The Families to Persons Case

Anthony Anjorin
Paderborn University, Germany
anthony.anjorin@upb.de

Thomas Buchmann
Univ. of Bayreuth, Germany
thomas.buchmann@uni-bayreuth.de

Bernhard Westfechtel
Univ. of Bayreuth, Germany
bernhard.westfechtel@uni-bayreuth.de

Abstract

The Families to Persons case is a well-known example problem for bidirectional transformations. This paper proposes an implementation of this case within the recently developed Benchmarx framework [2], based on previous conceptual work [1].

1 Introduction

Bidirectional transformations (bx) are transformations which may be executed from source to target and vice versa. They are required in a variety of scenarios, including bidirectional data converters, round-trip engineering, or view updates [5].

A wide variety of bx languages and tools have been developed which differ with respect to underlying data models, supported scenarios, incremental behaviour, etc. To compare these approaches, the need for benchmarks has been recognized for long [5]. In response to this need, a conceptual framework for bx benchmarks was proposed in [1].

Only recently, however, the conceptual proposal was materialized into an implemented infrastructure called Benchmarx [2]. The tool transformation case described below exploits this framework to provide for an implementation of a well-known bx case dealing with the transformation between heterogeneous databases. One of these maintains a collection of families and their members; its opposite contains a flat collection of persons.

Section 2 describes the Families to Persons case. Section 3 provides a survey of the tool architectures supported by the Benchmarx framework. Section 4 explains the test cases against which the solutions are executed. Section 5 sketches the steps to be performed for implementing the benchmark in a specific tool. Section 6 is devoted to the evaluation of the proposed solutions. Section 7 concludes the paper.

2 Case Description

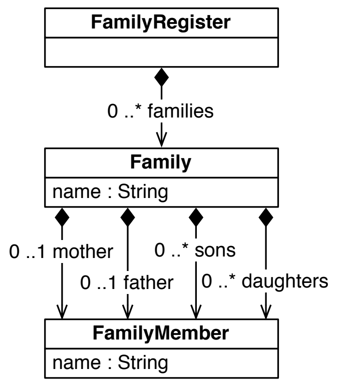
The Families to Persons case is part of the ATL[1] transformation zoo[2] and was created as part of the "Usine Logicielle" project. The original authors are Freddy Allilaire, Frédéric Joault, and Jean Bézivin, all members of the ATLAS research team at the time it was created (December 2006) and published (2007). The variant proposed for the TTC is described below.

We selected this case for several reasons: (1) The underlying data structures are rather simple. Thus, the case can be implemented e.g. in tools operating on trees rather than on graphs. (2) The case is rather small and simple to understand. (3) Furthermore, it may be implemented with acceptable effort. (4) Finally, it demonstrates a number of problems typically occurring in the transformation between heterogeneous data structures (information loss, flattening of hierarchies, different representation of the same information, etc.).

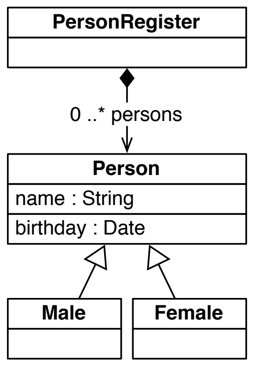
In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

[1]The Atlas Transformation Language: `https://www.eclipse.org/atl/`

[2]Available from `https://www.eclipse.org/atl/atlTransformations/#Families2Persons`



(a) Families metamodel



(b) Persons metamodel

Figure 1: Source and target data structures

Although solutions to the case need not be implemented in model transformation tools and do not have to live in a specific technological space such as EMF, we assume EMF models in the following explanation. Thus, the data structures to be manipulated are defined by Ecore models in Figure 1.

We assume a unique root in each model. A family register stores a collection of families. Each family has members who are distinguished by their roles. The metamodel allows for at most one father and at most one mother as well as an arbitrary number of daughters and sons. A person register maintains a flat collections of persons who are either male or female. Note that key properties may be assumed in neither model: There may be multiple families with the same name, name clashes are even allowed within a single family, and there may be multiple persons with the same name and birthday. Furthermore, all collections are assumed to be unordered.

