Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Benchmarking Bidirectional Transformations: Theory, Implementation, Application, and Assessment

Anthony Anjorin · Thomas Buchmann · Bernhard Westfechtel · Zinovy Diskin · Hsiang-Shang Ko · Romina Eramo · Georg Hinkel · Leila Samimi-Dehkordi · Albert Zündorf

Received: date / Accepted: date

Abstract Bidirectional transformations (bx) are relevant for a wide range of application domains. While bx problems may be solved with unidirectional languages and tools, maintaining separate implementations of forward and backward synchronizers with mutually consistent behavior can be difficult, laborious, and error-prone. To address the challenges involved in handling bx problems, dedicated languages and tools for bx have been developed. Due to their heterogeneity, however, the numerous and diverse approaches to bx are difficult to compare, with the consequence that fundamental differences and similarities are not yet well understood. This motivates the need for suitable benchmarks that facilitate the comparison of bx approaches.

This paper provides a comprehensive treatment of benchmarking bx, covering theory, implementation, application, and assessment. At the level of theory, we introduce a conceptual framework that defines and classifies architectures of bx tools. At the level of implementation, we describe *Benchmarx*, an infrastructure for benchmarking bx tools which is based on the conceptual framework. At the level of application, we report on a wide variety of solutions to the well-known Families-to-Persons benchmark, which were developed and compared with the help of Benchmarx. At the level of assessment, we reflect on the usefulness of the Benchmarx approach to benchmarking bx, based on the experiences gained from the Families-to-Persons benchmark.

Keywords Bidirectional transformation · Benchmark · Model synchronization · Framework

This paper is an extended version of work by Anjorin et al. [4]. We would like to thank Erhan Leblebici and our anonymous reviewers for their useful input and suggestions.

Anthony Anjorin
Paderborn University, Germany
E-mail: anthony.anjorin@upb.de

Thomas Buchmann
University of Bayreuth, Germany
E-mail: thomas.buchmann@uni-bayreuth.de

Bernhard Westfechtel
University of Bayreuth, Germany
E-mail: bernhard.westfechtel@uni-bayreuth.de

Zinovy Diskin
McMaster University, Hamilton, Ontario, Canada
E-mail: diskinz@mcmaster.ca

Hsiang-Shang Ko
National Institute of Informatics, Tokyo, Japan
E-mail: hsiang-shang@nii.ac.jp

Romina Eramo
University of L'Aquila, Italy
E-mail: romina.eramo@univaq.it

Georg Hinkel
Wiesbaden, Germany
E-mail: georg.hinkel@gmail.com

Leila Samimi-Dehkordi
MDSE Research Group, University of Isfahan, Iran
E-mail: samimi@eng.ui.ac.ir

Albert Zündorf
University of Kassel, Germany
E-mail: zuendorf@uni-kassel.de

1 Introduction

Bidirectional transformations (bx) are mechanisms for specifying and maintaining consistency between two or more artifacts.[1] Bx have been studied in many areas, including model-driven software development, graphical user interfaces, and transformations between heterogeneous data formats [12].

[1] In this paper we consider only the case of two artifacts.

While bx problems may be solved using unidirectional languages and tools, maintaining a separate implementation of different synchronizers with mutually consistent behavior can be difficult, laborious, and error-prone. In an attempt to better address bx-specific challenges, a variety of dedicated languages and tools for bx have been developed to assist software developers in solving bx problems more efficiently and reliably. Indeed, a wide spectrum of bx approaches has been reported in the literature, for example, by Hidaka et al. [26], who provide a detailed, feature-based classification of numerous bx approaches and tools.

Due to their substantial heterogeneity, however, bx tools are difficult to compare with the consequence that fundamental differences and commonalities are still not well understood. This situation hinders the development of better bx tools that combine the strengths and avoid the weaknesses of existing tools. To promote the understanding of bx languages and tools, therefore, the need for bx *benchmarks* was identified early [12].

