Incremental Bidirectional Transformations: Evaluating Declarative and Imperative Approaches using the AST2Dag Benchmark

Matthias Bank, Sebastian Kaske, Thomas Buchmann and Bernhard Westfechtel
Chair of Applied Computer Science I, University of Bayreuth, Universitätsstrasse 30, 95440 Bayreuth, Germany

Keywords: Model-Driven Development, Model Transformations, Bidirectional Transformations, Incremental Transformations, BX, Benchmark, Evaluation, QVT-R, BXtend.

Abstract: Model transformation are the core of model-driven software engineering. Typically an initial model is refined throughout the development process using model transformations to derive subsequent models until eventually code is generated. In round-trip engineering processes, these model transformations are performed not only in forward, but also in backward direction. To this end, bidirectional transformation languages provide a single transformation definition for both directions. This paper evaluates the transformation languages QVT Relations (QVT-R) which allows to specify incremental bidirectional transformations declaratively at a high level of abstraction and BXtend - a framework for procedural specification of both forward and backward transformation in a single rule set. Both languages have been used to implement the AST2Dag transformation example. The benchmarx framework was used for a quantitative and qualitative evaluation of the obtained results.

1 INTRODUCTION

Model-Driven Software Engineering (MDSE) (Schmidt, 2006) aims at increasing the productivity of software engineers by replacing low-level program code with high-level models. Typically, *modeling languages* are defined with the help of *metamodels*, e.g., the *Meta Object Facility* standard *MOF* (Object Management Group, 2016b) provided by the Object Management Group (OMG). A subset of MOF called Essential MOF (EMOF) is available, which has been implemented in the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009). In the context of *model transformations*, a vast variety of languages and tools has been developed (Czarnecki and Helsen, 2006). They differ with respect to the underlying computational paradigm (procedural, functional, rule-based, object-oriented), application and scheduling strategies, directionality (uni- vs. bidirectional), support for in-place or out-place, batch or incremental transformations, etc.

This paper focuses on *bidirectional transformations*. It compares two languages that have been designed for bidirectional transformations: *QVT-R* (Object Management Group, 2016a), a *declarative language* in which transformations are defined by relations between source and target model elements, and *BXtend* (Buchmann, 2018), an *object-oriented framework* for bidirectional transformations in which transformations are programmed in Xtend. In QVT-R, it is possible to define a bidirectional transformation by a single set of relations, while both transformation directions have to be specified separately in BXtend. The comparison is performed with the help of a *benchmark* which we implemented on top of the Benchmarx framework for evaluating bidirectional transformations (Anjorin et al., 2017b; Anjorin et al., 2019). The evaluation is performed with respect to the following properties: size of the transformation definitions, functionality (measured in terms of passed test cases), and performance (run time).

For our comparison, we selected the *AST2Dag* transformation case that was described first in (Westfechtel, 2018) and was added later to the bx wiki for bidirectional transformations [1]. An *abstract syntax tree* represents an arithmetic expression as a tree, while a *directed acyclic graph* representation of the same expression includes each subexpression only once. Even though both models are equivalent representations of the same information, the AST2Dag case proved to be challenging for bidirectional model transformation languages and tools.

[1] http://bx-community.wikidot.com

Bank, M., Kaske, S., Buchmann, T. and Westfechtel, B.
Incremental Bidirectional Transformations: Evaluating Declarative and Imperative Approaches using the AST2Dag Benchmark.
DOI: 10.5220/0009206602490260
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 249-260
ISBN: 978-989-758-421-3

2 BIDIRECTIONAL TRANSFORMATIONS

2.1 Overview

A broad range of different languages and tools addressing model transformations has been developed in the past decade. They differ with respect to computational paradigm (rule-based, functional, object-oriented, procedural), directionality (unidirectional vs. bidirectional), support for in-place, out-place, incremental or batch transformations, etc. Numerous model transformation languages emerged accordingly, ranging from general ones, like ATL (Jouault et al., 2008), graph-based languages like Henshin (Arendt et al., 2010) or eMoflon (Anjorin et al., 2011) to domain-specific languages such as, e.g., the Epsilon family of transformation languages (Rose et al., 2014) or EMG (Popoola et al., 2016).

The vast majority of transformation languages supports only unidirectional batch transformations with accompanying fairly well developed tool support. However, bidirectional transformations are required in a variety of use cases of industrial relevance. For example, *round-trip engineering* integrates forward and reverse engineering into a coherent process which needs to be supported by bidirectional transformation tools: After transforming a source model into a target model, additions or modifications of the target model may be necessary. Consequently, changes to the source model need to be propagated in a way which allows to retain the modifications of the target model. These change propagations call for *incremental* rather than batch transformations. Additionally, changes to the target model may have to be propagated back to the source model, resulting in *bidirectional* transformations. Furthermore, integration of heterogeneous tools calls for *data converters* which ideally perform lossless transformations in all directions.