A families model is consistent with a persons model if a bijective mapping between family members and persons can be established such that (i) mothers and daughters (fathers and sons) are paired with females (males), and (ii) the name of every person p is "$f.name$, $m.name$", where m is the member (in family f) paired with p.

After running a transformation in any direction, it is required that the participating models are consistent according the definition given above. However, this requirement does not suffice to define the functionality of transformations in a unique way. Below, we first consider batch transformations, where the target model is created from scratch.

The functionality of a forward transformation is straightforward: Map each family member to a person of the same gender and compose the person's name from the surname and the first name; the birthday remains unset. The backward transformation is more involved: A person may be mapped either to a parent or a child, and persons may be grouped into families in different ways.

To reduce non-determinism, we introduce two boolean parameters controlling the backward transformation, resulting in a configurable backward transformation: PREFER_CREATING_PARENT_TO_CHILD controls whether a person is mapped to a parent or a child. PREFER_EXISTING_FAMILY_TO_NEW controls whether a person is added to an existing family (if available), or a new family is created along with a single family member. If both parameters are set to true, the second parameter takes precedence: If a family is available with a matching surname, but there is no matching family with an unoccupied parent role, the member is inserted into an existing family as a child. Please note that the configuration parameters do not completely resolve non-determinism in the presence of multiple matching candidate families if preferExistingToNewFamily is set to true.

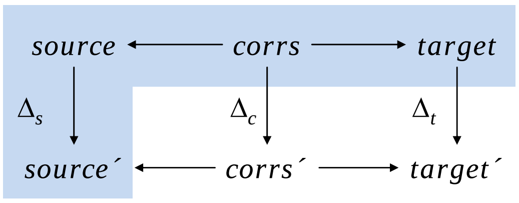
In this scenario, information loss occurs in both directions: In forward direction, the distinction between parents and children is lost as well as the aggregation into families. In backward direction, birthdays are lost. Altogether, this case constitutes an example of a round-trip engineering scenario, where both models may be edited and changes have to be propagated back and forth.

For incremental transformations, updates such as insertions, deletions, changes of attribute values, and move operations have to be considered. In forward direction, insertion of a family has no effect on the target model. Insertion of a member results in insertion of a corresponding person; likewise for deletions. If a family is deleted, all persons corresponding to its members are deleted. If a member is renamed, the corresponding person is renamed accordingly. If a family is renamed, all persons corresponding to family members are renamed. If a member is moved, different cases have to be distinguished. If the gender is retained, the corresponding person object is preserved; otherwise, it is deleted, and a new person object with a different gender is created whose attributes are copied from the old person object. A local move within a family does not affect the corresponding person's name; a move to another family results in a potential update of the person's name.

In backward direction, the effect of an update depends on the values of the configuration parameters, which may be changed dynamically. Please note that the parameter settings must not affect already established correspondences; rather, they apply only to future updates. Deletion of a person propagates to the corresponding family member. If a person is inserted, it depends on the configuration parameters how insertion propagates to the families model (see above). Persons cannot be moved because the persons model consists of a single, flat, and unordered collection. Changes of birthdays do not propagate to the opposite model. If the first name of a person is changed, the first name of the corresponding family member is updated accordingly. Finally, if the surname of a person is changed, this change does not affect the current family and its members: The family preserves its name even if it does not contain other members; thus, the update has no side effects on the existing family. Rather, the corresponding family member is moved to another family, which may have to be created before the move; the precise update behaviour depends on the parameter settings.

3 Tool Architectures

The Benchmarx infrastructure has been designed such that it supports a wide range of tool architectures. To this end, tools are considered as black boxes with respect to the data structures maintained by the tools. As mentioned earlier, the data structures to be synchronized may be represented in a tool-specific way; EMF models merely serve as an example in this paper. Furthermore, the tools may be based on different architectures which are sketched below (see [2] for a more comprehensive description).



$$\begin{tikzcd} source \arrow[d, "\Delta_s"'] & corrs \arrow[l] \arrow[r] \arrow[d, "\Delta_c"'] & target \arrow[d, "\Delta_t"'] \\ source' & corrs' \arrow[l] \arrow[r] & target' \end{tikzcd}$$

Figure 2: Input data for update propagation

The shaded region in Figure 2 comprises the input data used for update propagation. Tools may rely on correspondences between source and target elements (horizontal arrows). Furthermore, they may assume deltas between old and new states (vertical arrows). A delta may be operational (o-delta, a sequence of operations from the old to the new state) or structural (s-delta, defined in terms of differing elements). The output produced by a tool comprises at least a new target state, as well as optional correspondences and deltas.

