Benchmarx Reloaded: A Practical Benchmark Framework for Bidirectional Transformations

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

Zinovy Diskin
McMaster University, Canada
diskinz@mcmaster.ca

Frédéric Jouault
ESEO, France
frederic.jouault@eseo.fr

Hsiang-Shang Ko
NII, Japan
hsiang-shang@nii.ac.jp

Erhan Leblebici
TU Darmstadt, Germany
erhan.leblebici@es.tu-darmstadt.de

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

Abstract

Bidirectional transformation (bx) approaches provide a systematic way of specifying, restoring, and maintaining the consistency of related models. The current diversity of bx approaches is certainly beneficial, but it also poses challenges, especially when it comes to comparing the different approaches and corresponding bx tools that implement them. Although a benchmark for bx (referred to as a *benchmarx*) has been identified in the community as an important and currently still missing contribution, only a rather abstract description and characterisation of what a benchmarx should be has been published to date. In this paper, therefore, we focus on providing a practical and pragmatic framework, on which future concrete benchmarx can be built. To demonstrate its feasibility, we present a first non-trivial benchmarx based on a well-known example, and use it to compare and evaluate three bx tools, chosen to cover the broad spectrum of bx approaches.

1 Introduction and Motivation

Bidirectional transformations (bx) are mechanisms for specifying and maintaining consistency between two (in general multiple) related sources of information. Bx is currently an active research area as the challenge of "consistency management" in software systems is still largely open. Inconsistencies arise as different actors (tools, devices, or users) operate independently on shared information, and being able to maintain consistency will become even more important with the increasing complexity of software systems and usage workflows.

Bx approaches are quite diverse and are based on different foundations from various communities including programming languages, constraint solving, and graph transformation. As a consequence, although sharing common goals, bx tools differ in many aspects such as typical application scenarios, ways of specifying and restoring consistency, traceability support, and performance (to name but a few).

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: R. Eramo, M. Johnson (eds.): Proceedings of the Sixth International Workshop on Bidirectional Transformations (Bx 2017), Uppsala, Sweden, April 29, 2017, published at http://ceur-ws.org

This diversity is certainly advantageous from a research community point of view as it provides a broad perspective on bx, but it also makes discerning the exact capabilities and limitations of individual approaches challenging. The need for a bx benchmark (referred to as a *benchmarx* in the following) has been discussed in detail by Anjorin et al. [3], together with an abstract description of such a benchmarx. A benchmarx should enable direct comparisons of bx tools based on a maintained set of examples and *executable* test cases. While benchmarking is a common and well established practice for conducting evaluation and comparison in general, a benchmarx exhibits additional characteristic properties and challenges that are unique to bx.

Although Anjorin et al. [3] discuss what should be provided by a benchmarx in an attempt to encourage concrete benchmarx efforts, there has been no practical realisation of a benchmarx until now. Providing a series of benchmarx has been a recurring topic discussed at different bx events (e.g., [1, 5]) involving representatives from all bx sub-communities. A bx example repository [4] has been established and is being actively maintained (prevalently via examples introduced in bx-related talks and publications), but there is still no non-trivial executable test suite for any of the examples that can be run, adapted, and interpreted for the numerous bx tools that exist. We believe this is because many *practical* challenges have not been addressed, including how to structure the test suite to simplify interpretation of test results for diverse bx tools with different capabilities.

In this paper, we close this gap by providing a pragmatic and practical benchmarx *framework* that can be used to establish further benchmarx implementations. Our contribution is thereby twofold: We propose a benchmarx *template* based on an analysis of bx tool workflows and a classification according to supported features. This classification into features is used to derive a concrete test suite structure that can be used not only to identify a meaningful subset of tests for a given bx tool, but also to evaluate test coverage. Furthermore, we provide an open-source reference implementation[1] of our framework, together with a concrete benchmarx using a simple but non-trivial example to evaluate and compare solutions with three very different bx tools.

The rest of the paper is structured as follows: A well-known bx example is introduced in Sect. 2 and used consequently throughout the paper to explain basic terminology, illustrate our benchmarx design, and demonstrate its usage. The architecture of our benchmarx implementation is provided in Sect. 3. Solutions for our running example (one functional programming-based, one rule-based, and one constraint-based) are presented in Sect. 4. In Sect. 5, these solutions are integrated into, evaluated, and discussed with our benchmarx implementation to demonstrate how to handle such a heterogeneous choice of bx tools. These results are a first demonstration of a tool comparison conducted with our benchmarx implementation, and should serve as encouragement for (more detailed) future tool comparisons. Finally, Sect. 6 concludes the paper and provides an outlook on future work.

2 Our Benchmarx Example: Families to Persons

To give concrete examples throughout the paper, we shall make consequent use of our chosen example for the benchmarx: "Families to Persons", a well-known demonstrative transformation in the model transformation community. The example is part of the ATL[2] transformation zoo[3] 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).

Figure 1 depicts the two data structures involved, which we shall refer to in the following as the "source" and "target" of the bidirectional transformation between them. The motivational story is that two administration offices of some city wish to maintain their respective data in a consistent state. Both source and target offices keep track of individuals registered in the city, but at different abstraction levels.

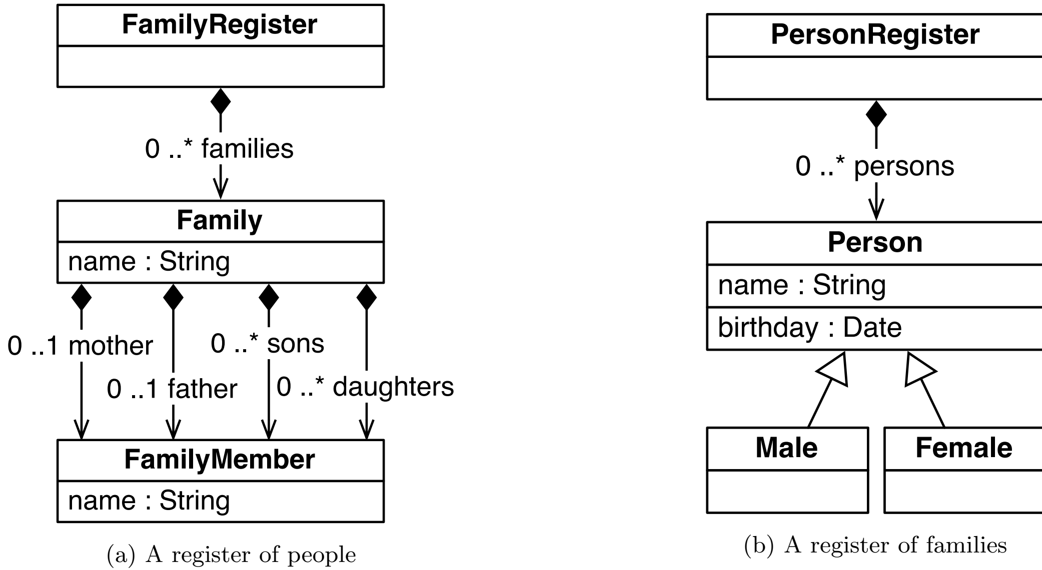
As depicted in Fig. 1a as a metamodel, the source data structure has the core concept of a family (`Family`), consisting of family members (`FamilyMember`). Both concepts are named entities, and family members play different roles in a family. Note that the associations are all composites (denoted with the black diamonds), implying that a family can only be in one family register, and that a family member can only be in one family at a given point in time. Furthermore, due to the multiplicities `0..1`, a family can have at most one mother and at most one father. This is just a simplification, ruling out realistic situations where a man can be both a father and son in different families, and a family can of course have two mothers or two fathers.

In contrast, the target data structure (Fig. 1b) only has the primary concept of a person (`Person`), who can be either male or female, but has a birthday in addition to a name. In the target domain, a person's name is formed by concatenating surname and first name, separated by a comma and space.