As a first step towards this goal, a curated *repository* of *bx examples* was set up and is continuously being extended [10]. Subsequently, a proposal for additional requirements that a bx example must fulfill to become a bx benchmark was made [3]. Contrary to expectations, however, these preparatory activities and proposal did not result in any actual bx benchmarks. We claim this is due to additional and substantial *practical* challenges involved in setting up a bx benchmark that can be implemented with and executed for diverse bx tools.

This paper provides a comprehensive treatment of benchmarking bx, covering theory, implementation, application, and assessment. At the level of *theory*, we introduce a conceptual framework which defines and classifies architectures of bx tools in terms of their input and output data, the basic operations from which bx are composed, as well as their composition into different processing chains. The conceptual work is crucial to understand the landscape of bx tools and languages, and constitutes an essential prerequisite for developing an infrastructure for benchmarking heterogeneous bx tools.

At the level of *implementation*, we describe *Benchmarx* [4], an infrastructure for benchmarking bx tools based on the conceptual framework mentioned above. In particular, Benchmarx takes the heterogeneity of bx tools into account by abstracting from technological spaces, specific tool architectures, and the internal data maintained by the tools. A benchmark for a specific bx problem is implemented by providing a suite of test cases. A solution which is realized in a specific bx tool can be executed and evaluated using the provided test suite. Furthermore, the interpretation of test results is supported by identifying a set of tool features required by bx tests, thus allowing for a fine-grained classification into expected and unexpected passed and failed test cases. This enables a fair evaluation of specialized bx tools that can never pass all tests.

At the level of *application*, we report on a wide variety of solutions to the well-known *Families-to-Persons benchmark*, which were developed and compared with the help of the Benchmarx infrastructure. The Families-to-Persons benchmark, originally proposed as part of the ATL [33] transformation zoo,[2] involves synchronizing a database of families consisting of mother, father, daughters, and sons, with a database containing a set of unconnected male and female persons with birthdays that cannot be inferred from the families database. While being small and implementable with acceptable effort, this case poses a number of challenges to be addressed by bx tools implementing the case. To solicit a wide range of solutions, we submitted the Families-to-Persons benchmark as a case description to the Transformation Tool Contest 2017 (TTC 2017) [2]. For this paper, we selected seven solutions in significantly different bx tools: *BiGUL* [35], *BXtend* [6], *eMoflon* [38], *EVL+Strace* [46], *JTL* [11], *NMF* [29], and *SDMLib*.[3]

At the level of *assessment*, we reflect on the usefulness of the Benchmarx approach to benchmarking bx, based on comprehensive experiences gained from the Families-to-Persons benchmark. These experiences support the following claims:

Claim 1 *The Benchmarx infrastructure supports the implementation of benchmarks to be executed by heterogeneous bx tools, which differ with respect to technological spaces, tool architectures, bx paradigms, and bx languages.*

Claim 2 *The Benchmarx infrastructure is based on a conceptual framework that allows for a balanced interpretation of evaluation results, taking the heterogeneity of bx tools into account.*

Claim 3 *Comparing different solutions for the same benchmark problem assists in understanding the fundamental differences between bx approaches.*

The rest of this paper is structured as follows (Fig. 1): Section 2 defines the terminology used throughout this paper. Section 3 introduces the Families-to-Persons case, used as running example in all subsequent sections. Section 4 provides the conceptual foundations for our work.

