Compiling Graph Transformation Rules into a Procedural Language for Behavioral Modeling

Sabine Winetzhammer and Bernhard Westfechtel

Applied Computer Science I, University of Bayreuth, Universitaetstrasse 30, 95440 Bayreuth, Germany
{sabine.winetzhammer, bernhard.westfechtel}@uni-bayreuth.de
http://btn1x4.inf.uni-bayreuth.de/modgraph/homepage

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 [5], 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 [9] 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 [15], Fujaba [13], GReAT [1], GrGen.NET [12], Henshin [3], MDELab [11], VIATRA2 [17], eMOFLON [2], and ModGraph [4]. 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* (Fig. 1). The compiler accepts *ModGraph* rules and translates them into *Xcore* [7], 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 [20] and [21]. Xcore interacts with ModGraph in order 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* (Fig. 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.

---

1 This paper is an extension of [21] including parts of [20].
2 Overview

The *Eclipse Modeling Framework (EMF)* [16] 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) [14]). 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.

*Xcore* [7] 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 [8] 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* [4] is an EMF-based language for specifying graph transformation rules. With ModGraph, an operation defined in an Ecore or Xcore 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), 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. Pre- and postconditions may be written in OCL or Xbase.

The interplay between ModGraph and Xcore is illustrated in Fig. 2. The user defines the structural model in Xcore’s textual notation or migrates an existing Ecore model to Xcore. With respect to behavioral modeling, the user may choose between the procedural and the rule-based paradigm. Simple operations may be defined directly in Xcore using Xbase to implement its body. 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 Fig. 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 building a search plan. 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 [19].

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 shown in Listing 1.1. 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 \(^2\).

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

\(^2\) See eclipse help: http://help.eclipse.org/kepler/index.jsp
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 as implemented by the ModGraph rule depending on its 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 innermost 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 operation 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 a unidirectional reference to a bidirectional one and collapsing the hierarchy between two classes as defined by Fowler [10]. These examples have been selected (and slightly adapted for demonstration purposes) from a bachelor thesis [6] 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 1.2. A refactoring class references the 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
Listing 1.2. 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 className, 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)
}

Fig. 3. ModGraph rule for changeUniToBidirectionalReference

used instead of strings to focus on other issues of code generation. In the actual implement-
mentation [6], model objects are identified consistently by strings for implementation-
specific reasons.

To get a clear impression of the pattern matching we use in Xcore, we consider the
refactoring rule to change a unidirectional reference into a bidirectional one as shown
in Fig. 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 1.3. In line 1 a Genmodel
annotation is used to mark the operation as generated by ModGraph. Line 2 checks the
OCL precondition (in Fig. 3 at the top), using the EMF OCL Pivot evaluator via an
annotation. Lines 3 and 4 show the Xcore generated operation head. For each unbound
object in the rule’s graph pattern, the generator declares variables as shown in lines
5–7. Lines 8–20 show the matching, which is implicitly given in the ModGraph rule.
Listing 1.3. Generated Xcore implementation of method changeUniToBidirectionalReference

```java
@GenModel(documentation="Generated by ModGraph.")

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

  var 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
}
```

Nested for-loops are built up according to the spanning forest described in 3.1. These for-loops also use the Xbase $\lambda$-expression language to filter the collections they iterate by the constraints given to the objects in the rule, e.g. name == class1Name. The innermost 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$^3$.

Line 20 checks if matching has succeeded; otherwise, an exception is raised.

Line 21 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 22 and its name is set to the calculated one in line 23. Lines 24–28 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$^4$. Collapsing a hierarchy means eliminating either the

---

$^3$ 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.