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

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

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



(a) A register of people

(b) A register of families

Figure 1: Source and target data structures

In summary, source and target models contain not only shared information on which they must agree (family members are persons), but also information only contained in one of the models (birthdays, family roles).

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.

This example, of which many variants exist including *FamilyToPersons* [2] in the bx example repository [4], was chosen for the following reasons, which can be taken into account when searching for a new benchmarx example: It is already well-known and well-accepted by a substantial community, indicating that the example is accessible to a wide range of people. It is also simple and invokes an intuitive expectation of consistency (e.g., that there should be a bijection between all family members in the source and all persons in the target). It can nonetheless be used to demonstrate a fair number of synchronisation challenges including information loss, non-determinism, flexible alignment, and update policies. Finally, the data structures involved are simple trees, simplifying its interpretation in other technological spaces.

3 Proposed Template for a Benchmarx

A major challenge in benchmarking bx tools is that different tools may use different input data. Addressing this challenge requires a unifying design space, in which different tool architectures can be placed and respectively classified. In Sect. 3.1 we shall define such a space and distinguish seven basic bx tool architectures according to the type of their input data. In Section 3.2, we then define a respective design space of test cases, and discuss how a tool's architecture influences the interpretation of test results.

3.1 Design space of bx tools

To establish a common foundation, we first present a vocabulary of basic notions, constructs, and terms based on previous work by Diskin et al. [7], which we shall use for discussing consistency restoration via change propagation. Let us refer to the two domains to be kept synchronised as the *source* and the *target model spaces*. Source models are denoted by A, A', etc., target models by B, B', etc. Transformations from the source to the target are called *forward (fwd)*, from the target to the source *backward (bwd)*.

3.1.1 Objects: models, deltas, and corrs

The objects that bx tools operate on are models and relationships/links between them. Relationships between models from different model spaces are called *correspondence links*, or just *corrs*. We shall denote corrs by double bidirectional arrows, e.g., $R\colon A \Leftrightarrow B$, and depict them in our diagrams horizontally. Normally, a corr is a set of links $r(a, b)$ between elements (a in A, b in B), which are compatible with the models' structure.

Relationships between models from the same model space are called *deltas*, denoted by directed arrows, e.g., $\Delta\colon A \to A'$, where we typically consider A' to be an updated version of A. To visually distinguish corrs from deltas, we shall depict deltas in our diagrams vertically. Deltas can be *structural* or *operational*.

A *structural delta (s-delta)* is a collection of links between elements of the original and the updated model, which is compatible with structure, e.g., is type preserving. In fact, an s-delta is a (usually simple) *vertical* corr between models of the same space. We denote s-deltas by double unidirectional arrows, e.g., $\Delta\colon A \Rightarrow A'$.

An *operational delta (o-delta)* is an operational specification δ of changes to be performed on a model to update it. Examples of such operational deltas include edit logs, in-place transformations, or transformation rules. A given o-delta δ can be applicable to some models but not to others. If it is applicable to a model A, its application results in (i) a new model A' denoted by $\delta@A$ (δ applied to A), and (ii) an s-delta $\Delta\colon A \Rightarrow A'$ relating A and A', which we denote by $\overrightarrow{\delta@A}$. It is important to distinguish s-deltas from o-deltas as different o-deltas may result in the same model and even the same s-delta, i.e., $\overrightarrow{\delta_1@A} = \overrightarrow{\delta_2@A}$ (and $\delta_1@A = \delta_2@A$) for $\delta_1 \neq \delta_2$. For instance, when the *order* in which updates are performed is important, then a tool must accept o-deltas as input. If o-delta δ is applicable to A, we write $\delta\colon A \xrightarrow{@}$.

Example (E1). Our source model space consists of Family Registers, and the target space of Person Registers. Consider a source model A consisting of a single family register with the `Simpson` family as a single family, consisting of family members `Homer` (as father) and `Marge` (as mother), and a target model consisting of a single person register with a single female person `Simpson, Marge`, and a single male person `Simpson, Homer` (with their birthdays). This pair of source and target models can be related by a corr $R\colon A \Leftrightarrow B$ including links connecting the family register with the person register, `Homer` with `Simpson, Homer`, and `Marge` with `Simpson, Marge`. As a source delta $\Delta\colon A \to A'$, consider adding a new family member `Bart` as a son in the `Simpson` family to produce a new source model A'. This source delta can either be given as an s-delta $\Delta\colon A \Rightarrow A'$ consisting of links between elements in A and A' including, e.g., a link connecting `Homer` in A with `Homer` in A'. The source delta can also be given as an o-delta $\delta\colon A \xrightarrow{@}$, e.g., some code creating a new family member `Bart` and adding it as a son in the `Simpson` family. This code can be applied to A to produce $\delta@A = A'$.

3.1.2 Basic operations employed by bx tools

Our next goal is to classify bx tools based on their (expected) capabilities. This is directly related to the input that a tool accepts and should exploit. Based on existing bx taxonomies [6, 8], as well as current bx tools under active development,[4] the following basic operations (visualised to the left of Fig. 2) can be used to provide a schematic overview of different bx tool architectures:

Alignment (vAln, hAln). There are two alignment operations: *vertical* and *horizontal* alignment, denoted by vAln and hAln, respectively.[5] The former takes two models A, A' from the same model space, and produces an s-delta between them: $\mathsf{vAln}(A, A')\colon A \Rightarrow A'$. The latter takes two models A, B from different model spaces, and produces a corr $\mathsf{hAln}(A, B)\colon A \Leftrightarrow B$. There is also an operation $*$ of corr composition (re-alignment) that takes a corr $R\colon A \Leftrightarrow B$ and either a source s-delta $\Delta\colon A \Rightarrow A'$ to produce a new corr $\Delta * R\colon A' \Leftrightarrow B$, or dually, a target s-delta $\mathrm{E}\colon B \Rightarrow B'$ to produce $R * \mathrm{E}\colon A \Leftrightarrow B'$.

Consistency is modelled by a predicate (a Boolean-valued function) check on the set of all corrs. For any corr $R\colon A \Leftrightarrow B$, either $\mathsf{check}(R) = 1$, i.e., the corr R is consistent, or $\mathsf{check}(R) = 0$, i.e., R is inconsistent.

Consistency Restoration (fCR, bCR). The operation of *forward consistency restoration* fCR, takes a corr $R\colon A' \Leftrightarrow B$ and restores consistency by changing the target model B and keeping A' intact. It thus produces an s-delta $\mathsf{fCRv}(R) = \mathrm{E}\colon B \Rightarrow B'$, and a new consistent corr $\mathsf{fCRh}(R) = R'\colon A' \Leftrightarrow B'$. Backward consistency restoration bCR operates dually by updating the source model and keeping the target model intact, i.e., it produces $\mathsf{bCRv}(R) = \Delta\colon A \Rightarrow A'$, and a consistent corr $\mathsf{bCRh}(R) = R'\colon A' \Leftrightarrow B'$.

Update Propagation (fUP, bUP). There are also two operations of o-delta propagation. Forward propagation fUP, takes a corr $R\colon A \Leftrightarrow B$ and an A-applicable o-delta $\delta\colon A \xrightarrow{@}$, and produces a B-applicable o-delta $\epsilon\colon B \xrightarrow{@}$. Backward propagation bUP operates analogously with $\mathsf{bUP}(R, \epsilon) = \delta$.