[2] `http://www.eclipse.org/atl/atlTransformations/#Families2Persons`

[3] `www.sdmlib.org`

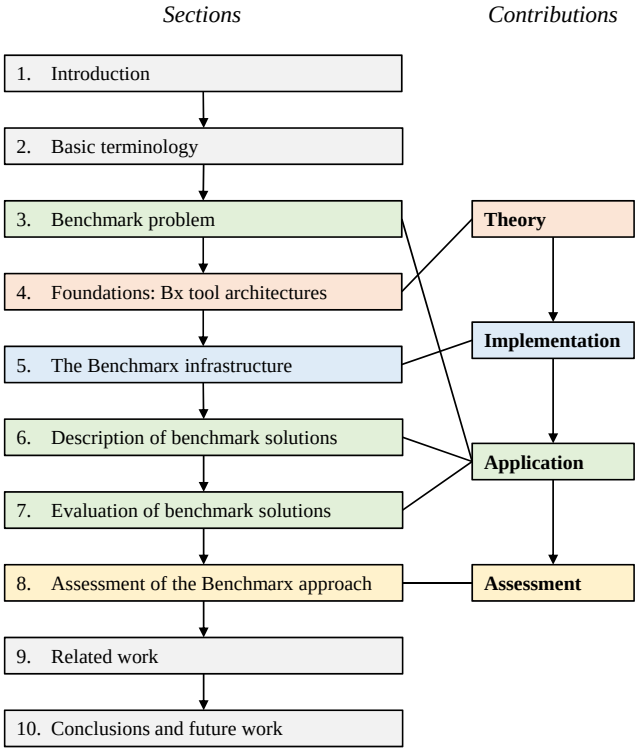


Fig. 1 Structure of the paper

Section 5 describes the design of the Benchmarx framework. The solutions to the Families-to-Persons benchmark are presented in Sect. 6, together with an introduction and classification of the bx tool used to implement each solution. A comparison and evaluation of these solutions follows in Sect. 7. An assessment of the Benchmarx approach is performed in Sect. 8. Section 9 discusses related work. Section 10 concludes the paper.

Figure 1 also maps the core sections of this paper to the levels of our contribution. Depending on research interests, the paper may be read either completely or selectively, focusing on specific sections. For example, readers interested in theory may focus on Sect. 4, readers who want to get an overview of the Benchmarx infrastructure and the value of using it may consult Sect. 5 and Sect. 8, and application-oriented readers may work through Sect. 3, Sect. 6, and Sect. 7.

2 Basic terminology

This section provides brief definitions for basic notions to be used throughout this paper. Additional, more specific terms will be introduced in detail in Sect. 4. All notions are collected together in a glossary (Appendix A). For a more comprehensive terminology and taxonomy for the domain of bx, the reader is referred to, e.g., Hidaka et al. [26].

2.1 Artifacts

Bidirectional transformations have been studied in a wide range of application domains. The *artifacts* manipulated by them include, e.g., data stored in databases, program data, and models. Throughout this paper, we will adopt terminology from model-driven software engineering [13] and refer to all artifacts as models.

A *model* is an abstraction of a system under study, which is more suitable than the system itself for certain purposes. A *metamodel* is a model that defines the structure of a set of models, i.e., a modeling language.

For metamodeling, we employ *Ecore* — which is similar to Essential MOF (EMOF), a subset of MOF [44], provided by the Eclipse Modeling Framework (EMF) [49]. Although we present the Families-to-Persons benchmark using Ecore (Sect. 3), bx tools do not have to be EMF-based to be able to implement the benchmark.

2.2 Transformations

A *transformation* reads, creates, or changes a set of $n \geq 1$ models. A *transformation definition* is a program that controls the execution of a transformation.

A *bidirectional transformation (bx)* is a transformation that maintains consistency between a source model and a target model. The terms *source model* and *target model* are used to distinguish between the models involved in a bidirectional transformation; they do not imply a specific transformation direction.

2.3 Synchronization

The act of executing a bidirectional transformation with the intent of establishing or maintaining consistency is denoted as *synchronization*. Synchronizations may be classified along different dimensions.

A *directed synchronization* operates on a *master* and a *dependent model*, and is said to maintain consistency *in the direction* of the dependent model. The master model is read, and the dependent model is created or changed to make it consistent with the master model. Required input for the synchronization can include the old master model, the exact changes applied to the master model, as well as preferences for making decisions during the synchronization process. A *forward synchronization* is a directed synchronization in the direction of the target model; a *backward synchronization* is performed in the opposite direction.

In contrast to a directed synchronization, both participating models have equal rights in a *concurrent syn-*

chronization. Thus, both source and target models may be changed to establish or maintain consistency.

