and line 18 retypes the references.

5 DISCUSSION

This section discusses the advantages of our staged transformation approach. To this end, we consider the three implementations to delete a subclass in an Ecore model in order to collapse the hierarchy: the ModGraph rule (Figure 4), the generated Xcore implementation (Listing 5), and the Xcore generated Java code (Listing 6).

Comparing the ModGraph rule to the Xcore implementation, we observe that a rule is still more intuitive than the generated code: its clearly structured format with the graphical, color-coded, nodes and edges visualize the pattern to be matched and the actions to be performed.

The Xcore code is a clearly structured, target language independent text which we consider to be still concise and simple enough to be human readable. Its high level of abstraction increases the readability especially when the functional expressions provided by Xbase come into play.

The generated Xcore code shown in Listing 5 could be written more concisely if written by hand. In fact, lines 5–14 could be expressed by only three lines of hand-written code. In contrast, the code generator creates declarations of variables which are assigned values only when a complete match has been found. During the matching, final variables are used to store partial matches. In this way, it can be checked conveniently whether matching has succeeded (if it has not, the non-final variables will still be null). This code generation approach supports the most general case, in which matching has to be performed in (potentially nested) loops (see Listing 3 for the refactoring rule converting a unidirectional to a bidirectional reference).

This example demonstrates that hand-written code may be shorter than generated code. This is not surprising and quite common. Nevertheless, the generated code is still concise and readable. Thus, debugging may be performed quite conveniently on the generated code.

The result of compiling the Xcore code of Listing 5 to Java code is shown in Listing 6. Comparing them, we note a significant difference in length: The generated Java code is much longer than the Xcore code. Furthermore, the Java code is much more difficult to read. This results from Xcore's higher level abstraction. Compare, e.g., the precondition initially written as one line Xcore expression in the rule in Figure 4 and the Xcore implementation in Listing 5. The Java implementation uses lines 6–19 to ensure this condition. A closer look at the Xcore generated Java code reveals that internal functions need to be called or even implemented. The filter function shown in Listing 5, lines 10–11 to filter the references typed

Listing 6: Xcore Generated Java Code for method `removeSub`.

```
/**
 * <!-- begin-user-doc --> <!-- end-user-doc -->
 * @generated
 */
 public void removeSub(final EClass superClass, final EClass subClass){
    try {
      boolean _and = false;
      boolean _notEquals = (!Objects.equal(superClass, null));
      if (!_notEquals) {
        _and = false;
      } else {
        boolean _notEquals_1 = (!Objects.equal(subClass, null));
        _and = (_notEquals && _notEquals_1);
      }
      boolean _not = (!_and);
      if (_not)  {
        GTFailure _gTFailure = new GTFailure();
        throw _gTFailure;
      }
      EList<EStructuralFeature> structuralFeatures = null;
      EList<EOperation> operations = null;
      EList<EReference> externalReferences = null;
      final EList<EOperation> _operations = subClass.getEOperations();
      final EList<EStructuralFeature> _structuralFeatures =
                 subClass.getEStructuralFeatures();
      Refactoring _this = this;
      EList<EReference> _referenceToEReference =
                _this.getReferenceToEReference();
      final Function1<EReference,Boolean> _function =
                 new Function1<EReference,Boolean>()
      {
          public Boolean apply(final EReference e){
            EClassifier _eType = e.getEType();
            boolean _equals = _eType.equals(subClass);
            return Boolean.valueOf(_equals);
          }
        };
      Iterable<EReference> _filter = IterableExtensions.<EReference>filter(
          _referenceToEReference, _function);
      final EList<EReference> _externalReferences = ECollections.<EReference>asEList(
          ((EReference[])Conversions.unwrapArray(_filter, EReference.class)));
      structuralFeatures = _structuralFeatures;
      operations = _operations;
      externalReferences = _externalReferences;
      EcoreUtil.remove(subClass);
      EList<EStructuralFeature> _eStructuralFeatures =
                             superClass.getEStructuralFeatures();
      _eStructuralFeatures.addAll(structuralFeatures);
      EList<EOperation> _eOperations = superClass.getEOperations();
      _eOperations.addAll(operations);
      final Procedure1<EReference> _function_1 = new Procedure1<EReference>()
      {
          public void apply(final EReference e) {
            e.setEType(superClass);
          }
        };
      IterableExtensions.<EReference>forEach(externalReferences, _function_1);
    } catch (Throwable _e){
      throw Exceptions.sneakyThrow(_e);
      }
 }
```