In the light of this diversity, the OMG issued the *QVT* standard (*Queries, Views, and Transformations*) which defines a family of model transformation languages located at different levels of abstraction (Object Management Group, 2016a). The most high-level language defined in the QVT standard is *QVT Relations* (*QVT-R*). QVT-R provides a language for the declarative specification of transformations between MOF-based models. Furthermore, QVT-R reuses the *Object Constraint Language* (OCL), an expression language for MOF models (Object Management Group, 2014). QVT-R addresses a wide spectrum of model transformation scenarios, including enforcing and checking, unidirectional and bidirectional, batch and incremental, and n:1 transformations. However, bidirectionality is not a unique feature of QVT-R (Czarnecki et al., 2009). The most prominent competitors are languages and tools based on *triple graph grammars*, which were introduced much earlier (Schürr, 1994) and have been developed further in numerous research contributions (Königs and Schürr, 2006; Becker et al., 2007). In contrast to the grammar-based approach, QVT-R follows a relational paradigm where a transformation is specified by a set of relations among patterns to be instantiated in the participating models.

2.2 Languages and Tools

In the following, we describe the languages and tools used for our AST2Dag benchmark.

QVT-R. While QVT-R provides many interesting and promising features, the language has not been adopted widely. This stands in sharp contrast to other OMG standards such as MOF and OCL. To the best of our knowledge, currently no tool is available that fully conforms to the standard defined by the OMG. For the benchmark, we used *medini QVT*[2], which runs only under old Eclipse versions, but still can be downloaded from the cited web site. The tool QVT Declarative[3] is under development, but supports the QVT-R standard only partially so far.

Medini QVT conforms to the QVT-R standard syntactically, but implements a deviating semantics. In medini QVT, a *persistent trace* is stored which records relations between source and target elements. The trace is used to support incremental transformations. When source elements have been inserted, they are transformed because they do not appear in the trace so far. When source elements have been deleted, deletion is propagated to target elements via the trace elements. Finally, certain changes to source elements (modifications of attribute values) may be propagated to target elements such that consistency is restored.

BXtend. The BXtend (Buchmann, 2018) framework was created as a result to a case study in round-trip engineering between UML class models and Java source code. Approaches using TGGs (Buchmann and Westfechtel, 2016) and QVT-R (Greiner and Buchmann, 2016) clearly revealed the drawbacks of highly declarative approaches, when right-hand sides of rules contain variability in creation patterns (e.g.,

[2]http://projects.ikv.de/qvt

[3]https://projects.eclipse.org/projects/modeling.mmt.qvtd

it is not possible to specify rules between abstract superclasses and create the corresponding subtypes during rule execution). This leads to massive copying and pasting similar rules. As a consequence BXtend uses an object-oriented approach, where the framework infrastructure comprising abstract rules, generic creation of source and target elements and handling the correspondence model is generated automatically. The transformation developer only needs to specify transformation patterns for forward and backward directions respectively using the Xtend programming language[4], a fully Java compatible, less verbose language which comes with high-level constructs such as powerful lambda expressions.

3 THE TRANSFORMATION CASE: EXPRESSION TREES AND DAGS

3.1 Description

We study a problem from compiler construction: Given an *abstract syntax tree* of an *expression*, a *directed acyclic graph* (*dag*) is constructed which contains common subexpressions only once (Figure 1). From the dag created by this *folding* transformation, the original tree may be reconstructed by *unfolding* the dag. In contrast to a case study performed earlier (Westfechtel and Buchmann,), where both source and target model contained different information, the same information is represented in different ways here. A pair of a tree and a DAG is consistent, if traversing both the tree and the DAG in the same order (e.g., inorder) results in the same expression represented by the data structure.

Figure 2 displays the metamodels for trees and dags. In the tree metamodel, left and right are containment references from operators to left and right operands, respectively. In contrast, these references do not own the containment property in the dag metamodel because an expression may occur at multiple locations. Thus, in the dag representation of an expression all nodes are immediate components of the root object of the dag model, and the hierarchy of expressions is represented by non-containment references.

This transformation case is highly problematic for rule-based and grammar based approaches and potentially highlights current limitations. While the tree is best constructed in a *top-down* way by unfolding the DAG, the DAG is best constructed *bottom-up* by folding the tree.

[4]https://www.eclipse.org/xtend/

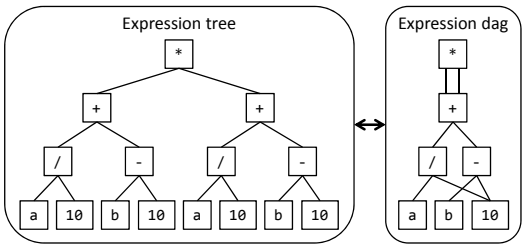

Figure 1: Tree and dag for (a/10 + (b-10))*(a/10 + (b-10)).

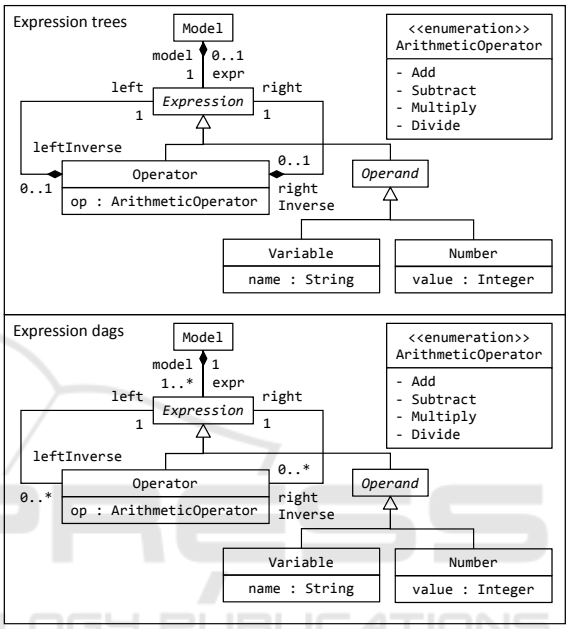

Figure 2: Metamodels for Expression Trees and Dags.