A *batch synchronization* is a directed synchronization that creates the dependent model from scratch. In contrast, an *incremental synchronization* — which may be directed or concurrent — modifies an existing model.

A *view-based synchronization* is performed between a model and a *view* (an abstraction of the model that may be fully computed from the model). If neither model is a view of the other, the synchronization is *symmetric*.

An *interactive synchronization* is partially controlled by user interactions performed during the synchronization. An *automatic synchronization* can be run without any user interaction.

Finally, synchronization may be performed only *on-demand*, i.e., on explicit user request (e.g., via a dedicated synchronization command) or implicit user request (e.g., by saving a model to trigger synchronization implicitly). In contrast, *live synchronization* is performed immediately after each elementary change.

2.4 Consistency and bx laws

A pair of source and target models is *consistent* if both models agree on shared information. Consistency may be defined formally by a *consistency relation* — a relation that includes all pairs of related source and target models that satisfy some consistency condition. A consistency relation is *deterministic* in the *forward direction* if there is at most one consistent target model for a given source model; likewise for the *backward direction*.

A *bx law* is a condition on the behavior of bidirectional transformations. The notion of a bx law is very strong: A bx language/tool satisfies a bx law only if the law is guaranteed to hold for every bidirectional transformation written in the respective language and executed in the respective tool. A number of bx laws have been proposed in the literature. In the following, we mention a few that are particularly relevant in the context of this paper. The reader should note that bx laws may be specific to certain classes of bx approaches and cannot be sensibly applied to all other approaches.

A bidirectional transformation is *correct* if it produces consistent pairs of related models, and *hippocratic* if it does not change models which are already mutually consistent [50].

If the consistency relation is *non-deterministic*, the correctness property does not determine the result of a bidirectional transformation in a unique way. In such cases, the principles of least change and least surprise aim at further determining desirable synchronization behavior. A *least change* transformation [40] restores consistency such that the changed model has a minimal distance to the original model (with respect to a suitably defined metric). A *least surprise* transformation [9] behaves as closely as possible to users' expectations.

A *round-trip law* states a property that refers to a round trip of directed synchronizations performed in sequence. For view-based synchronization, for example, an immediate recreation of the view must have no effect after a view update has been propagated to the model, and writing back an unmodified view must not change the model [21].

There are two more properties which are relevant for this paper: A bidirectional transformation *terminates* if its execution halts after a finite number of steps. Finally, a bidirectional transformation is *complete* if it has a well-defined domain on which it may be successfully executed. Completeness is of practical relevance as some bx tools apply search-based heuristics that might not guarantee successful execution on all possible inputs.

2.5 Bx tools

A *bx tool* is a tool for managing bidirectional transformations. It may be based on a *bx language*, a domain-specific language for defining (programming) bidirectional transformations.

The *consistency relation* which a bx tool is supposed to maintain may be defined either explicitly or implicitly. An *explicit* definition may be given, e.g., by a set of *constraints* on pairs of related models [43], or by a *grammar* [48] generating all consistent pairs of related models. In contrast, an underlying consistency relation may be implied only *implicitly* by the steps required to establish or maintain consistency.

A bx tool may support different kinds of synchronization, according to the taxonomy introduced in Sect. 2.3. Synchronization may be *controlled* either *explicitly*, by programming the steps to be executed for restoring consistency, or *implicitly*, by deriving the steps automatically from other artifacts; forward (backward) synchronization may be derived from backward (forward) synchronization, or both directions may be derived from an explicit consistency relation.

For model synchronization, a bx tool must in some way process changes between (old and new) model versions. Here, the term *version* denotes a state of an evolving model at a specific point in time, defined by the model's contents. The general term *change* subsumes any modification of the state of a model. A change may be represented by a *delta*, i.e., a difference between two versions of the same model. Deltas may be classified further into *structural deltas*, which are defined in terms

of structural elements contained in both or only one of two model versions, and *operational deltas*, which are composed of sequences of change operations applied to transition from an old to a new version of some model.