Table 1: Graph transformation languages and tools.

language/tool	*interpreter*	*compiler*	*target language(s) (if compiled)*
eMOFLON (Anjorin et al., 2011)	x	x	Java
Fujaba (Norbisrath et al., 2013)	x	x	Java
GReAT (Agrawal et al., 2006)	x	-	-
GrGen.NET (Jakumeit et al., 2010)	-	x	C#
Henshin (Arendt et al., 2010)	x	-	-
MDELab (Giese et al., 2009)	x	-	-
ModGraph	(x)	x	*Xcore* or Java
PROGRES (Schürr et al., 1999)	x	x	C or Java
VIATRA2 (Varró and Balogh, 2007)	-	x	Java

over the subclass is mapped to lines 38–41 in Listing 6. An additional Java filter function is called to map the procedural expression to Java. The foreach-expression shown in Listing 5, line 18, even forces a re-implementation to be mapped to Java, see lines 51–57.

Altogether, these considerations reinforce the claims stated at the end of Section 1: In fact, the generated Xcore code is concise, readable, and simple. In addition, it is also portable since Xcore could be mapped to other target languages, as well. Thus, we have clearly demonstrated the advantages of the staged translation approach (Figure 1).

6 RELATED WORK

In (Winetzhammer and Westfechtel, 2013) we already gave an overview on how Xcore and ModGraph interact on *code level*. Here we make Xcore and ModGraph interact on *model level*.

Code level interaction means that a graph transformation rule is specified which is based on a method defined in an Xcore model; subsequently, Java code is generated from the ModGraph rule and injected into the Xcore generated Java code. That means the Xcore model is completely independent of the ModGraph rule and leads to a fully compiled solution: both tools *meet at the generated Java code*.

The approach described here goes beyond our previous work. As before, an operation defined in an Xcore model is specified by a ModGraph rule. However, the implemented operation is not compiled directly to Java code. Instead, we use a *staged translation approach*: The rule is compiled into Xcore code and injected at *model level* into the textual Xcore model. Hence we do not leave the *model level* as we do in (Winetzhammer and Westfechtel, 2013). In this way, ModGraph code is *independent* of the *programming language*. In our new approach, the code generation to Java (and possibly other languages) is completely left to the Xcore generator. Another advantage of compiling into a procedural behavioral modeling language relies on the fact that such a language resides on a higher level of abstraction than a conventional programming language.

As a side note, we mention that generation of Xcore code implies partial support of interpretation of ModGraph rules: Since the generated code is quite concise and human readable, the Xcore interpreter may be used conveniently for testing and debugging ModGraph rules. This is a low-cost alternative to implementing a full-fledged rule-level interpreter.

Concluding, we developed a compiler from graph transformation rules using a procedural language for behavioral modeling. The result of compilation is interpretable as well as compilable. A staged transformation approach is used to make the generated code independent of the programming language which is eventually used for execution.

Table 1 provides a short comparison of related tools / languages. Here we consider only tools related to EMF and based on the theory of graph transformation. Some tools provide a direct interpreter. Quite a number of tools compile graph transformation rules into widely used programming languages such as C, C#, or Java. Only ModGraph provides *model-level code generation* (into Xcore code). None of the competing tools supports a *staged translation approach* as illustrated in Figure 1.