Example (E2). Consider the source delta $\Delta\colon A \to A'$ from $(E1)$, resulting in a new source model A' with `Bart` as a new family member. Given only A and A', vertical alignment vAln can be used to produce an s-delta

[4] `http://bx-community.wikidot.com/relatedtools`

[5] Remember that corrs and deltas are typically depicted horizontally and vertically, respectively.

vAln(A, A): $A \Rightarrow A'$. This could be achieved by making certain (arguably) reasonable assumptions, e.g., that families and family members with the same name in A and A' are actually the same objects and should not be deleted and re-created by the resulting s-delta. Given the source and target models A and B from $(E1)$, horizontal alignment hAln can be applied to establish a corr hAln(A, B): $A \Leftrightarrow B$ connecting them. This requires further assumptions, e.g., that family names, names of family members in a family, and names of persons in the person register are unique. These assumptions for vAln and hAln do *not* hold in general for our example.

Consider now the s-delta Δ: $A \Rightarrow A'$ (adding `Bart` as a son) and the old corr R: $A \Leftrightarrow B$. The operation of re-alignment $*$ can be applied to produce the corr $R^* = \Delta * R$: $A' \Leftrightarrow B$. In this case, as nothing is deleted, the links in R can simply be transferred to R^*, using Δ to identify the corresponding elements in A'.

The predicate check for our running example can be derived from the constraints stated in Sect. 2. For models A and B from example $(E1)$, it must be possible to extract the pairs (m_H, p_H) and (m_M, p_M) from a consistent corr R: $A \Leftrightarrow B$, where m_H, m_M are the family members with names `Homer` and `Marge`, and p_H, p_M the persons with names `Simpson, Homer` and `Simpson, Marge`, respectively. The new, re-aligned corr R^* is not consistent according to check as a bijection cannot be derived (there is no corresponding person for `Bart` in the person register). To restore consistency, the operation of forward consistency restoration fCR can be used to produce a target s-delta E: $B \Rightarrow B'$, and a new corr R': $A' \Leftrightarrow B'$. In this case, this could be achieved by creating a new person `Simpson, Bart` in the person register to produce the new target model B', and adding a new link, connecting the family member `Bart` with the person `Simpson, Bart`, to result in the new, now consistent corr R'. Analogously, the o-delta $\delta : A \xrightarrow{@}$ (update) adding `Bart` as a son to the `Simpson` family, can be forward propagated to the appropriate o-delta fUP$(R, \delta) = \epsilon$: $B \xrightarrow{@} B'$, adding `Simpson, Bart` to the person register.

3.1.3 A zoo of bx tool architectures

Based on these basic concepts and operations, seven bx tool architectures are depicted to the right of Fig. 2. We have restricted ourselves to realistic architectures for which we have found actual implemented bx tools (outlined in blue, red, and green), or for which such architectures are at least suggested in the literature (often with prototypical implementations). Various additional hybrids and variants are of course possible including, e.g., a state-based tool that also takes A and performs both vertical and horizontal alignment. For each entry in the table, the row and column represent the input that the tool accepts. For presentation purposes, we assume that both models A and A' (B) can be extracted from δ: $A \to A'$ $(R$: $A \Leftrightarrow B)$.

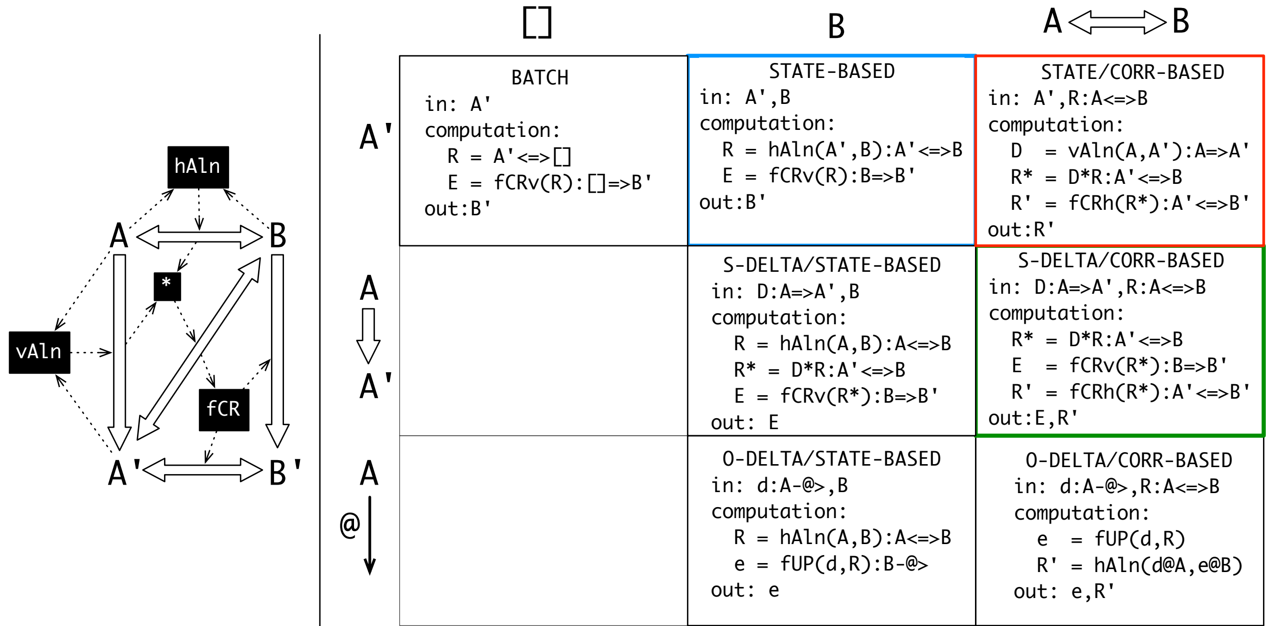


Figure 2: Basic operations (left) and input-based classification (right) of bx tools

A *batch* bx tool takes as input only the source model A' (the "empty" model is represented as `[]`) and produces as output B'. The involved computational tasks indicate that batch tools cannot exhibit good or efficient synchronisation behaviour in general: A batch tool can only assume trivial alignment of A' with the empty model `[]`, and is forced to reconstruct B' from scratch.

State-based tools take both the new source model A' and the old target model B as input, producing an updated target model B' as output. The required computations now make more sense, as horizontal alignment can be applied, and B used as a starting point from which to produce a consistent B'. This means that a state-based tool can handle information loss in the target domain.

To avoid horizontal alignment, *state/corr-based* tools take A' and a corr $R\colon A \Leftrightarrow B$. Such tools can handle cases where it is impossible to compute $hAln$ based on conventions.

Similarly, if conventions can be used to compute horizontal alignment, but *vertical* alignment is problematic, *s-delta/state-based* and *o-delta/state-based* tools can be used.

Tools that are *s-delta/corr-based* are able to exploit available correspondence information in both horizontal and vertical dimensions (Fig. 2). In such tools, alignment is completely replaced with re-alignment.

Finally, an *o-delta/corr-based* tool takes an o-delta directly as input and can thus handle cases where the *order* of changes is important.

Taken from the domain of (software) product line engineering, a *feature model* [9] consisting of features and constraints over features can be used to succinctly capture the space of valid variants or *products*.

Figure 3 depicts a feature model for bx tool architectures. Every bx tool architecture must be a valid "product" of the product line described by this feature model. 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 seven bx tool architectures described in Fig. 2 in terms of involved computation steps.