In [2], we identified seven tool architectures which differ with respect to their input and output data. A batch tool takes the new source and produces a new target from scratch. All other types of tools operate incrementally inasmuch as they update an already existing target. A state-based tool takes the new source and the old target and produces an updated target. A corr-based tool relies on stored correspondences to improve update behaviour; in addition to an updated target, it produces updated correspondences. An s-delta-based tool takes the old source, the old target, and an s-delta between the old and the new source state, and calculates an s-delta between the old and the new target state (and thus implicitly an updated target). An s-delta/corr-based tool additionally relies on stored correspondences, which are updated along with the calculation of the s-delta on the target. Similarly, a distinction is made between o-delta-based and o-delta/corr-based tools.

Selection of a tool architecture constitutes a classical engineering trade-off. On one hand, the preciseness of update propagation grows with the amount of available input data. For example, stored correspondences may be used to improve update propagation when correspondences cannot be inferred from source and target alone (e.g., when there are two family members with the same full name). On the other hand, a tool relying on input data such as deltas and correspondences may be used only in contexts where these data are available. For example, a tool relying on stored correspondences may not be applicable if source and target have been created independently. In this sense, there is no optimal tool architecture.

4 Test Cases

Figure 3 depicts a feature model for bx tool architectures used to classify the different bx approaches. The nodes of the tree are the "features" that a given bx tool architecture can possess. Features can be optional or mandatory, children features can only be chosen together with their parent feature, and children of the same parent can be either exclusive or non-exclusive alternatives. Features are abstract (grey fill) if they can be implied from the set of concrete features (white fill) in a given product. The feature model depicted in Fig. 3 yields exactly the bx tool architectures described above in terms of involved computation steps.

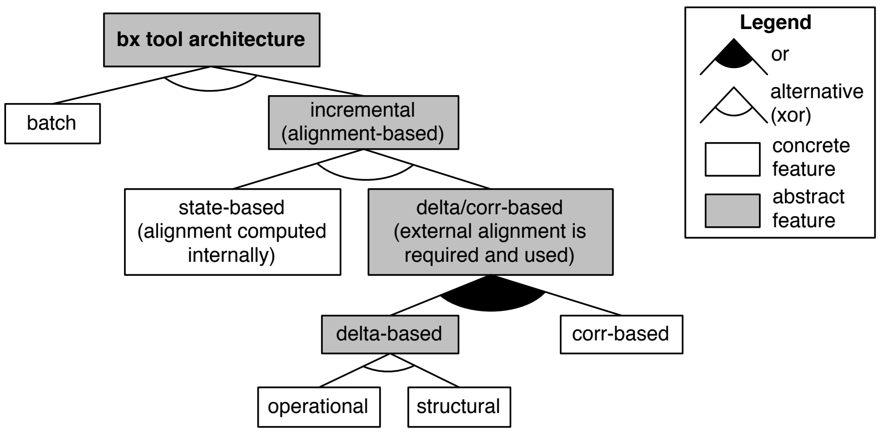

bx tool architecture
batch
incremental (alignment-based)
state-based (alignment computed internally)
delta/corr-based (external alignment is required and used)
delta-based
corr-based
operational
structural
Legend
or
alternative (xor)
concrete feature
abstract feature

Figure 3: Bx tool architecture variability as a feature model

As an extension of the feature model for bx tool architectures, Fig. 4 depicts a feature model for benchmarx test cases. Every benchmarx test case must state the required bx tool architecture (cf. Fig. 3), its direction to be fwd (forward), bwd (backward), or both (which means round-trip), the combination of different change types applied in the test, and the required update policy to successfully pass the test. The set of possible change types, currently del (deletion), add (addition), move, and attribute (attribute manipulation) can be extended in the future to accommodate more expressive frameworks. Note that a test case can require an update policy that is a mixture of fixed, i.e., design time preferences and conventions, and runtime configuration (in the Families to Persons example, only the backward direction needs configuration parameters).