A bx tool may provide *well-behavedness guarantees*, i.e., it may guarantee bx laws; see Sect. 2.4. Here, the term "guarantee" must be used with caution: The respective bx law must hold for each execution of each transformation definition. For example, a tool supporting view-based synchronization may guarantee roundtrip laws, either by composing the transformation from well-behaved bidirectional primitives [21], or by deriving one directed synchronization from its opposite [35].

Finally, bx tools may be classified according to their underlying architecture. The *architecture* of a *bx tool* is composed of (i) its external interface, defined by required inputs and provided outputs, and (ii) its internal processing, defined by processing steps and their organization; this is handled in detail in Sect. 4.

2.6 Benchmarks

Generally speaking, a *benchmark* is a standardized test that serves as a basis for comparison or evaluation.[4] More specifically, a *bx benchmark* is a benchmark tailored to the domain of bidirectional transformations. As mentioned already in the introduction (Sect. 1), our work targets comparison rather than evaluation, i.e., we are more interested in comparing bx tools and solutions implemented in these tools, rather than in an evaluation with the goal of identifying the "best" tool or solution. We use three criteria for comparison: *conciseness* (measured in terms of the size of a solution), *conformance* (measured in terms of failed and passed test cases), and *performance*, measured in the runtime required for executing scalability test cases.

3 The Families-to-Persons benchmark problem

In the Families-to-Persons benchmark, two related, but differently structured models have to be kept consistent: A *families model* with parents and children, and a *persons model* containing a flat set of males and females. Using the terminology introduced in the previous section, we present in this section the TTC 2017 variant of the Families-to-Persons case [2,54], which forms the basis of the benchmark reported in this paper.

[4] Merriam-Webster 2013

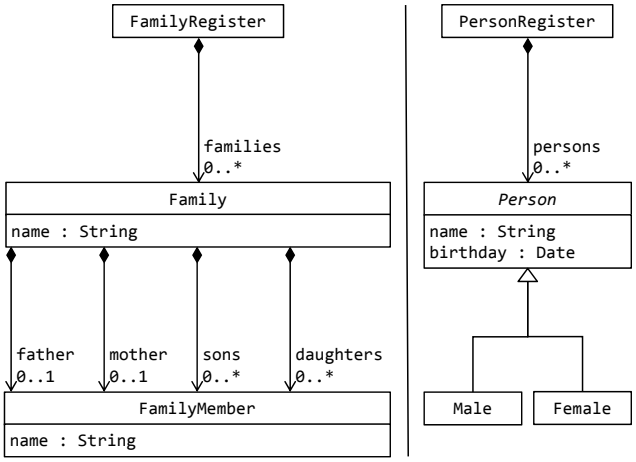


Fig. 2 Metamodels

3.1 Metamodels and consistency

Figure 2 depicts Ecore-based *metamodels* for families and persons models, respectively. Ecore-based bx tools implementing the benchmark may use these metamodels directly. As the Benchmarx framework hides models behind data abstraction interfaces, any equivalent definition suitable for an implementing tool may be used instead (e.g., algebraic data types in Haskell as required for BiGUL).

We assume a unique root in each model (a family and a person register, respectively). A family register stores an unordered collection of families. Each family has members who are distinguished by their roles. The metamodel permits at most one mother and at most one father as well as an arbitrary number of daughters and sons. A person register maintains a flat unordered collection of persons who have a birthday and are either male or female. Note that key (combinations of) properties cannot be assumed in either model: There may be multiple families with the same name, family members with the same name even within a single family, and multiple persons with the same name and even the same birthday.

A families model is *consistent* with a persons model if a bijective mapping between family members and persons can be established such that:

1. Mothers and daughters (fathers and sons) are paired with females (males).
2. The name of every person p is "$f.name$, $m.name$", where m is the member (in family f) paired with p.