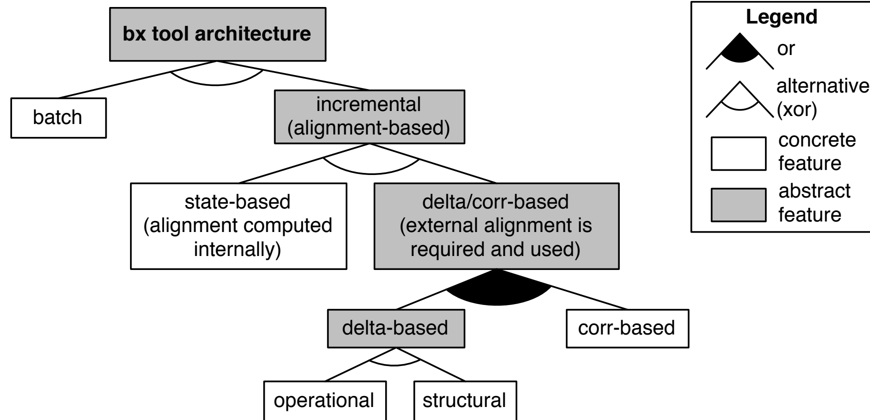


Figure 3: Bx tool architecture variability as a feature model

To further illustrate the architecture of the bx tools most relevant for this paper, workflow diagrams for state-based, state/corr-based, and s-delta/corr-based bx tools are depicted in Fig. 4, Fig. 5, and Fig. 6. Apart from visualising the computation flow, an update policy is indicated as additional input. The task of consistency restoration typically involves decisions: think of setting `Simpson, Bart`'s birthday in the forwards direction, or deciding if a female person corresponds to a mother or daughter in the backwards direction. An update policy is used to handle these choices, either at design time (e.g., hard-coded preferences and conventions concerning default values), or at runtime (e.g., user interaction). In the workflow diagrams, the symbol $\otimes$ is used to either suppress output, or restrict it to only a part of what is available (e.g., only B' from $\Delta\colon B \Rightarrow B'$).

Example (E3). In our example, *explicit corrs* are required as input to handle non-unique person names. *Explicit deltas* are required as input in our example to handle, e.g., renaming vs. create/delete operations. In both these cases, computing alignment internally based on conventions cannot work in general. As an example of how the *order of updates*, i.e., taking o-deltas directly as input, can be important, consider the fixed update policy of preferring to creating mothers to daughters in the backwards transformation. As families can have only

one mother, the *order* in which two females with the same family name are added to the person register, say `Simpson, Marge` and `Simpson, Maggie`, is important to avoid creating `Maggie` as mother and `Marge` as daughter of a new family in the register.

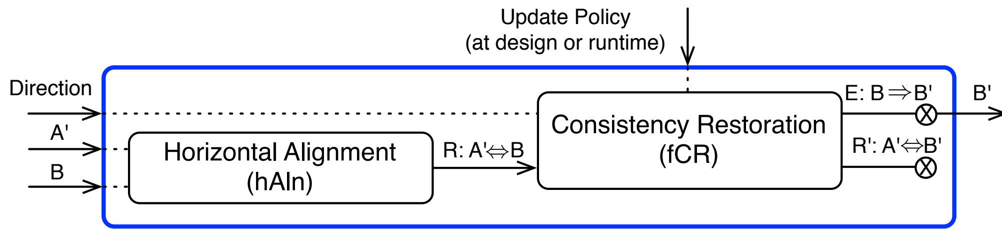


Figure 4: Forward workflow for state-based bx tools

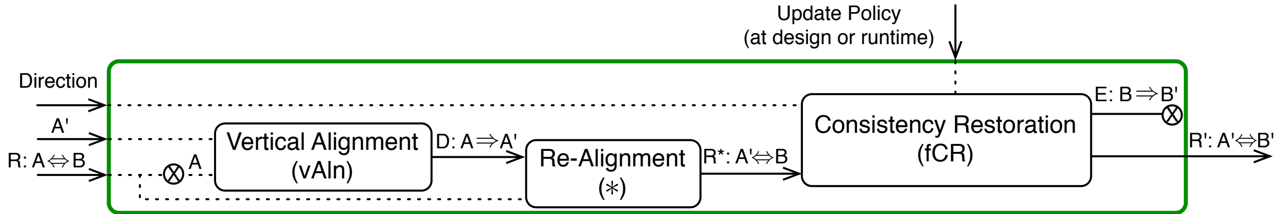


Figure 5: Forward workflow for state/corr-based bx tools

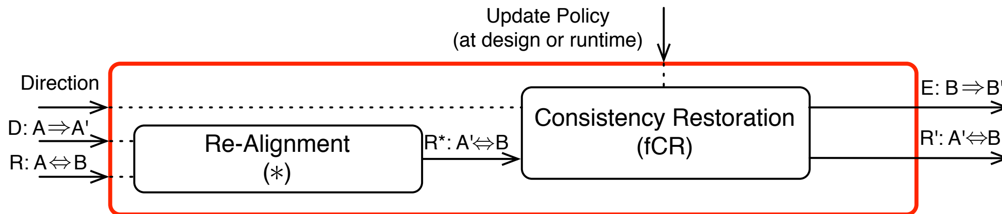


Figure 6: Forward workflow for s-delta/corr-based bx tools

3.2 Design space of test cases

The current[6] test suite for the FamiliesToPersons benchmarx consists of 52 test cases, and will be constantly revised as further implementations are established or extended. As such a test suite grows in size, adding test cases systematically is important to avoid redundancy and to have some measure for the current coverage and which *types* of tests are still missing. Based on our classification of bx tool architectures, this section identifies a respective design space for actual test cases, providing a feature-based classification of benchmarx test cases in Sect. 3.2.1, a basic test case structure in Sect. 3.2.2, and a classification of test results in Sect. 3.2.3.

3.2.1 Feature-based classification of test cases

As an extension of the feature model for bx tool architectures from Sect. 3.1.3, Fig. 7 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 one of `fwd` (forward), `bwd` (backward), or `round trip` (a mixture), 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), 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.

3.2.2 Test cases as synchronisation dialogues

Although it would be certainly interesting to directly assert the resulting corr and output delta of a synchronisation run, our investigations with bx tools have revealed that these data structures are typically tool-specific

[6] As of February 28, 2017.

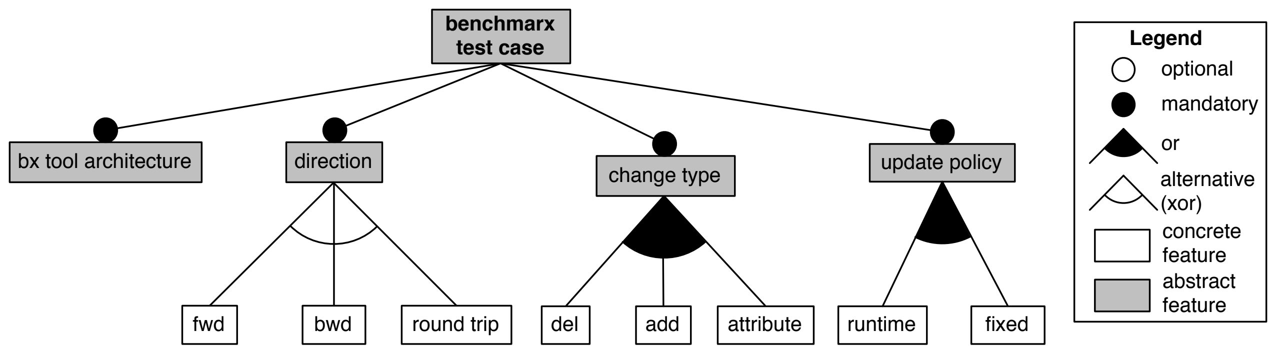


Figure 7: Test case variability as a feature model

and complicate the task of test case specification. To cope with this in a pragmatic manner, we propose 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. 8, with a concrete test case for our example to the right, following the proposed structure and representing an instantiation with JavaDoc, Java, and JUnit.