7 CONCLUSIONS

We have presented a new approach of compiling high level graph transformation rules into a procedural language for behavioral modeling (Xcore). Using this approach, the modeler may resort to graph transformation rules for complex operations, while simple operations may be directly implemented in Xcore using Xbase. We described the mechanisms of injecting graph transformation rules into the Xcore model.

The resulting code may be compiled as well as interpreted. It is much more concise, readable, and simple than programming language code due to the fact that we do not leave the modeling level. Furthermore, the Xcore code is portable since it is programming language independent. The approach presented here is unique with respect to these properties: All competing tools for generating code from graph transformation rules create code in a conventional programming language (see Section 6).

REFERENCES

Agrawal, A., Karsai, G., Neema, S., Shi, F., and Vizhanyo, A. (2006). The design of a language for model transformations. *Software and Systems Modeling*, 5:261–288.

Anjorin, A., Lauder, M., Patzina, S., and Schürr, A. (2011). eMoflon: Leveraging EMF and Professional CASE Tools. In *INFORMATIK 2011*, volume 192 of *Lecture Notes in Informatics*, page 281, Bonn. Gesellschaft für Informatik, Gesellschaft für Informatik. extended abstract.

Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: Advanced concepts and tools for in-place EMF model transformations. In Petriu, D. C., Rouquette, N., and Haugen, Ø., editors, *Proceedings 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135, Oslo, Norway. Springer.

Buchmann, T., Westfechtel, B., and Winetzhammer, S. (2011). ModGraph — A Transformation Engine for EMF Model Transformations. In *Proceedings of the 6th International Conference on Software and Data Technologies (ICSOFT 2011)*, pages 212 – 219, Sevilla, Spain.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646.

Dümmel, N. (2013). Refactoring mit Graphtransformationsregeln. Bachelor thesis, University of Bayreuth, Bayreuth, Germany.

Eclipse Foundation (2013). *Xcore*. http://wiki.eclipse.org/Xcore.

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., Hasselbring, W., von Massow, R., and Hanus, M. (2012). Xbase: Implementing domain-specific languages for java. In *GPCE '12 Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, pages 112–121. ACM, New York, NY, USA.

Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., editors (1999). *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Giese, H., Hildebrandt, S., and Seibel, A. (2009). Improved flexibility and scalability by interpreting story diagrams. In Boronat, A. and Heckel, R., editors, *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, volume 18 of *Electronic Communications of the EASST*, York, UK. 12 p.

Jakumeit, E., Buchwald, S., and Kroll, M. (2010). GrGen.NET — the expressive, convenient and fast graph rewrite system. *International Journal on Software Tools for Technology Transfer*, 12:263–271.

Norbisrath, U., Zündorf, A., and Jubeh, R. (2013). *Story Driven Modeling*. CreateSpace Independent Publishing Platform. ISBN-10: 1483949257.

OMG (2011). *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1*. OMG.

Schürr, A., Winter, A., and Zündorf, A. (1999). The PROGRES approach: Language and environment. In *Handbook of Graph Grammars and Computing by Graph Transformation: vol. 2: Applications, Languages, and Tools*, pages 487–550. World Scientific Publishing.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley, Boston, MA, 2nd edition.

Varró, D. and Balogh, A. (2007). The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234.

Winetzhammer, S. (2012). ModGraph — generating executable EMF models. In Margaria, T., Padberg, J., Taentzer, G., Krause, C., and Westfechtel, B., editors, *Proceedings of the 7th International Workshop on Graph Based Tools*, volume 54 of *Electronic Communications of the EASST*, pages 32–44, Bremen, Germany. EASST.

Winetzhammer, S. and Westfechtel, B. (2013). ModGraph meets Xcore: Combining rule-based and procedural behavioral modeling for EMF. In Tichy, M., Ribeiro, L., Margaria, T., Padberg, J., and Taentzer, G., editors, *Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2013)*, volume 58 of *Electronic Communications of the EASST*, page 13 p., Rome, Italy. EASST.