$^4$ The presence of another subclass may be excluded by a negative application condition in a similar way as already demonstrated in Fig. 3.
Listing 1.4. Xcore implementation of method collapseHierarchy

```xcore
op void collapseHierarchy(ClassType classType, EClass superClass, EClass subClass) {
  if (classType == ClassType::SUPERCLASS) try {
    removeSuper(superCl, subCl)
  } catch (GTFailure f) { /* do sth. */
  }
  if (classType == ClassType::SUBCLASS) try {
    removeSub(superCl, subCl)
  } catch (GTFailure f) { /* do sth. */
  }
}
```

Fig. 4. ModGraph rule for removeSub

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 Listing 1.4. The method selection is realized by a condition, depending on the given value of enumeration classType. Exception handling, using try-catch-blocks, is necessary because each method implemented by a graph transformation rule may raise an exception of type GTFailure.\(^5\)

The rule to remove the subclass is shown in Fig. 4. An Xcore precondition ensures the parameters not to be null\(^6\). 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 can be executed anyway. The ModGraph rule for removing the superclass works analogously.

Listing 1.5 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

\(^5\) This is true for the latest Xcore version. Using try-catch-blocks was an implementation decision, of course, one may also delegate the exception to the calling method.

\(^6\) This precondition is redundant and has been added to demonstrate that Xcore expressions may be used alternatively to OCL expressions.
Listing 1.5. Generated Xcore implementation of method `removeSub`

```xcore
@GenModel (documentation = "Generated by ModGraph: Removes the subclass. Part of the collapse hierarchy refactoring.")
op void removeSub (EClass superClass, EClass subClass) {
  if (! (superClass != null && subClass != null)) throw new GTFailure
  var EList<E StructuralFeature> structuralFeatures = null
  var EList<EOperation> operations = null
  var EList<EReference> externalReferences = null
  val _operations = subClass.EOperations
  val _structuralFeatures = subClass.EStructuralFeatures
  val _externalReferences = referenceToEReference
    .filter (e | e.EType.equals (subClass)).asEList
  structuralFeatures = _structuralFeatures
  operations = _operations
  externalReferences = _externalReferences
  org::eclipse::ecore::util::EcoreUtil::remove (subClass)
  superClass.EStructuralFeatures.addAll (structuralFeatures)
  superClass.EOperations.addAll (operations)
  externalReferences.forEach (e | e.EType = superClass)
}

op void removeSuper (EClass superClass, EClass subClass) {
  /* analogously to removeSub */
}
```

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

5 Discussion

This section discusses the advantages of our staged transformation approach.

First, we consider the three implementations to delete a subclass in an Ecore model in order to collapse the hierarchy: the ModGraph rule (Fig. 4), the generated Xcore implementation (Listing 1.5), and the Xcore generated Java code (Listing 1.6).

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 1.5 could be written more concisely if written by hand. In fact, lines 5–14 could be expressed by only three lines of handwritten 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).
Listing 1.6. Xcore generated Java code for method removeSub

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

This code generation approach supports the most general case, in which matching has to be performed in (potentially nested) loops (see Listing 1.3 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 1.5 to Java code is shown in Listing 1.6. 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 Fig. 4 and the Xcore implementation in Listing 1.5, line 4. The Java implementation uses lines 7–19 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 1.5, lines 10–11 to filter the references typed over the subclass is mapped to lines 38–42 in Listing 1.6. An additional Java filter function is called to map the procedural expression to Java. The foreach-expression shown in Listing 1.5, line 18, even forces a reimplementation to be mapped to Java, see lines 52–58.

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 (Fig. 1).

The second case in point is to consider the work presented here as part of EMF as a whole and towards total model driven software engineering with it. Xcore and ModGraph interact well with Ecore, EMF’s metamodel for structural modeling, because of its full integration. Even migration is not a problem, as Ecore may be converted to Xcore. Hence there are three ways to use Xcore and ModGraph eventually together with Ecore:

**The new way:** Starting with an Xcore model, marked with a solid arrow denoted with ‘Start’ (red) in Figure 5, the modeler may define the structural model and simple or procedural operations directly. Complex operations may be specified by ModGraph rules, taking the added value of graph transformation rules into account. Doing so, the modeler marks an operation in the Xcore file and selects ‘Implement as ModGraph graph transformation rule’ from the pop-up menu shown in Figure 6, depicted by the bold arrow ‘implement’ in Fig. 5. This arrow is also marked with ‘transform’, because of the already described model-to-model transformation from ModGraph to Xcore. The code ModGraph generates for rules implementing Xcore operations is injected right into the Xcore model. Hence, ModGraph operations may call and be called by Xcore operations.