Let us briefly summarize the formal properties of forward and backward transformations: Trees and dags are equivalent representations of expressions. Both the forward and the backward transformation are total, deterministic, injective, and surjective.

3.2 Benchmarx

In order to evaluate the solutions provided in this paper, we use the benchmarx framework (Anjorin et al., 2017b; Anjorin et al., 2019). It is designed as a generic framework for benchmarking bx tools and uses JUnit test cases designed as synchronization dialogs. Such a synchronization dialog always starts from the same agreed upon initial consistent state, and then applies a sequence of edits to the source and target models. After the edits, the transformation is executed and only the resulting models are directly asserted by comparing them with predefined expected versions. In this paper, we provide a benchmark for a

specific bx problem (Ast2Dag) by providing a suite of test cases which are used to evaluate the solutions to the transformation problem specified in QVT-R and BXtend. The test suite contains test cases for both forward and backward direction as well as batch and incremental tests for each direction. Batch test cases always start with an empty model and an edit delta which creates the starting state of the source model of the transformation (AST model for forward direction, DAG model for the backward direction). After the model state is established, the transformation is started and the obtained result is compared with a predefined expected state. Incremental test cases on the other hand start with an initial consistent state of both models. Afterwards a series of edit deltas is applied to one of those models and the transformation is rerun. The test result is obtained by comparing the actual state of the models with the expected one. Please note that the incremental test cases also check if the propagated change is minimal.

3.3 Test Cases

Batch Forward. The batch forward test cases comprise the following scenarios:

Initialization: An empty model with the root element Model is created.

Single Expression: This test case creates an expression with a single number in an empty AST. The expected result is an expression DAG with a corresponding single number.

Complex Expression: Extension of the previous test case: a complex expression tree is created, which is transformed to a corresponding DAG.

Multiple Expressions: An expression tree with multiple expressions is created. Expressions appear more than one time. The corresponding DAG should contain multiple references to corresponding expression elements.

Multiple Complex Expressions: Analogously to the previous test case multiple expressions over several hierarchy layers which now have identical subtrees are created. The corresponding DAG should contain multiple references to corresponding elements.

Batch Backward. Analogously to the forward test cases, the batch backward test cases address the following scenarios:

Single Expression: An expression with a single number in an empty DAG is created. The expected result is a single number expression in the corresponding AST.

Multiple Expressions: Multiple expressions are created in the DAG, its references are single valued. The expected result are multiple expressions in the corresponding AST.

Multiple Complex Expressions: Multiple expressions in the DAG which have multi-valued references. The corresponding unfolding operation should create multiple occurrences of the expressions in the AST.

Multiple Expressions with Identical Subtrees: Similar to the previous test case, but containing expressions that have identical subtrees.

Incremental Forward. The incremental test cases start with a consistent pair of AST and DAG models. Afterwards, the AST is modified and the changes need to be propagated to the DAG.

Incremental Inserts: Leaves are replaced with a complex expression in an existing AST (which has been transformed to a corresponding DAG in a previous step). The expected result is a corresponding DAG.

Incremental Deletions: Nodes from an existing AST are removed and replaced with a number. Please note, that inner nodes are removed, as removing only leaves would result in an invalid AST. Children of deleted nodes are removed recursively. The expected result is an accordingly updated DAG.

Incremental Modifications: An existing AST is modified. A modification is either changing an attribute of a node or switching between variable and number. The expected result is a corresponding DAG.

Incremental Modifications Resulting in Deletions: An existing AST is modified. The modification leads to merging of elements in the DAG.

Stability: This test re-runs the transformation after an idle source delta has been applied. The expected result is an unchanged target DAG.

Incremental Backward. Incremental backward test cases also start with a consistent pair of DAG and AST models. After a change to the DAG, corresponding updates are propagated to the AST.

Incremental Inserts: New nodes are inserted into the DAG. Changes are propagated to the AST.

Incremental Deletions: Nodes are deleted from the DAG. The resulting AST must be valid, i.e. Children of affected AST nodes need to be deleted recursively.

Incremental Modificactions: An existing DAG is modified. A modification comprises changes of node values and switching the type of a node (variable and number). The corresponding AST needs to be updated accordingly.

Stability: The stability of the transformation is tested. An idle delta to the DAG is propagated to the AST, resulting in an unchanged AST.

4 SOLUTIONS

4.1 QVT-R

We failed in the construction of a single bidirectional transformation. Instead, we developed separate unidirectional transformations operating *bottom-up* and *top-down*, respectively. The solution was described in (Westfechtel, 2018) and is summarized briefly below to make this paper self-contained. Please note the separate specification of both transformation directions renders round-trip engineering impossible because forward and backward transformation employ different trace models.

The forward transformation employs *backward chaining* of relations through relation calls in **when** clauses (Listing 1). The relations Variable2Variable and Number2Number collapse multiple occurrences of the same atomic operand. Collapsing of multiple occurrences works recursively since it also applies to operators. The *key declarations* in lines 2–4 ensure that only one element per key value is created in the dag model.