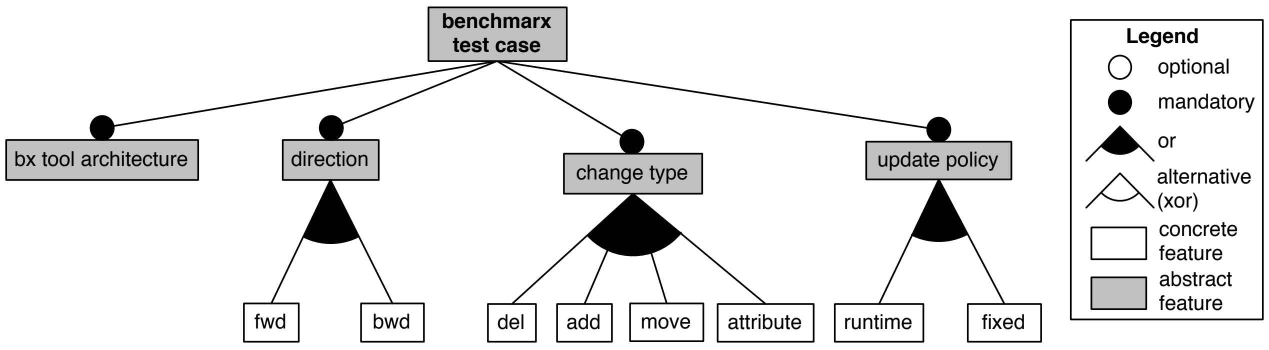


Figure 4: Test case variability as a feature model

The test cases provided for the Families to Persons case are separated in two different categories: (1) batch test cases and (2) incremental test cases. Each category comprises test cases for each transformation direction. Please note that in the following the term *forward* is used for transformations from the Families model to the Persons model. Accordingly, a *backward* transformation describes a transformation from the Persons model to the Families model.

While the batch test cases are used to check if a given source model is transformed in to a target model correctly, incremental ones also take updates of the corresponding source models into account. This comprises in particular, renaming, deleting and moving family members or persons respectively. Please note that we tried to keep the number of test cases manageable. To this end, only the batch category contains separate test cases for each combination of parameters. In the incremental category, the parameters are changed dynamically within a backward test case.