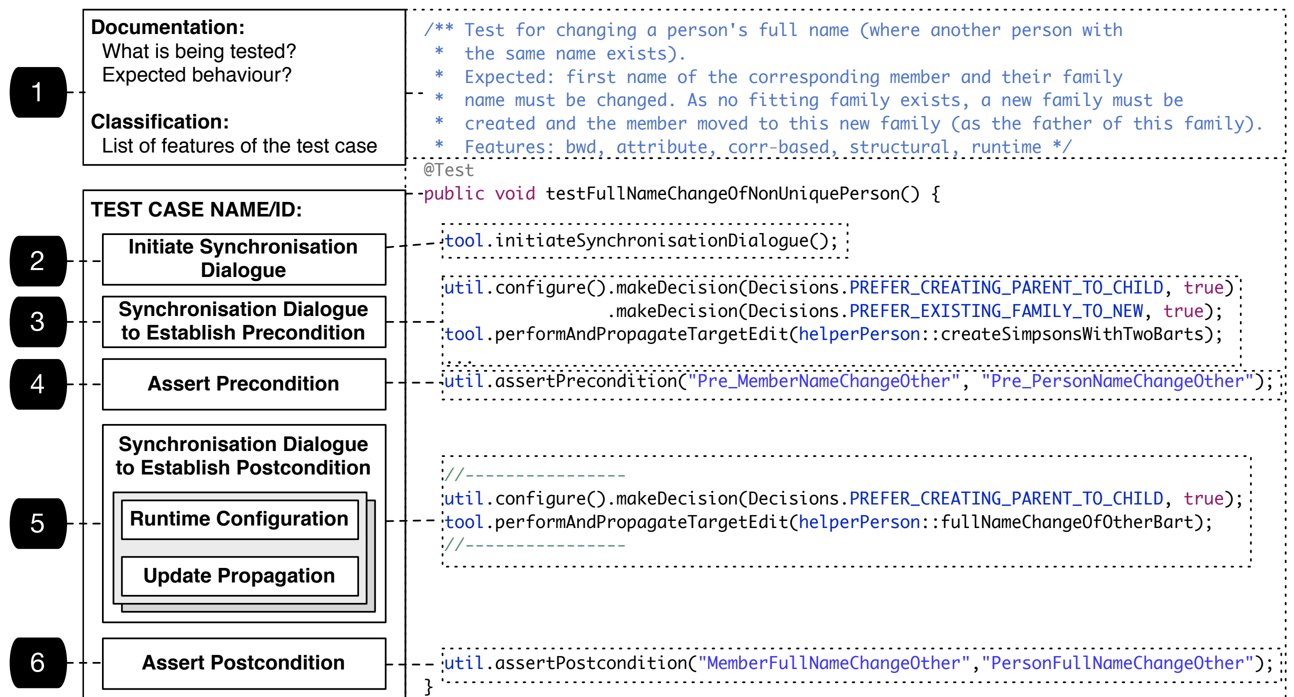


Figure 8: A benchmarx test case as a synchronisation dialogue

Every test case should be documented (cf. Label 1 in Figure 8) stating (i) what is being tested, (ii) expected behaviour, and (iii) a list of the concrete features of the test case taken from Fig. 7 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 FamiliesToPerson benchmarx this comprises a single empty family register and corresponding single empty persons 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[7] for all further details.

3.2.3 Interpretation of test results: Failure, limitation, or (un)expected pass

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, referred to as a *limitation*.
- A test that fails even though the tool should pass it (based on its classification), referred to as a *failure*.

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, referred to as an *expected pass*.
- A test that the tool passes even though this is unexpected in general, 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.

4 The FamiliesToPersons (F2P) Benchmarx: Version 1.0

In this section, we defend our choice of three bx tools, present three implementations of the ($F2P$) benchmarx with these tools, and discuss initial results and insights.

To demonstrate how to integrate diverse bx tools, we have chosen three bx tools with the aim of covering all the primary groups of BX approaches: BiGUL,[8] eMoflon,[9] and medini QVT[10]. Bx approaches can be very broadly classified into:

Functional programming (FP) based approaches: The basic idea here is to program a single function (forward or backward) that can be used to infer the other function (backward or forward) such that the pair obeys a set of round tripping laws. Referring to forward propagation as *get* and backward propagation as *put*, as is usual in an asymmetric setting, this idea can be realised either by using *get* to infer *put* (get-based approaches) or by using *put* to infer *get* (putback-based approaches). A final strategy is to provide a library of pairs of (get, put) that already obey the round tripping laws, and use a set of *combinators* to allow composition (combinator-based approaches). In all cases, the underlying consistency relation is not explicitly specified. We have chosen BiGUL as a representative of this group of bx approaches.

Grammar-based approaches: While FP-based approaches allow for fine granular control of all details of consistency restoration, grammar-based approaches provide a high-level, rule-based language, with which the language of all consistent pairs of source and target models can be generated. Such a specification is under-specified as the consistency relation (which equates here to language membership) is fully specified, but not all details of consistency *restoration* are fixed. Although this technique is not restricted to graphs and can be transferred to strings, lists, or trees, Triple Graph Grammars (TGGs) [10] are the primary representative of this group, and have been implemented by various tools. We have chosen eMoflon as a representative of this group.

[7] Available from `https://github.com/eMoflon/benchmarx`
[8] `www.prg.nii.ac.jp/project/bigul/`
[9] `www.emoflon.org`
[10] `http://projects.ikv.de/qvt`

Constraint-based approaches: Even more high-level than grammar-based approaches, constraint-based approaches leave all details of consistency restoration open and only require a specification of the consistency relation. Consistency restoration is often realised via a constraint solver, guided via a global objective function characterising "preferred" strategies. We have chosen medini QVT[11] as a constraint-based bx tool.

Besides the fundamental differences regarding the underlying bx approach, the tools also differ with respect to their architectures: We decided to use BiGUL to implement a state-based solution (Fig. 4) with a design-time update policy. In contrast, we used the TGG tool eMoflon to implement an s-delta/corr-based solution (Fig. 6) with support for a runtime update policy, while medini QVT was used to implement a state/corr-based solution (Fig. 5) with a design-time update policy.

While other cases can possibly be supported by the same tools, e.g., by encoding explicit delta and correspondence information in "states" for BiGUL, or calculating the "best fitting" correspondence model and implementing an additional delta discovery procedure for eMoflon, the choices above represent the "standard" case for these tools, i.e., the sweet spot for the tool for which minimal effort is required. We now give an overview of the three solutions to the F2P benchmarx in the following.

4.1 A solution to the F2P benchmarx with BiGUL

A BiGUL program describes how consistency is established for two models having *asymmetric* information, in the sense that one model contains more information than the other. When programming in BiGUL, the programmer assumes that two (possibly inconsistent) models are given, and describes how to put all data in the less informative model into proper places in the more informative model, thereby producing an updated version of the latter model that is consistent with the former model.

For symmetric scenarios such as the F2P benchmarx, symmetric consistency restoration can be achieved by combining two asymmetric consistency restorers that synchronise the two models with a common intermediate model. This intermediate model can either incorporate all the information of both models, or only contain the information shared by both models. Our BiGUL implementation of the F2P benchmarx adopts the latter approach: A family register is consistent with a person register exactly when the two registers are consistent with the same intermediate model, which is a collection of shared people (names and genders) with the Haskell `type MediumR = [((String, String), Bool)]`. The overall structure of the BiGUL implementation can thus be thought of as two programs putting `MediumR` into `FamilyRegister` (Figure 1a) and `PersonRegister` (Figure 1b) respectively. If one register is changed, say the family register, the collection of people it represents will be computed and put into the person register; the updated person register will represent the same collection of people, thus re-establishing consistency.