In contrast, the backward transformation makes use of *forward chaining* through relation calls in **where** clauses (Listing 2). The specification contains three top-level relations which handle three cases of expression root nodes (variable, number, and operator). Starting from the root, the expression tree is built top-down. For each occurrence of a subexpression, exactly one relation call is executed. For each type of nested operand, there are two relations transforming it as a left or right operand of the enclosing operator, respectively.

4.2 BXtend

The BXtend solution to our example scenario consists of four rule classes, one for each concrete class in the respective source and target models (Model, Number, Operator, and Variable). Within those rule classes bodies for the automatically generated method stubs for sourceToTarget() and targetToSource() are supplied by the user, specifying transformation patterns for each execution direction separately. Since the transformation problem describes folding/unfolding the respective data structures, a n:1 correspondence model is required. Since the correspondence model is shared between both transformation directions, round-trip engineering may be supported (in contrast to the solution in medini QVT).

Since the transformation developer has full control over rule orchestration, we specified a bottom-up traversal of the AST for the forward direction and a top-down creation of the AST for the backward direction. Listing 3 depicts a cutout of the rule class responsible for handling numbers. The method sourceToTarget() (lines 3-25) is called from the framework whenever a forward transformation is executed. We iterate over all instances of Number in the model and check if there is already a correspondence model element available (line 6). If no correspondence model element is found the method addToTargetElem is called. Since a folding of multiple AST elements with same values to a single DAG element is performed in this transformation direction, the method findTargetElem is called to retrieve matching DAG elements. If no matching element is found, a new target model element dag.Number is created (line 30). Otherwise the current source model element is added to the list of matching elements (line 33). The incremental behavior of the transformation is realized in lines 11-18. A check is performed, if all source correspondences which contain the current source model element contain the same value attribute (line 11-12). If the current source value was updated, this check fails and the transformation proceeds in line 19, where the source element is removed from the correspondence list and addToTargetElem is called to establish a correct matching.

Listing 4 depicts the method targetToSource from the rule class responsible for transforming Operators. Since the corresponding AST is constructed in a top-down way, an ordering of the elements is required (lines 4-9). An Xtend switch statement is used to map the operator types of both models. Since variables and numbers may only occur at the leaves of the AST, the hierarchy of operators is established by traversing the left and right references of the current element (lines 33-47) and creating or updating corresponding AST elements.

Listing 1: Transformation of trees to dags.

```
transformation ast2dag(astModel : ast, dagModel : dag) {
    key dag::Variable {model, name};
    key dag::Number {model, value};
    key dag::Operator {model, op, left, right};
    top relation Model2Model {...}
    relation Expression2Expression {
        checkonly domain astModel ast_expression : ast::Expression {};
        enforce domain dagModel dag_expression : dag::Expression {}; }
    top relation Variable2Variable {
    name : String; ast : ast::Model; dag : dag::Model;
    checkonly domain astModel ast_variable : ast::Variable { name = name };
    enforce domain dagModel dag_variable : dag::Variable {
      model = dag,
      name = name };
    when { ast = root(ast_variable).model;
      Model2Model(ast, dag); }
    where {
      if dag_variable.oclIsUndefined() then false
      else Expression2Expression(ast_variable, dag_variable)
      endif; }
    }
    top relation Number2Number {...}
    top relation Operator2Operator {
        ast_op : ast::ArithmeticOperator; ast : ast::Model; dag : dag::Model;
    checkonly domain astModel ast_operator : ast::Operator {
      op = ast_op, left = ast_left : ast::Expression {},
      right = ast_right : ast::Expression {}
    };
    enforce domain dagModel dag_operator : dag::Operator {
      model = dag, op = correspondingDagOperator(ast_op),
      left = dag_left : dag::Expression{},
      right = dag_right : dag::Expression{}
    };
    when { ast = root(ast_operator).model;
      Model2Model(ast, dag);
      Expression2Expression(ast_left, dag_left);
      Expression2Expression(ast_right, dag_right); }
    where { if dag_operator.oclIsUndefined() then false
      else Expression2Expression(ast_operator, dag_operator)
      endif; }
    }
    ...
```

5 EVALUATION

5.1 Size of the Transformation Specification

Both solutions presented in this paper support a textual concrete syntax with which transformation definitions can be specified. Thus, a quantitative impression of the size of the transformation definitions can be obtained by counting the number of *lines of code* (excluding empty lines and comments), the *number of words* (character strings separated by whitespace) in these lines, and the *number of characters* in these words. The values obtained for these metrics are depicted in Table 1. In general, the solution in QVT-R is expected to be shorter for three reasons: (1) It is possible to define both transformation directions simultaneously by a single relational specification. (2) The specification is declarative, at a high level of abstraction. (3) Incremental consistency restoration need (and can) not be specified at all. In the AST2Dag case, the first reason does not hold because even in QVT-R both transformation directions have to be defined separately. Even then, we expect savings for the other reasons. In contrast, the size metrics for the solutions in QVT-R and BXtend are in the same order of magnitude. Thus, BXtend's procedural approach does not