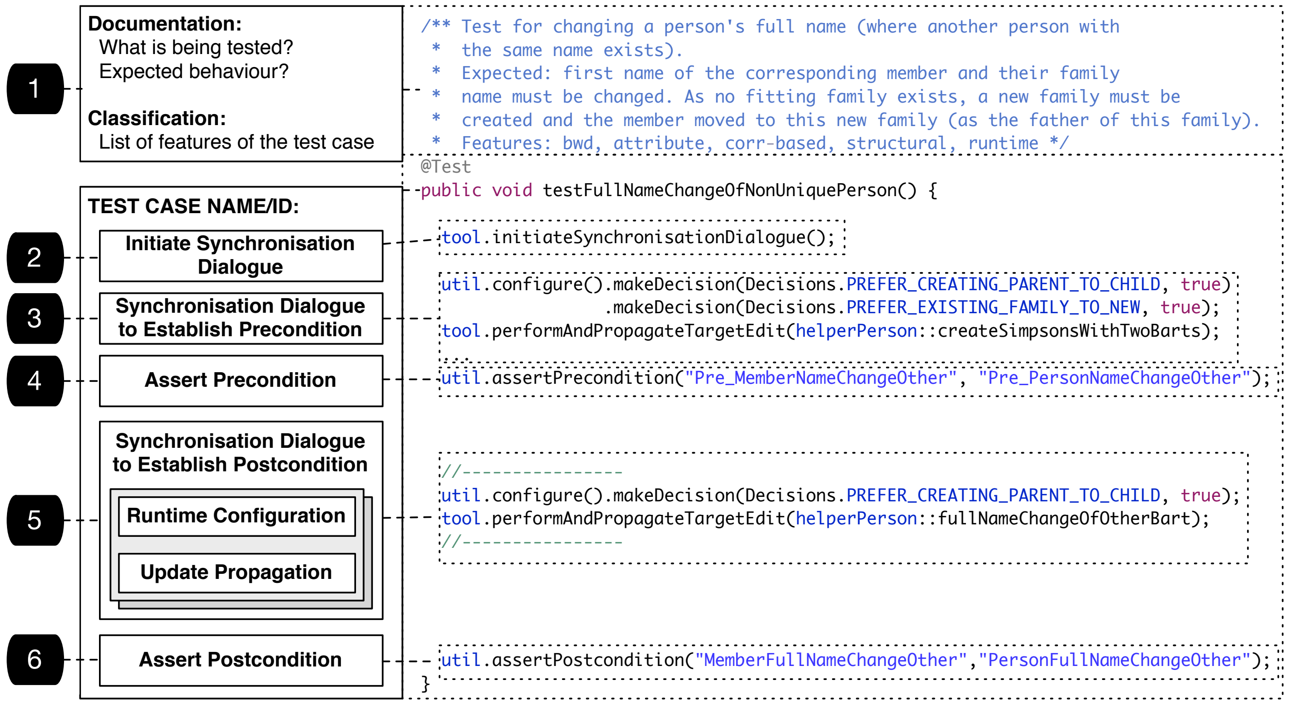
While the behaviour is clear for almost all test cases, testIncrementalRenamingDynamic needs some clarification: In the precondition for this test case, several persons are created, who are then transformed into corresponding families and members in the family model. In an edit delta, a person is renamed (full name is changed), such that the surname matches another existing family. The subsequent propagation to the Families model with parameter set to PREFER_EXISTING_FAMILY_TO_NEW should then move the existing family member to the corresponding family and change the first name accordingly. Please note that afterwards an empty family remains in the family register in case the old family only had one member (c.f., test case testIncrementalRenamingDynamic in class IncrementalBackward).

5 Implementation

5.1 Test Case design

Since we strive to provide a generic framework for benchmarking BX tools, and not all of the tools provide access to internal data structures like correspondence models or output deltas of a synchronization run, we decided to design each test case as a *synchronisation dialogue*, always starting from the same agreed upon consistent state, from which a sequence of deltas are propagated. Only the resulting output models are to be directly asserted by comparing them with expected versions.

A benchmarx test case is depicted schematically to the left of Fig. 5, with a concrete test case for our example to the right, following the proposed structure and representing an instantiation with JavaDoc, Java, and JUnit.



```
/** Test for changing a person's full name (where another person with
 *  the same name exists).
 *  Expected: first name of the corresponding member and their family
 *  name must be changed. As no fitting family exists, a new family must be
 *  created and the member moved to this new family (as the father of this family).
 *  Features: bwd, attribute, corr-based, structural, runtime */
@Test
public void testFullNameChangeOfNonUniquePerson() {
  tool.initiateSynchronisationDialogue();

  util.configure().makeDecision(Decisions.PREFER_CREATING_PARENT_TO_CHILD, true)
                  .makeDecision(Decisions.PREFER_EXISTING_FAMILY_TO_NEW, true);
  tool.performAndPropagateTargetEdit(helperPerson::createSimpsonsWithTwoBarts);
  util.assertPrecondition("Pre_MemberNameChangeOther", "Pre_PersonNameChangeOther");

  //----------------
  util.configure().makeDecision(Decisions.PREFER_CREATING_PARENT_TO_CHILD, true);
  tool.performAndPropagateTargetEdit(helperPerson::fullNameChangeOfOtherBart);
  //----------------

  util.assertPostcondition("MemberFullNameChangeOther","PersonFullNameChangeOther");
}
```

Figure 5: A benchmarx test case as a synchronisation dialogue

Every test case should be documented (cf. Label 1 in Figure 5) stating (i) what is being tested, (ii) expected behaviour, and (iii) a list of the concrete features of the test case taken from Fig. 4 to clarify at a glance if a given bx tool can pass the test or not.