To better organise the implementation, we introduce another intermediate model `MediumL`, whose structure is closer to `FamilyRegister`: A model of `type MediumL = [(String, [(String, Bool)])]` is a collection of families, each of which consists of a family name and a list of first names and genders of its members. A more accurate depiction of the structure of the implementation is thus:

$$\texttt{FamilyRegister} \xleftarrow{\texttt{syncL}} \texttt{MediumL} \xleftarrow{\texttt{syncM}} \texttt{MediumR} \xrightarrow{\texttt{syncR}} \texttt{PersonRegister}$$

where the arrows indicate the directions of the constituent *put* programs. Besides straightforward format conversion (e.g., converting `True` and `False` to `Male` and `Female`), each of the three *put* programs is in charge of a part of the consistency restoration policy: (i) `syncR` aligns a collection of people with a person register, and describes how to update (retain, create, or delete) the birthdays in the person register; (ii) `syncM` aligns a collection of people with a collection of families, and describes how families are created or deleted, and how people are assigned to families; (iii) `syncL` describes how members in each family are assigned their roles.

As an example of a BiGUL specification, the code for `syncM` is depicted in Listing 1. The three "normal" branches deal with the cases where the collections of people and families are consistent, while the last "adaptive" branch describes how the collection of families can be updated to become consistent with the collection of people, applying the fixed update policy of matching each person with an entry with the same name and gender in an existing family, or creating a new entry in the first family with matching family name. If no matching family can be found, a new family is created.

[11]Even though "QVT" is in its name, we regard medini QVT as simply a constraint-based bx tool that just happens to have a QVT-R inspired textual concrete syntax (like many others such as Echo: `http://haslab.github.io/echo/` or JTL: `http://jtl.di.univaq.it`). We do not regard medini QVT as an implementation of the QVT-R standard, as its semantics diverge significantly.

```haskell
syncM :: BiGUL MediumL MediumR
syncM = Case
  [ $(normalSV [p| [] |] [p| [] |] [p| [] |])
    ==> $(update [p| _ |] [p| [] |] [d| |])
  , $(normalSV [p| ((familyName, []):_) |] [p| _ |] [p| (_, []):_ |])
    ==> $(update [p| _:rest |] [p| rest |] [d| rest = syncM |])
  , $(normal [| \((familyName, (firstName, gender):_):_) vs ->
                  ((familyName, firstName), gender) `elem` vs |]
             [p| (_, _:_):_ |])
    ==> $(rearrS [| \((familyName, (firstName, gender):ns):ss) ->
                       (((familyName, firstName), gender), (familyName, ns):ss) |])$
          (Replace `Prod` syncM) `Compose` extract
  , $(adaptive [| \_ _ -> otherwise |])
    ==> adapt
  ]
  where
    extract :: (Ord a, Show a) => BiGUL (a, [a]) [a]
    extract = Case
      [ $(normal [| \(s, _) (v:_) -> v == s |] [| \(s, ss) -> null ss || head ss >= s |])
        ==> $(update [p| (x, xs) |] [p| x:xs |] [d| x = Replace; xs = Replace |])
      , $(normal [| \(s, _:_) (v:_) -> v < s |] [| \(s, s':_) -> s' < s |])
        ==> $(rearrS [| \(s, s':ss) -> (s', (s, ss)) |])$
              $(rearrV [| \(v:vs) -> (v, vs) |])$
                Replace `Prod` extract
      , $(adaptive [| \(s, []) (v:_) -> v < s |])
        ==> \(s, _) _ -> (s, [undefined])
      ]
    adapt :: MediumL -> MediumR -> MediumL
    adapt [] vs = map ((fst . fst . head) &&& map (snd *** id))
                      (groupBy ((==) `on` (fst . fst)) vs)
    adapt ((familyName, ns):ss) vs =
      let ns' :: [(String, Bool)]
          ns' = concat (map snd (filter ((== familyName) . fst) ss))
          vs' :: [(String, Bool)]
          vs' = map (snd *** id) (filter ((== familyName) . fst . fst) vs) \\ (ns' \\ ns)
      in  (familyName, vs') : adapt ss (vs \\ (map ((familyName,) *** id) vs'))
```

Listing 1: An excerpt from the solution in BiGUL

4.2 A solution to the F2P benchmarx with eMoflon

The recommended way of "thinking in TGGs" is to describe how two consistent models *are to evolve simultaneously*, independently of how the TGG is to be ultimately operationalised, i.e., used to derive a directed (forward/backward) propagation of changes in one model to the other. Considering our running example, an obvious step in a simultaneous construction is the creation of a family member (father, mother, daughter, or son) in the family register, and the corresponding creation of a person (male or female) in the person register, indicating that four different cases must be handled at the minimum. Another orthogonal dimension is whether a family member is created in an existing family or in a new family, i.e., a new family could be created with its very first member. There are thus eight (4x2) different cases required to handle the construction of family members and corresponding persons simultaneously.

An excerpt from the TGG rules designed with this understanding of simultaneous construction in mind (and specified with eMoflon) is depicted in Figure 9. We omit the trivial construction step of required root containers (creating an empty family register together with an empty person register), and focus on the different cases for creating family members and corresponding persons. The TGG rules in Figure 9 are depicted using eMoflon's textual syntax (eMoflon also provides a read-only visualisation in the common visual syntax of graph grammars). In all rules, created elements are green and provided with a ++-markup, while context elements (which must be already present in order to apply a rule) are black.

To the left, an *abstract* rule `FamilyMemberToPerson` (note the keyword `#abstract`) is given which creates a family, its first family member, a person, and a correspondence link between the family member and the person. Two decisions are left open: the exact role of the family member in the family, and the concrete type (male or female) of the person. Two rules in the middle (`MotherToFemale` and `FatherToMale`) *refine* this abstract rule (note the `#extends` keyword) and clarify these open decisions for mothers and fathers (analogous rules creating sons and daughters are present but not explicitly depicted).

In the right part of Figure 9, two further rules (`MotherOfExistingFamilyToFemale` and `FatherOfExistingFamilyToMale`) refine the previous ones by requiring the family as existent context (as opposed to creating it). In both rules, an additional Negative Application Condition (NAC), depicted blue and with a `!`-markup, forbids the existence of a mother or father (depending on what is created) in the family, in order to prevent generating families with multiple mothers or fathers. Analogous rules creating sons and daughters in existing families are again present but not explicitly depicted. These rules for sons and daughters do not, however, have the NAC as multiple sons and daughters are allowed in families.

```
#abstract #rule FamilyMember2Person
#source {
   fr : FamilyRegister {
      ++ -families->f
   }
   ++ f : Family
   ++ fm : FamilyMember
}

#target {
   pr : PersonRegister {
      ++ -persons->p
   }
   ++ p : Person
}

#correspondence {
   fr2pr : FamiliesToPersonsCorr {
      #src->fr
      #trg->pr
   }
   ++ fm2p: FamilyMemberToPerson {
      #src->fm
      #trg->p
   }
}

#attributeConditions {
   concat(", ", f.name,
          fm.name, p.name)
}
```

```
#rule MotherToFemale
#extends
FamilyMember2Person

#source {
   ++ f : Family {
      ++ -mother->fm
   }

   ++ fm : FamilyMember
}

#target {
   ++ p : Female
}
```

```
#rule FatherToMale
#extends
FamilyMember2Person

#source {
   ++ f : Family {
      ++-father->fm
   }

   ++ fm : FamilyMember
}

#target {
   ++ p : Male
}
```

```
#rule MotherOfExistingFamilyToFemale
#extends MotherToFemale

#source {
    fr:FamilyRegister {
       -families -> f
    }
    f:Family {
       -mother -> existingMother
    }
    !existingMother:FamilyMember
}
```

```
#rule FatherOfExistingFamilyToMale
#extends FatherToMale

#source {
    fr:FamilyRegister {
      -families -> f
    }
    f : Family {
      -father->existingFather
    }
    !existingFather : FamilyMember
}
```

Figure 9: An excerpt from the eMoflon solution used for the benchmarx.