An example of mutually consistent models, conforming to the metamodels in Fig. 2, is depicted as an object diagram in Fig. 3. The *consistency relation* between families models and persons models is *non-deterministic* in both directions: For a given families model, there are

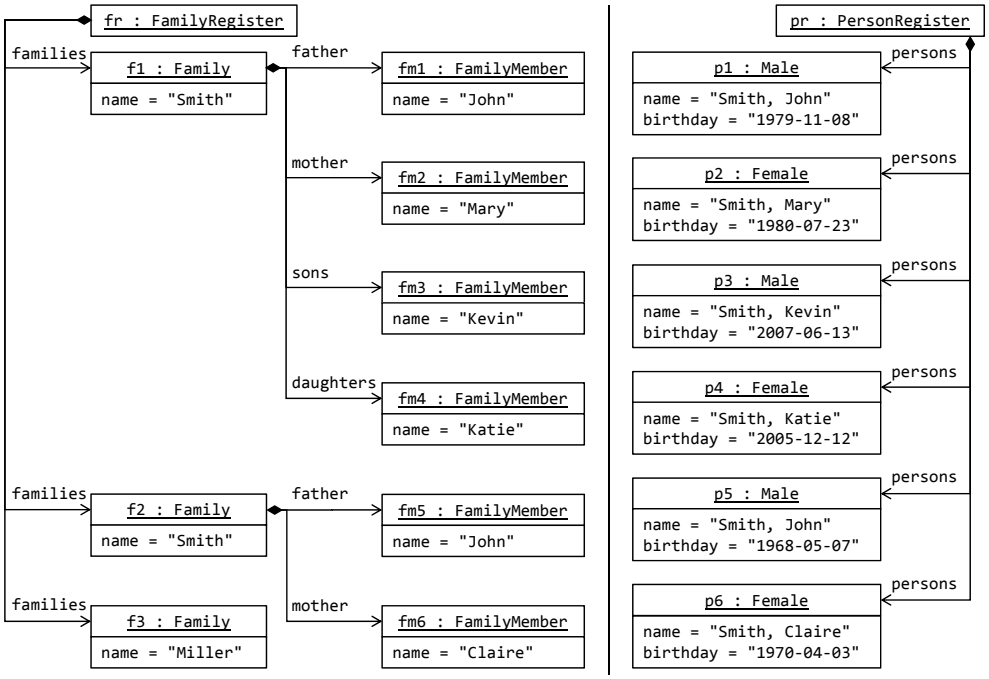


Fig. 3 Example of mutually consistent models

multiple consistent persons models (birthdays may be chosen arbitrarily). Conversely, a given persons model is consistent with multiple families models (due to different groupings into families and different roles, e.g., a female person can correspond to either a mother or daughter).

3.2 Synchronization

3.2.1 Properties of synchronization

The Families-to-Persons case is *symmetric* [15], i.e., neither model is a strict view of the other and information loss may occur in both transformation directions. We consider only *directed synchronization*, assuming that changes have been applied to the master model, while the dependent model has not been changed at all or the changes do not affect the consistency relation (e.g., changes of birthdays in the persons model are allowed).

We cover both *batch synchronization*, where the dependent model is created from scratch, and *incremental synchronization*, where the dependent model is changed to maintain consistency with the master model. Finally, changes are provided to the tools indirectly via an ordered[5] sequence of change operations referred to as *edits*. These properties describe how the benchmark is designed but do not, however, prescribe the architecture of a bx tool that can be used to implement the benchmark (see Sect. 4). For example, a bx tool can apply the provided edits and decide either to record operational deltas, structural deltas, or just to use only the old and new versions of the model.

[5]The results of some test cases depend on the order of elementary change operations.

3.2.2 Bx laws

For the purpose of benchmarking, all bx laws are considered as *properties* which may be satisfied or not. We test whether these laws hold for two reasons: First, a specific bx tool may not be designed to guarantee a certain bx law. Second, even if the tool has been designed to guarantee some law, its implementation may still contain faults, which can result in failing test cases.

From the bx laws introduced in Sect. 2.4, the benchmark takes *correctness* into account by the design of the test cases: All test cases check for consistency by requiring a dependent model that is consistent with the master model. Similarly, *termination* and *completeness* are considered implicitly, as a violation of these properties would result in failing test cases. Finally, a few test cases of the benchmark explicitly address *hippocraticness* by performing updates that do not affect consistency, respectively, to the master model.