Listing 2: Transformation of dags to trees.

```
transformation dag2ast(dagModel : dag, astModel : ast) {
    top relation RootVariable2Variable {...}
    relation LeftVariable2Variable {
        name : String;
    checkonly domain dagModel dag_operator : dag::Operator {
      left = dag_variable : dag::Variable {
        name = name }};
    enforce domain astModel ast_operator : ast::Operator {
      left = ast_variable : ast::Variable {
        name = name }};
    }
    relation RightVariable2Variable {...}
    top relation RootNumber2Number {...}
    relation LeftNumber2Number {...}
    relation RightNumber2Number {...}
    top relation RootOperator2Operator {
        dag_op : dag::ArithmeticOperator;
    ast_op : ast::ArithmeticOperator;
    checkonly domain dagModel dag : dag::Model {
      exprs = dag_operator : dag::Operator {
        op = dag_op } };
    enforce domain astModel ast : ast::Model {
      expr = ast_operator : ast::Operator {
        op = ast_op } };
    when {  dag_operator.leftInverse->isEmpty();
      dag_operator.rightInverse->isEmpty(); }
    where { ast_op = correspondingAstOperator(dag_op);
      LeftVariable2Variable(dag_operator, ast_operator);
      LeftNumber2Number(dag_operator, ast_operator);
      LeftOperator2Operator(dag_operator, ast_operator);
      RightVariable2Variable(dag_operator, ast_operator);
      RightNumber2Number(dag_operator, ast_operator);
      RightOperator2Operator(dag_operator, ast_operator); }
    relation LeftOperator2Operator {...}
    relation RightOperator2Operator {...}
}
```

increase the size of the transformation definition.

Table 1: Size of the Transformation Definitions of Both Solutions.

	medini QVT	BXtend
Lines of code	353	282
Number of words	702	832
Number of characters	7576	9974

5.2 Qualitative Analysis

In order to perform a qualitative analysis, test cases for the different transformation directions have been specified and executed for both batch and incremental mode of operation. We assume a test case to be passed, if the resulting model matches a predefined expected model state. The BXtend solution is able to pass all tests specified in Section 3.3. The QVT-R solution passes batch forward and backward tests and also incremental backward tests, but it fails in two incremental forward tests: *incremental modifications* and *incremental modifications resulting in deletions*.

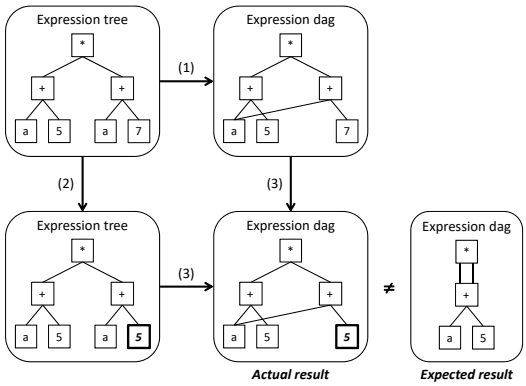

Figure 3: Test Case failing in Medini QVT.

The failures are due to the fact that the incremental

Listing 3: Transformation of trees to dags.

```
class Number2Number extends Elem2Elem {

  override def sourceToTarget() {
    sourceModel.allContents.filter(typeof(Number))
      .forEach[n |
        val corr = n.getCorrModelElem  as MultiElem
        if(corr == null) {
          n.addToTargetElem
        } else {
          val t = corr.targetElement as dag.Number
          if(corr.sourceElements.forall[it instanceof Number &&
            (it as Number).value == n.value]) {
            val newTarget = n.findTargetElem
            if(newTarget != null)
              (newTarget.getCorrModelElem as MultiElem).sourceElements += n
            else
              t.value = n.value
          }
          if(t.value != n.value) {
            corr.sourceElements -= n
            n.addToTargetElem
          }
        }
      ]
  }

  def private addToTargetElem(ast.Number e) {
    var newTarget = e.findTargetElem
    if(newTarget == null) {
      newTarget = createTargetElement(dag.DagPackage.eINSTANCE.number) as dag.Number
    }
    val newCorr = newTarget.getOrCreateCorrModelElement(ruleID) as MultiElem
    newCorr.sourceElements += e
    newTarget.value = e.value
    newTarget.model = e.model.getCorrModelElem.targetElement as dag.Model
    elementsToCorr.put(newCorr)
  }

  def private findTargetElem(ast.Number e) {
    (e.model.getCorrModelElem.targetElement as dag.Model).exprs
      .filter(typeof(dag.Number)).findFirst[it.value == e.value]
  }

  ...
}
```

behavior built into medini QVT deviates from the required behavior. One of the failing test cases is illustrated in Figure 3. In an initial batch transformation (1), an expression tree for (a+5)*(a+7) is transformed successfully into a dag, in which variable a is shared. In step (2), the number 7 is replaced with the number 5. Since both operands of the multiply operator are the same, the incremental transformation (3) should result in the dag displayed at the right-hand side. In contrast, medini QVT merely propagates the modification of the attribute value, without checking for common subexpressions. The second test case fails for the same reason, but even results in a dag which represents a different expression than the tree. These failures are symptoms of a general problem with the design of QVT-R (note that this is not a problem specific to medini QVT): In QVT-R, relations between source and target elements are specified declaratively; incremental behavior is provided "for free". However, the provided behavior may deviate from the required behavior. In contrast, in BXtend incremental behavior has to be specified explicitly, but can tailored to specific requirements. For example, when a number is changed, it is checked whether the same number is already present in the dag. In that case, the dag is reorganized such that the number is present only once; this reorganization is applied recursively updwards in the dag.