4.3 A solution to the F2P benchmarx with medini QVT

Medini QVT follows a constraint-based approach to bx. A transformation is defined in terms of a set of relations which must hold between the participating models, and is executed in the direction of one of them. Consistency restoration ensures that for each relation and each instance of a source domain (source pattern), there is a corresponding instance of the target domain such that all constraints on the domain instances are satisfied; the same must hold in the opposite direction.

An excerpt of the solution to the F2P benchmarx with medini QVT is depicted in Listing 2. It involves five relations: one relation between the root elements, and one relation for each role of a family member. In the forward direction, each family member is mapped to a person of the same gender with the same full name. Without further provisions, each person would be transformed twice in the backward direction: once as a parent and once as a child. As this violates the consistency constraint that demands a bijective mapping between family members and persons, the application of one of these relations is suppressed in the backward direction by querying the binding state of some variable in the target domain. As the when clause is executed before the target domain is instantiated in the forward direction, the variables in the target domain will still be unbound when the transformation is executed and the condition `female.oclIsUndefined()` will thus only then be satisfied.

```
transformation families2persons (famDB : Families, perDB : Persons) {
    top relation FamilyRegister2PersonRegister {
        enforce domain famDB familyRegister : Families::FamilyRegister {};
        enforce domain perDB personRegister : Persons::PersonRegister {};
        where {
            Father2Male(familyRegister, personRegister);
            Son2Male(familyRegister, personRegister);
            Mother2Female(familyRegister, personRegister);
            Daughter2Female(familyRegister, personRegister);
        }
    }
    relation Mother2Female {
        familyName, firstName, fullName : String;
        enforce domain famDB familyRegister : Families::FamilyRegister {
            families = family : Families::Family {
                name = familyName,
                mother = mother : Families::FamilyMember { name = firstName }
            }
        };
        enforce domain perDB personRegister : Persons::PersonRegister {
            persons = female : Persons::Female { name = fullName }
        };
        where {
            fullName = familyName + ', ' + firstName;
            firstName = firstName(fullName);
            familyName = familyName(fullName);
        }
    }
    relation Daughter2Female {
        ... -- Like 'Mother2Female', replace 'mother' with 'daughters'
        when {
            female.oclIsUndefined();
        } -- Prevents application in backward direction
        ... -- See 'Mother2Female'
    }
    ...
}
```

Listing 2: An excerpt from the solution in medini QVT

5 A first tool comparison based on the F2P benchmarx

We compare in the following BiGUL, eMoflon, and medini QVT based on the F2P benchmarx. Our goal is to impart a first impression of what can be achieved with our proposed benchmarx infrastructure and its reference implementation, and to inspire future tool comparisons.

We chose to investigate the following research questions (referred to as **RQ** from now on) with our tool comparison, covering the different capabilities of bx tools (**RQ-1**), performance (**RQ-2.a,b,c**), and required implementation effort (**RQ-3**):

RQ-1: Which subset of bx test cases (cf. Section 3) can be handled by each of the three tools?
RQ-2.a: How do the tools scale in an initial batch transformation from scratch?
RQ-2.b: Does propagation time scale with the size of the update or with the size of all models?
RQ-2.c: Do the tools exhibit symmetric behaviour in runtime in both forward and backward directions?
RQ-3: How do the tools differ with respect to the size of the required transformation specification?

To investigate **RQ-1**, we executed 52 tests differing in their features such as direction, change type, and required update policy.[12] Figure 10 depicts a representative excerpt showing the results for eight test cases. As discussed in Sect. 3.2, we distinguish between *expected passes*, *failures*, *unexpected passes*, and *limitations* for each result with a given bx tool.

Our test results show that all tools are able to propagate simple updates such as creation of new families (#1 in Figure 10) or birthday changes of persons (#2) which should not trigger any changes in families. Renaming family members (#3) is propagated by all tools as renaming of persons (as the test requires). BiGUL, supporting

[12]The entire set of test cases and the results are available as part of our reference implementation (cf. Section 1).

#	Direction	Update Policy	Change Type	Test Case Description	BiGUL	eMoflon	medini QVT
1	fwd	fixed	add	create new families	PASS	PASS	PASS
2	roundtrip	fixed	attribute	change birthdays (nothing should happen)	PASS	PASS	PASS
3	fwd	fixed	attribute	rename family members	pass	PASS	PASS
4	roundtrip	runtime	attribute	rename person who should start a new family (altough a family with the same name exists)	fail	PASS	fail
5	fwd	fixed	del	delete family members whose names are not unique in persons	pass	PASS	PASS
6	roundtrip	runtime	add	create a male for whom a father exists	fail	PASS	fail
7	roundtrip	fixed	del	delete person	PASS	PASS	FAIL
8	roundtrip	fixed	del	delete person who was the first member in the family	PASS	FAIL	FAIL

PASS : expected pass **FAIL : *failure*** **pass : unexpected pass** **fail : *limitation***

Figure 10: An excerpt from the F2P test results

neither deltas nor corrs, cannot necessarily be expected to pass this test, i.e., a completely new person could be created with the new name. We therefore identify this result as an unexpected pass. In a similar test where renaming a person should start a new family (#4), eMoflon passes the test while medini QVT and BiGUL fail as expected (limitation) due to a missing mechanism to configure the required update policy. When deleting a family member whose name is not unique (#5), all tools pass the test (by deleting the respective person). BiGUL could potentially fail (unexpected pass) due to missing corrs, which are necessary to determine the exact mapping of persons to the non-unique family members. Creating a male person who should become a son (#6) is only propagated by eMoflon as required, while medini QVT and BiGUL fail, again due to a non-configurable update policy. Deleting a unique person (#7) is propagated by eMoflon and BiGUL correctly (by deleting the respective family member) whereas medini QVT fails in this test by performing a backward transformation from scratch and thus creating new families without sons or daughters. Finally, eMoflon also fails in a similar test if the deleted person was the first member of their family (#8), due to a suboptimal strategy for handling deletion updates. This result is thus a failure for eMoflon.

From all 52 test cases, eMoflon has 51 expected passes and one failure, medini QVT has 11 expected passes, 15 unexpected passes, four failures, and 22 limitations, and finally, BiGUL has 12 expected passes, 19 unexpected passes, and 21 limitations. We thus conclude the following for **RQ-1**: Assuming a bx scenario where consistency specification is feasible with all three tools, they still differ in the set of their supported bx test cases. eMoflon strives to cover all cases with its configurable update policy as well as its s-delta and corr-based architecture. It can, however, still fail in some cases, by deleting more elements than absolutely necessary for consistency restoration. The BiGUL implementation supports a rather small subset of the tests for which configuration of the update policy, explicit deltas, and correspondences are not necessary. Nevertheless, BiGUL does not fail in any test case where it is expected to pass, indicating that it allows adequate control of the consistency restoration process. Medini QVT addresses a similar subset of the tests as BiGUL, additionally supporting correspondences, but often fails for round-tripping tests as it always (re-)transforms all persons from scratch to families without sons or daughters. As a final remark, the numerous unexpected passes indicate that more non-trivial tests are necessary to better reveal the limitations of the tools.

To investigate **RQ-2.a,b,c**, we generated source models of increasing size (consisting of uniquely named families with three children), performed a forward transformation from scratch and subsequently propagated an update adding a new daughter to one of the families. The same experiment was conducted analogously in the backward direction. The plots in Figure 11 depict our measurement results. The y-axis shows the median runtime (in seconds) of five repetitions executed on a standard computer (Mac OS X, 1.7 GHz Intel Core i7, 8 GB RAM) with Eclipse Neon, Java 8, and 4 GB available memory for the eMoflon and medini QVT solutions,

and GHC 8.0.1 for BiGUL. The x-axis shows the total number of all source and target model elements, ranging from about a thousand to a million elements (objects and links but not attributes). No measurement results are available for cases where a tool either runs out of memory or requires more than 5 minutes.