Every test starts with an initialisation command invoked on the bx tool under test (Label 2), giving it the chance to establish the agreed upon starting point (e.g., for the Families to Persons benchmarx this comprises a single empty family register and corresponding single empty person register), and create all necessary internal auxiliary data structures.

The next part of a test case (Label 3) is a series of propagation steps, used to establish the precondition of the test. Although this creates a dependency to other tests (asserting exactly this precondition), this simplifies handling the required correspondence model, as the bx tool can build up the necessary internal state that must go along with the precondition. This means that the old consistent corr is “passed” implicitly to the bx tool via a series of preparatory propagation steps. The precondition is finally asserted (Label 4), representing a well-defined starting point. If the test requires a runtime update policy, this is configured just before propagating the actual input delta (Label 5). The last part of a test case (Label 6) is an assertion of the postcondition, checking if the final source and target models are as expected.

In the concrete exemplary test case, a number of persons are created in the person register and then backward propagated to establish a consistent family register that is asserted as a precondition. As part of the actual test, a person named `Simpson, Bart` is now renamed in the person register; this change is backward propagated with the update policy to prefer creating parents (if possible) to creating children. The interested reader is referred to the actual test cases[3] for all further details.

5.2 Supplied Test Cases

In its current state, the Benchmarx Framework provides in total 34 pre-defined test cases (see documentation on Github) for the Families to Persons example. They cover both batch and incremental scenarios for each transformation direction. As stated above, incremental backward tests change the parameter configuration at runtime during test case execution to avoid specifying a fixed test scenario for each possible parameter combination. In the batch cases, empty and non-empty families, families with duplicate names and families with duplicate member names are transformed in forward direction. The backward direction for the batch mode checks for each parameter combination the transformation of male or female persons and how duplicate names are handled. The incremental test cases take deletions, insertions, attribute changes (e.g., renaming), moving and a combination of deleting and inserting into account. However, if some cases are missing they may be supplied following the schema of the sample test case as discussed in the previous section.

5.3 Implementing the Test Runner

In order to integrate a specific BX tool into our Benchmarx Framework, the interface BXTool needs to be implemented. Please note, although the Benchmarx Framework itself is written in Java, it is possible to also use it with non-JVM-based BX tools (c.f., our reference implementation for the tool BiGUL [6]). Fig. 6 gives an overview and explanation of the methods to be implemented.

Reference implementations for *eMoflon* [7], *BiGUL* [6], *medini QVT*[4] and *BXtend* [4] may be found in the Github repository. Please note that for EMF-based tools, an abstract class BXToolForEMF already exists where tool integrators may subclass from. This class already contains implementations for both assert methods. As stated above, the method initiateSynchronisationDialogue is invoked in the beginning of every test and is used to establish a common starting point, including a single empty family register and its corresponding single and empty persons register as well as all necessary and tool-specific internal data structures. The methods performAndPropagateTargetEdit and performAndPropagateSourceEdit are called from the test cases when corresponding edit deltas should be performed and propagated on the corresponding models. Contrastingly, the methods performIdleTargetEdit and performIdleSourceEdit are used to modify source and target models respectively, without propagating the change. This method should be used whenever a change in the respective model does not affect the opposite model. In the test cases these methods are used to create empty families or edit the birthdates of persons in the person register.

Please note, that we use parameterized test cases, where the BXTools are the parameters. In order to run the test cases for a single tool, the respective test runner has to be instantiated in the corresponding collection in class FamiliesToPersonsTestCase.