5.3 Performance Analysis

In order to evaluate the efficiency and scalability of the resulting transformation with respect to increasing model size, two experiments were conducted in both forward and backward directions resulting in four sets of measurements: (1) batch transformations in forward and backward directions, and (2) incremental transformations in forward and backward directions. The batch transformations test how the solutions scale when creating the corresponding opposite model of increasing size, starting with an AST with only one single expression up to a tree with a depth of 20 hierarchy layers. The incremental transformations apply the exact same edit to master models of increasing size. After correspondence has been established, one single value of a leaf is changed. The time required to locate and propagate this change to the dependent model is measured. The tests were performed on the

Listing 4: Transformation of dags to trees.

```
class Operator2Operator extends Elem2Elem {
  override def targetToSource() {
    var List<dag.Operator> preOrder = new ArrayList<dag.Operator>();
    for (var it = targetModel.allContents.filter(typeof(dag.Operator)); it.hasNext(); ) {
      val dag.Operator op = it.next();
      if (op.leftInverse.empty && op.rightInverse.empty) {
        preOrder += op;
      }
    }
    if (preOrder.size == 0) {
      return;
    } else if (preOrder.size > 1) {
      throw new AssertionError("Dag has multiple root elements.");
    }

    var List<Operator> preOrderSrc = new ArrayList<Operator>();
    val corrRoot = preOrder.get(0).getOrCreateCorrModelElement(ruleID) as MultiElem;
    val srcRoot = corrRoot.getOrCreateSourceElem(sourcePackage.operator, [true]) as Operator;
    preOrderSrc.add(srcRoot);
    srcRoot.model = preOrder.get(0).model.corrModelElem.sourceElement as Model;

    while (!preOrder.empty) {
      val dag.Operator current = preOrder.remove(0);
      val Operator currentSrc = preOrderSrc.remove(0);
      switch (current.op) {
        case ADD: currentSrc.op = ast.ArithmeticOperator.ADD
        case DIVIDE: currentSrc.op =  ast.ArithmeticOperator.DIVIDE
        case MULTIPLY: currentSrc.op = ast.ArithmeticOperator.MULTIPLY
        case SUBTRACT: currentSrc.op = ast.ArithmeticOperator.SUBTRACT
      }
      if (current.right instanceof dag.Operator) {
        preOrder.add(0, current.right as dag.Operator);
        val corrRight = preOrder.get(0).getOrCreateCorrModelElement(ruleID) as MultiElem;
        val srcRight = corrRight.getOrCreateSourceElem(
            sourcePackage.operator, [e | (e as Operator).rightInverse == currentSrc]) as Operator;
        preOrderSrc.add(0, srcRight);
        currentSrc.right = srcRight;
      }
      if (current.left instanceof dag.Operator) {
        preOrder.add(0, current.left as dag.Operator);
        val corrLeft = preOrder.get(0).getOrCreateCorrModelElement(ruleID) as MultiElem;
        val srcLeft = corrLeft.getOrCreateSourceElem(
            sourcePackage.operator, [e | (e as Operator).leftInverse == currentSrc]) as Operator;
        preOrderSrc.add(0, srcLeft);
        currentSrc.left = srcLeft;
      }
    }
  }
  ...
}
```

same machine and in isolation for each solution. A standard machine with an Intel Core i7-4770 CPU was used, running at 3.40 GHz, with 16 GB of DDR3 RAM and with Microsoft Windows 10 64-bit as operating system. We used Java 1.8.0_191, Eclipse Neon (4.6.3), and EMF version 2.12.0 to compile and execute the Java code for the scalability test suite. Each test was repeated 5 times and the median measured time was computed. The four measurement results are depicted in Fig. 4, 5, 6, and 7. A figure is composed of two plots – a plot with linear/linear scale to the left, and a plot with log/log scale to the right. While the linear plot provides a realistic impression for the actual complexity of each solution, the logarithmic one zooms into finer details for smaller models and zooms out for larger models, allowing to qualitatively present large differences in runtime. The BXtend solution outperforms the Medini QVT solution in every benchmark. The gap between both solutions becomes clearly evident in the Figures showing linear/-linear scale. Medini QVT shows sharp increases in runtime kicking in already at relatively small models of around 10k elements, while the BXtend solution is able to handle signifcantly larger models.

6 RELATED WORK

While a broad range of languages and tools exist for model transformations in general and bidirectional transformations in particular, thorough evaluations of different approaches applied to various transformation problems are still missing in the literature. The transformation tool contest (TTC)[5], aims at bringing together tool developers in a competition-like annual event. Several predefined transformation problems are given which have to be solved by the participants and their respective tools. However, to the best of our knowledge so far only one case for bidirectional transformations has been posed and solved, so far (see be-

[5]https://www.transformation-tool-contest.eu/

Figure 4: Forward Batch Transformation: Linear/linear Scale (Left) and Log/log Scale (Right).