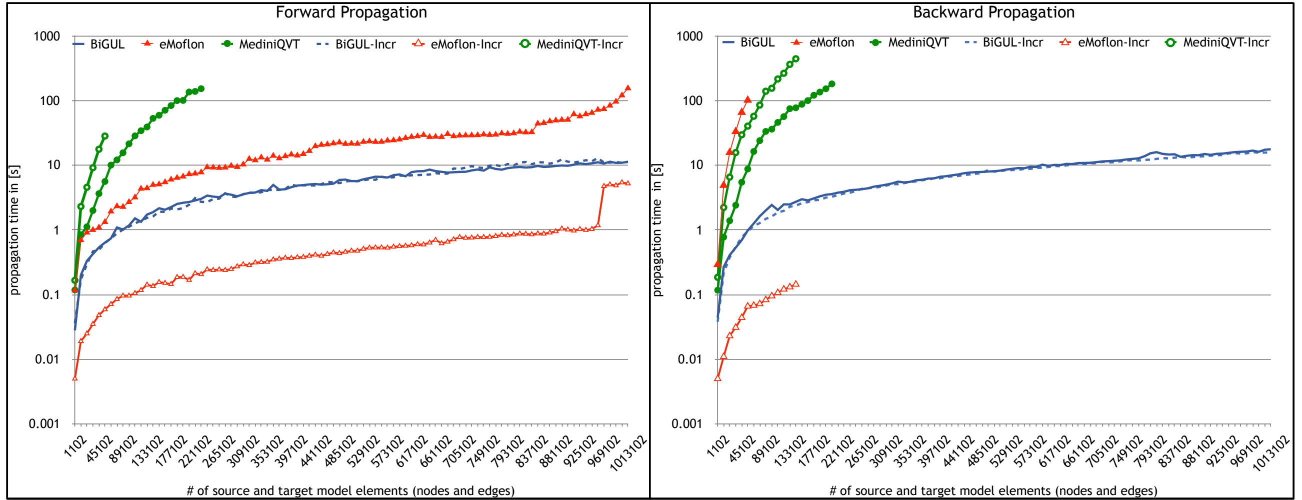


Figure 11: Runtime measurements for fwd (left) and bwd (right) direction

We answer in the following our performance-related research questions based on the measurement results:

RQ2.a: The initial transformation with BiGUL scales linearly with model size and remains under 20 seconds for all cases up to a million elements. Forward transformation with eMoflon remains under 30 seconds with linear scalability for up to about 900K model elements, but faces scalability issues for the largest case with about a million elements requiring 96s before running out of memory. In the backward direction, eMoflon reaches its limits already at about 60K elements. Medini QVT suffers from scalability issues in both directions and reaches its limits at 221K elements requiring more than 150 seconds for the forward as well as the backward direction.

RQ-2.b: Representing a state-based solution, runtime for an update propagation with BiGUL scales with model size (and not with update size) requiring almost the same runtime with the initial transformation in all cases. Update propagation with eMoflon takes under one second for all cases up to 900K elements showing that the update size is the decisive factor. Nevertheless, a relatively slow but steady linear growth in propagation time can be observed with increasing model size, especially when eMoflon starts running out of memory for more than 900K elements. Update propagation with medini QVT takes longer than the initial transformation, probably because correspondence information is handled as additional constraints. This indicates a critical dependency on model size for solutions depending on a costly vertical alignment.

RQ-2.c: BiGUL and medini QVT exhibit symmetrical behaviour with respect to their scalability. In contrast, the backward transformation with eMoflon is much slower than its forward transformation. This can be partly attributed to the price for enabling a flexible runtime update policy that requires maintaining *all* possible decisions (male to father or son, female to mother or daughter, in existing or new families).

To investigate **RQ-3**, we measured the size of F2P solutions developed with each individual tool. The most compact solution (in terms of pure typing effort) is with eMoflon, consisting of 192 lines of code (256 words, 3195 characters), distributed over 12 files. The decomposition into multiple files (one TGG rule per file) is suggested but optional; if a single file is used the solution is even more compact (as no imports are required): 168 lines (208 words, 1995 characters). The medini QVT solution consists of 144 lines (374 words, 4163 characters) in a single file, while the BiGUL solution consists of 181 lines (710 words, 6629 characters), again in a single file.

The first tool comparison based our proposed benchmarx framework indicates the heterogeneity of the bx tool landscape: Efficient batch transformation is provided by BiGUL, and to some extent, eMoflon, but they swap places when propagating updates. While BiGUL is reliable for a limited set of test cases, eMoflon, and to some extent medini QVT, strive to support a broader spectrum but can fail in some cases with limited support for fine grained control over consistency restoration. Specifications tend to be more compact in a declarative bx approach (eMoflon or medini QVT) as compared to a programming-based approach (BiGUL).

6 Conclusion and Future Work

The need for a benchmark for bidirectional transformations has been recognised for long in the bx community. The benchmarx framework presented in this paper provides an infrastructure that takes the heterogeneity of bx tools into account. The functional tool interface is based on a synchronisation dialogue; the representation of data is tool-specific rather than prescribed by the framework. The framework supports a zoo of tool architectures, including batch, incremental, state-based, delta-based and correspondence-based tools. As a pilot application, we selected the well-known Families2Persons case and implemented it in three tools with significantly differing architectures (BiGUL, eMoflon, and medini QVT). The F2P benchmarx provides a rich set of test cases which is organised systematically with the help of a feature model. The solutions developed with the tools mentioned above were evaluated with respect to both functionality and performance. This evaluation was performed primarily to obtain feedback concerning the benchmarx infrastructure, the design of the F2P benchmark, and the organisation of test cases according to the proposed feature model; we did not strive to optimise the solutions.

To disseminate, evaluate, and improve the benchmarx infrastructure, we are planning to elaborate the Families2Persons benchmark into a tool transformation case. Since this transformation problem is well known and may be implemented with acceptable effort, we hope that this case will be solved with a wide range of bx tools, ideally covering the entire spectrum of tool architectures. Furthermore, we intend to apply the benchmarx infrastructure to other bx problems to provide additional insights into the capabilities of bx languages and tools.

Acknowledgements

We would like to thank Robin Oppermann, Patrick Robrecht, Chandni Rikin Shah (Paderborn University), all participants of the benchmarx working group of the BX Shonan meeting 2017, and our anonymous reviewers.

References

[1] Bidirectional Transformations - NII Shonan Meeting Seminar 091. `http://shonan.nii.ac.jp/seminar/091/`. Date retrieved: February 28, 2017.

[2] Anthony Anjorin. FamilyToPersons v0.1 in Bx Examples Repository. `http://bx-community.wikidot.com/examples:home`. Date retrieved: February 28, 2017.

[3] 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.

[4] James Cheney, James McKinna, Perdita Stevens, and Jeremy Gibbons. Towards a Repository of Bx Examples. 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)*, volume 1133 of *CEUR Workshop Proceedings*, pages 87–91. CEUR-WS.org, 2014.

[5] Alcino Cunha and Ekkart Kindler, editors. *Proceedings of the 4th International Workshop on Bidirectional Transformations (BX 2015)*, volume 1396 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[6] Zinovy Diskin, Arif Wider, Hamid Gholizadeh, and Krzysztof Czarnecki. Towards a Rational Taxonomy for Increasingly Symmetric Model Synchronization. In Davide Di Ruscio and Dániel Varró, editors, *ICMT 2014*, volume 8568 of *LNCS*, pages 57–73. Springer, 2014.

[7] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *JOT*, 10:6:1–25, 2011.

[8] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-Based Classification of Bidirectional Transformation Approaches. *SoSyM*, pages 1–22, 2015.

[9] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-Oriented Domain analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990.

[10] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *WG 1994*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.