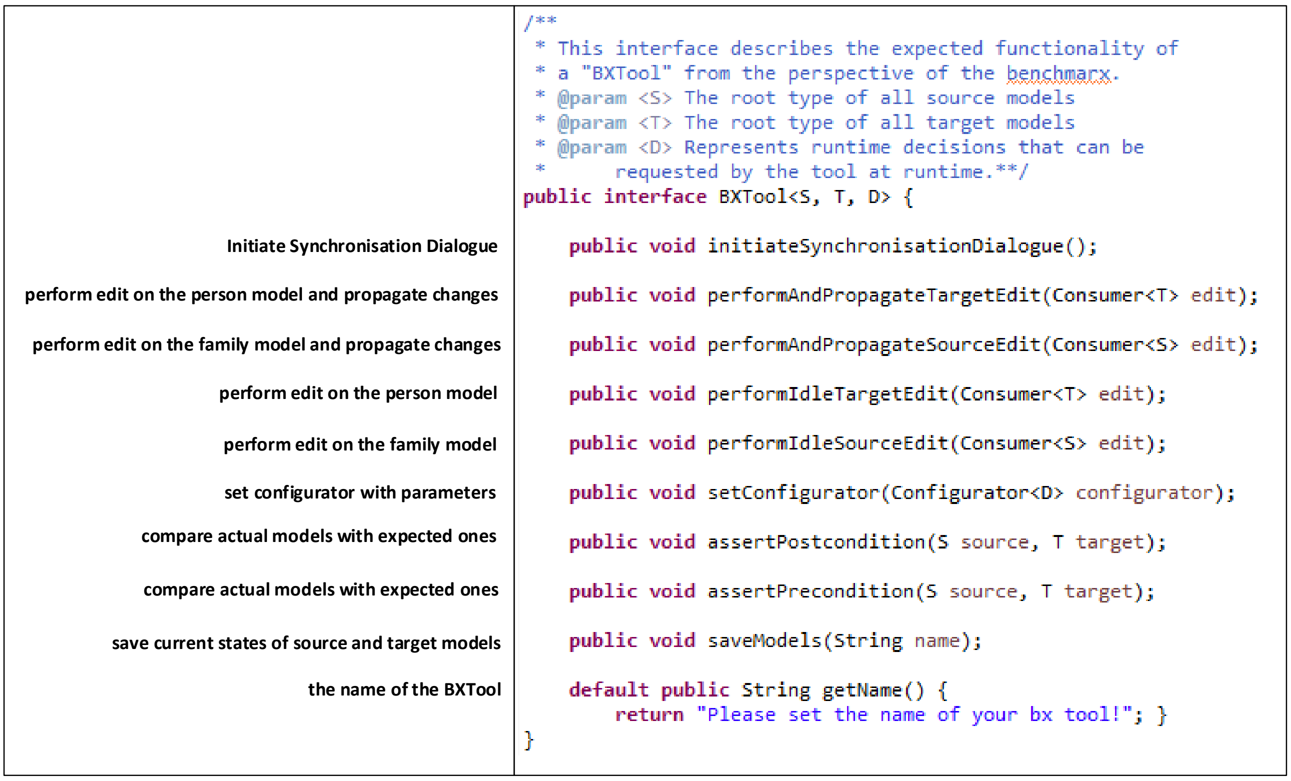
6 Evaluation

The goal of our benchmark is a qualitative evaluation of bx tools and the design of the test cases aims at revealing weaknesses or limitations of the chosen approaches. Due to the heterogeneity of bx tools, it is important to be able to easily and quickly distinguish between:

- A test that fails because it requires features that the tool does not support, is referred to as a *limitation*.
- A test that fails even though the tool should pass it (based on its classification), is referred to as a *failure*.

[3]Available from `https://github.com/eMoflon/benchmarx`

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


Initiate Synchronisation Dialogue
perform edit on the person model and propagate changes
perform edit on the family model and propagate changes
perform edit on the person model
perform edit on the family model
set configurator with parameters
compare actual models with expected ones
compare actual models with expected ones
save current states of source and target models
the name of the BXTool

```java
/**
 * This interface describes the expected functionality of
 * a "BXTool" from the perspective of the benchmarx.
 * @param <S> The root type of all source models
 * @param <T> The root type of all target models
 * @param <D> Represents runtime decisions that can be
 *      requested by the tool at runtime.**/
public interface BXTool<S, T, D> {

    public void initiateSynchronisationDialogue();

    public void performAndPropagateTargetEdit(Consumer<T> edit);

    public void performAndPropagateSourceEdit(Consumer<S> edit);

    public void performIdleTargetEdit(Consumer<T> edit);

    public void performIdleSourceEdit(Consumer<S> edit);

    public void setConfigurator(Configurator<D> configurator);

    public void assertPostcondition(S source, T target);

    public void assertPrecondition(S source, T target);

    public void saveModels(String name);

    default public String getName() {
        return "Please set the name of your bx tool!"; }
}
```