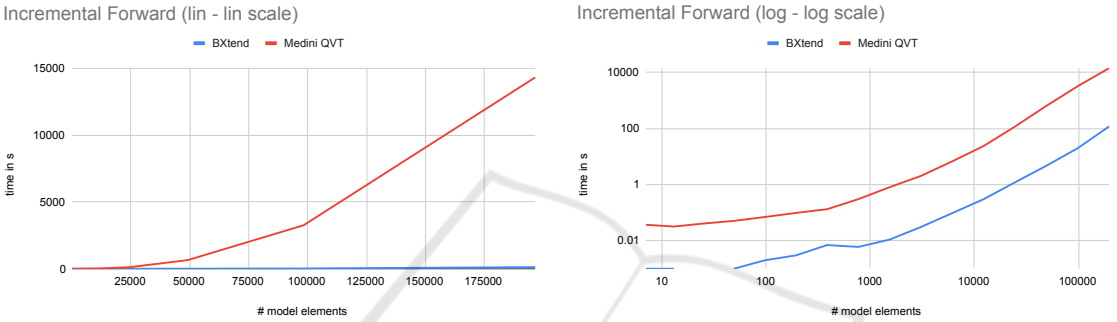

Figure 5: Forward Incremental Transformation: Linear/linear Scale (Left) and Log/log (Right).

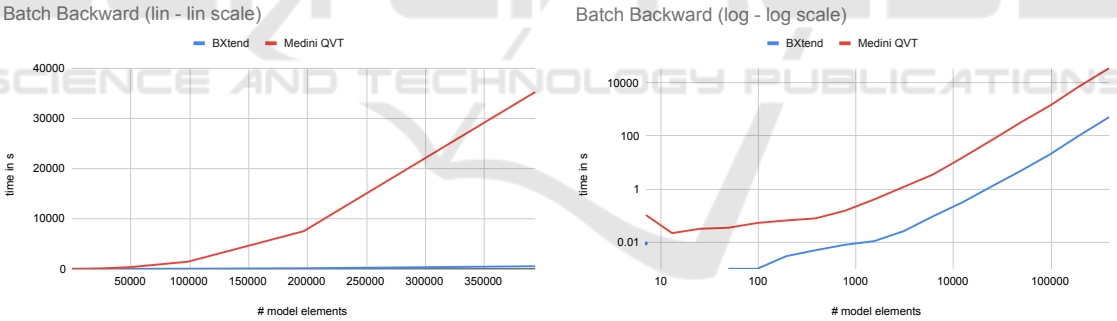

Figure 6: Backward Batch Transformation: Linear/linear Scale (Left) and Log/log Scale (Right).

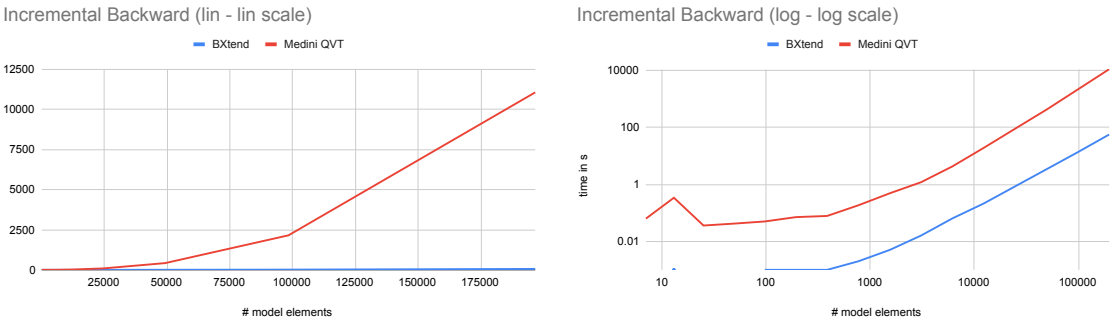

Figure 7: Backward Incremental Transformation: Linear/linear Scale (Left) and Log/log Scale (Right).

low).

The benchmarx (Anjorin et al., 2017b) framework is an important step towards providing a unified platform for assessing and benchmarking incremental model transformations. For the transformation tool contest 2017, a bidirectional transformation problem has been accepted, which addresses the transformation of family and person databases (Anjorin et al., 2017a). A wide range of solutions was submitted, which allowed for a comparison of seven significantly differing tools (Anjorin et al., 2019). An additional, more detailed comparison of BXtend and QVT-R applied to the Families to Persons case is described in (Westfechtel and Buchmann, 2018).

The transformation problem discussed in this paper differs significantly from the Families to Persons case. While the latter involves information loss and asymmetric specifications of both transformation directions, the AST2Dag benchmark is concerned with a symmetric transformation problem without any loss of information. Both the AST and the Dag represent the same information in different ways. Transformations in both directions require restructuring (folding in forward direction and unfolding in backward direction). Interestingly, currently there is no bidirectional language or tool known that is able to solve this bidirectional transformation problem in a single specification (please recall that our solutions in BXtend and QVT-R both require two unidirectional transformations that are specified separately). Thus, the AST2Dag benchmark constitutes a real challenge for bidirectional languages and tools.