**Up to the new:** Migrating an existing Ecore model, the modeler may convert the model into an Xcore model via the generator model marked with ‘convert’ in the grey part

---

The new way: Starting with an Xcore model, marked with a solid arrow denoted with ‘Start’ (red) in Figure 5, the modeler may define the structural model and simple or procedural operations directly. Complex operations may be specified by ModGraph rules, taking the added value of graph transformation rules into account. Doing so, the modeler marks an operation in the Xcore file and selects ‘Implement as ModGraph graph transformation rule’ from the pop-up menu shown in Figure 6, depicted by the bold arrow ‘implement’ in Fig. 5. This arrow is also marked with ‘transform’, because of the already described model-to-model transformation from ModGraph to Xcore. The code ModGraph generates for rules implementing Xcore operations is injected right into the Xcore model. Hence, ModGraph operations may call and be called by Xcore operations.

**Up to the new:** Migrating an existing Ecore model, the modeler may convert the model into an Xcore model via the generator model marked with ‘convert’ in the grey part.

---

There is also a fourth way, which we call “classical way”, using ModGraph only with Ecore, which is described in detail in [18]. It is the pre-Xcore version of ModGraph and leaves all control flow elements to be coded in Java. As it is not Xcore related, it is not explained here.
of Figure 5. As this is a model-transformation all ModGraph rules can be reused without any problems. Procedural and simple operations can be modeled in Xcore now. Finally the code generated from the ModGraph rules is injected into the Xcore model.

**Back to the roots:** Using the generated Ecore model to implement operations with ModGraph rules also starts with an Xcore model. One may model the structure and some operations in Xcore. The Xcore code generator generates an Ecore model within code generation. This model may be used to implement operations as ModGraph rules.

Concluding we provide a very flexible and seamlessly integrated approach: you may start with Ecore or Xcore and convert one into another taking the specified ModGraph rules with you in order to get an executable model. Within this approach we follow our constant goal to improve EMF’s modeling languages and tools, without reinventing the wheel, by the added value of graph transformations.

## 6 Related Work

In [20] 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
Table 1. Graph transformation languages and tools

<table>
<thead>
<tr>
<th>language/tool</th>
<th>interpreter</th>
<th>compiler</th>
<th>target language(s) (if compiled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eMOFLON [2]</td>
<td>x</td>
<td>x</td>
<td>Java</td>
</tr>
<tr>
<td>Fujaba [13]</td>
<td>x</td>
<td>x</td>
<td>Java</td>
</tr>
<tr>
<td>GReAT [1]</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GrGen.NET [12]</td>
<td>-</td>
<td>x</td>
<td>C#</td>
</tr>
<tr>
<td>Henshin [3]</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MDELab [11]</td>
<td>x</td>
<td>-</td>
<td>Xcore or Java</td>
</tr>
<tr>
<td>ModGraph</td>
<td>(x)</td>
<td>x</td>
<td>Xcore or Java</td>
</tr>
<tr>
<td>PROGRES [15]</td>
<td>x</td>
<td>x</td>
<td>C or Java</td>
</tr>
<tr>
<td>VIATRA2 [17]</td>
<td>-</td>
<td>x</td>
<td>Java</td>
</tr>
</tbody>
</table>

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 [20]. 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 interable 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 Fig. 1.

---

8 Please note, that PROGRES is not EMF related, but needs to be mentioned as godfather of practically oriented graph transformation tools.
7 Conclusion

We have presented a new approach of compiling high level graph transformation rules into a procedural language for behavioral modeling (Xcore). Summing up, Xcore and ModGraph offer the following general benefits:

1. Complex transformations which are awkward to program in Xcore may be specified with ModGraph’s high-level graph transformation rules.
2. Graph transformation rules may be composed with control structures provided by Xcore.
3. Simple operations may be encoded exclusively in Xcore.
4. Complete application code may be generated by relying on the code generators of EMF, Xcore, and ModGraph.
5. Re-targeting the ModGraph code generator to Xcore gains platform independence for ModGraph.

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