Figure 6: The BXTool interface

Limitations confirm the tool's set of (un)supported features, while failures indicate potential bugs in (the implementation of the benchmarx with) the bx tool. Similarly, one should clearly distinguish between:

- A test that the tool should (based on its classification) pass and passes, is referred to as an *expected pass*.
- A test that the tool passes even though this is unexpected in general, is referred to as an *unexpected pass*.

Expected passes confirm that the tool is correctly classified and behaves as expected, while unexpected passes indicate that the test case can either be improved, as it is unable to reveal the missing features of the tool, or that the bx tool has not been classified correctly.

Scalability of the transformation approaches is evaluated in simple transformations carried out on models while the number of model elements is being raised constantly. Our benchmark measures the time needed for both batch and incremental transformations in forward and backward direction.

Furthermore, we are interested in the metrics `[Lines of Code, Number of Words, Number of Characters]` of the transformation specifications. Those numbers should also be supplied in the spreadsheet[5].

Submissions to the Families to Persons case for the TTC should be made by cloning the provided SHARE VM [3] and contain the following:

- An implementation of the transformation including the test runner (c.f., Section 5.3).
- A spreadsheet containing the numbers retrieved from running the supplied test cases.
- The metrics of the transformation specification (see above).
- A plot similar to the ones shown in [2] based on the numbers retrieved from the performance test.

7 Conclusion

In this paper we propose as well-known example for bidirectional transformations as a case for the transformation tool contest. We propose to use the benchmarx [2] framework as a test environment to evaluate the solutions. The need for a benchmark for bidirectional transformations has been recognised for long in the bx community. The

[5]not possible for graphical approaches

benchmarx presented in this paper is targeted at classifying and evaluating heterogeneous bx tools and approaches using a common transformation scenario. The propsed transformation tool case helps us to disseminate, evaluate, and improve the benchmarx infrastructure and provides a rich overview about existing bx tools and their benefits and drawbacks on the other hand. Several reference implementations for the transformation case created with *eMoflon*, *BiGUL*, *medini QVT* and *BXtend* are also available in the repository[6].

References

[1] Anthony Anjorin, Alcino Cunha, Holger Giese, Frank Hermann, Arend Rensink, and Andy Schürr. Benchmarx. In K. Selccuk Candan, Sihem Amer-Yahia, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014)*, CEUR Workshop Proceedings, pages 82–86. CEUR-WS.org, 2014.

[2] Anthony Anjorin, Zinovy Diskin, Frédéric Jouault, Hsiang-Shang Ko, Erhan Leblebici, and Bernhard Westfechtel. Benchmarx reloaded: A practical framework for bidirectional transformations. In Romina Eramo and Michael Johnson, editors, *Sixth International Workshop on Bidirectional Transformations (BX 2017)*, CEUR Workshop Proceedings, April 2017.

[3] anthony.anjorin@upb.de. Online demo: Benchmarx. `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_BenchmarX.vdi`, 2017.

[4] Thomas Buchmann and Sandra Greiner. Handcrafting a triple graph transformation system to realize round-trip engineering between UML class models and java source code. In Leszek A. Maciaszek, Jorge S. Cardoso, André Ludwig, Marten van Sinderen, and Enrique Cabello, editors, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016.*, pages 27–38. SciTePress, 2016.

[5] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In Richard F. Paige, 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, June 2009. Springer-Verlag.

[6] Hsiang-shang Ko, Tao Zan, and Zhenjiang Hu. BiGUL: a formally verified core language for putback-based bidirectional programming. *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation - PEPM 2016*, pages 61–72, 2016.

[7] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Developing emoflon with emoflon. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, volume 8568 of *Lecture Notes in Computer Science*, pages 138–145. Springer, 2014.

[6]`https://github.com/eMoflon/benchmarx`