7 CONCLUSION

In this paper we provide an evaluation of declarative and imperative approaches to solve a bidirectional transformation problem. The problem is taken from compiler construction and addresses expressions represented as abstract syntax trees (AST) and directed acyclic graphs (DAG). Incremental and bidirectional transformations between the two data structures represented as EMF models have been created using QVT-R as a representative of declarative approaches and BXtend, an imperative framework specifically designed for incremental bidirectional transformations. The experiments revealed that we failed to specify a single bidirectional transformation with QVT-R. Rather, two unidirectional transformations were required. As a consequence, the quantitative analysis shows that the declarative approach does not provide a benefit in terms of size of the transformation description. The qualitative analysis is done using Benchmarx, a framework for evaluating bx tools. A set of test cases for each transformation direction and for each mode of operation had to be solved with each tool. While the BXtend solution was able to pass all test cases, the QVT-R solution failed in two incremental forward ones. Furthermore, the performance tests revealed that the QVT-R solution shows significant increases in runtime for even smaller models, while the BXtend solution scales much better with increasing model sizes.

As pointed out in the section on related work, the AST2Dag case constitutes a challenge for bidirectional transformation languages and tools. To gain further insights from this case, it would highly desirable to have this case implemented in further bidirectional tools (again using the benchmarx framework).

REFERENCES

Anjorin, A., Buchmann, T., and Westfechtel, B. (2017a). The Families to Persons case. volume 2026 of *CEUR Workshop Proceedings*, pages 27–34, Marburg, Germany.

Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H.-S., Eramo, R., Hinkel, G., Samimi-Dehkordi, L., and Zündorf, A. (2019). Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software and Systems Modeling*.

Anjorin, A., Diskin, Z., Jouault, F., Ko, H., Leblebici, E., and Westfechtel, B. (2017b). Benchmarx reloaded: A practical benchmark framework for bidirectional transformations. In Eramo, R. and Johnson, M., editors, *Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, April 29, 2017.*, volume 1827 of *CEUR Workshop Proceedings*, pages 15–30. CEUR-WS.org.

Anjorin, A., Lauder, M., Patzina, S., and Schürr, A. (2011). Emoflon: leveraging EMF and professional CASE tools. In Heiß, H., Pepper, P., Schlingloff, H., and Schneider, J., editors, *Informatik 2011: Informatik schafft Communities, Beiträge der 41. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 4.-7.10.2011, Berlin, Deutschland (Abstract Proceedings)*, volume 192 of *LNI*, page 281. GI.

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-Verlag.

Becker, S. M., Herold, S., Lohmann, S., and Westfechtel, B. (2007). A graph-based algorithm for consistency

maintenance in incremental and interactive integration tools. *Software and Systems Modeling*, 6(3):287–316.

Buchmann, T. (2018). Bxtend - A framework for (bidirectional) incremental model transformations. In Hammoudi, S., Pires, L. F., and Selic, B., editors, *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pages 336–345. SciTePress.

Buchmann, T. and Westfechtel, B. (2016). Using triple graph grammars to realise incremental round-trip engineering. *IET Software*, 10(6):173–181.

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F. (2009). Bidirectional transformations: A cross-discipline perspective. In Paige, R. F., editor, *Proceedings of the Second International Conference on Theory and Practice of Model Transformations (ICMT 2009)*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283, Zurich, Switzerland. Springer-Verlag.

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

Greiner, S. and Buchmann, T. (2016). Round-trip engineering UML class models and java models: A real-world use case for bidirectional transformations with QVT-R. *International Journal of Information System Modeling and Design (IJISMD)*, 7(3):72–92.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39.

Königs, A. and Schürr, A. (2006). Tool integration with triple graph grammars - a survey. In Heckel, R., editor, *Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Dagstuhl, Germany. Elsevier Science.

Object Management Group (2014). *Object Constraint Language Version 2.4*. Needham, MA, formal/14-02-03 edition.

Object Management Group (2016a). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.3*. Needham, MA, formal/16-06-03 edition.

Object Management Group (2016b). *OMG Meta Object Facility (MOF) Core Specification Version 2.5.1*. Needham, MA, formal/19-10-01 edition.

Popoola, S., Kolovos, D. S., and Rodriguez, H. H. (2016). EMG: A domain-specific transformation language for synthetic model generation. In Gorp, P. V. and Engels, G., editors, *Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings*, volume 9765 of *Lecture Notes in Computer Science*, pages 36–51. Springer.

Rose, L. M., Kolovos, D. S., Paige, R. F., Polack, F. A. C., and Poulding, S. M. (2014). Epsilon flock: a model migration language. *Software and System Modeling*, 13(2):735–755.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31.

Schürr, A. (1994). Specification of graph translators with triple graph grammars. In Mayr, E. W., Schmidt, G., and Tinhofer, G., editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition.

Westfechtel, B. (2018). Case-based exploration of bidirectional transformations in QVT Relations. *Software and Systems Modeling*, 17(3):989–1029.

Westfechtel, B. and Buchmann, T. Incremental bidirectional transformations: Comparing declarative and procedural approaches using the families to persons benchmark. In Damiani, E., Spanoudakis, G., and Maciaszek, L. A., editors, *Evaluation of Novel Approaches to Software Engineering - 13th International Conference, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018, Revised Selected Papers*, pages 98–118.

Westfechtel, B. and Buchmann, T. (2018). Incremental bidirectional transformations: Comparing declarative and procedural approaches using the families to persons benchmark. In Damiani, E., Spanoudakis, G., and Maciaszek, L. A., editors, *Evaluation of Novel Approaches to Software Engineering - 13th International Conference, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018, Revised Selected Papers*, volume 1023 of *Communications in Computer and Information Science*, pages 98–118. Springer.