Several variants of *round-trip laws* have been proposed in the literature. In the symmetric bx case considered in the Families-to-Persons benchmark, two round-trip laws are required to hold: If the backward (forward) transformation is executed immediately after the

forward (backward) transformation, the source model (target model) must not be changed. These laws are not tested explicitly because they are implied in the case of correct and hippocratic synchronizers.

In the design of all test cases, we generally assume *compositionality* of changes: The effect of the synchronization of a composite change is reduced to the composition of the effects of elementary changes. This corresponds to the *put-put law* for delta lenses, as introduced in [16]. Furthermore, we take the principles of *least change* and *least surprise* into account.

Unfortunately, the expected behavior of a synchronizer may not be derived in a unique way by applying bx laws. In particular, this applies to the backward transformation, which is highly non-deterministic. To resolve non-determinism (which complicates automatic testing and is often not desirable in practice), we describe the expected behavior explicitly and uniquely, taking bx laws into account. The description is informal, yet (hopefully) sufficiently precise (see below).

3.2.3 Synchronization behavior

In the following, we discuss the expected synchronization behavior for a series of "small" changes grouped according to model element (family, family member, person, ...) and then according to a limited set of change types (create, delete, update, move). This overview is sufficient to provide a high-level intuition for the benchmark. The actual test cases of course combine different changes and are more involved. However, the behavior of composite changes is *induced* by the behavior of elementary changes (see above).

Forward synchronization: In the forward direction, the persons model must be manipulated to be consistent with the changed families model. Defining the expected synchronization behavior is straightforward because the families model is determined uniquely, with the exception of birthday attributes. To resolve non-determinism, we require that a default value be assigned as the birthday of a newly created person. With this resolution, forward synchronization is deterministic.

Changes to *families* should be processed as follows:

Creation: New families can be created and must be inserted into the family register. This has no effect on the target model, which should not be changed.

Deletion: A family can be deleted together with all its family members. All persons corresponding to the deleted family members should be deleted.

Update: A family can be renamed. All persons corresponding to the family members in the renamed family should be renamed accordingly.

Move: Families cannot be moved as there is only a single family register and the collection of families is unordered.

Changes to *family members* should be processed in the following manner:

Creation: New family members can be created and must be immediately added to a family. A new person of the same gender as the family member should be created and added to the person register. The person's name should be appropriately composed from the family member's name and surname. The birthday of the person should have a default value (arbitrarily fixed by the benchmark).

Deletion: A family member can be deleted. The corresponding person in the person register should be deleted.

Update: A family member can be renamed. The corresponding person should be renamed accordingly.

Move: If a member is moved (defined as the combined deletion and creation of the link connecting the family member to a family), different cases have to be distinguished. If the gender of the family member is retained, the corresponding person object should be preserved; otherwise, the person should be deleted, and a new person with a different gender is created whose attributes are copied from the old person.
A 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.

Backward synchronization: In the backward direction, a person may be mapped either to a parent or a child, and persons may be grouped into families in different ways. As argued earlier, non-determinism should be resolved as far as possible. This may be achieved in different ways. A *default transformation* would fix specific mapping options (e.g., all persons may be mapped to children and grouped into the same family if their family names agree).

To provide for more flexibility, we decided to require a *configurable* backward synchronization, to be controlled by an *update policy*[6] that must set two Boolean parameters: (i) `preferParentToChild` controls whether a person is to be mapped to a parent or a child (if both options are possible), and (ii) `preferExistingToNewFamily` determines whether a person is to be mapped to a family member added to an existing family, or added to a newly created family containing only this single family member (again if both options are possible). If both parameters are set to true, the second parameter

[6] In practice this could either represent runtime user interaction or compile-time design preferences.

should take precedence: If the only existing family with a matching family name has no unoccupied parent role, the member is inserted into the family as a child (thus respecting (ii) and ignoring (i)).

It should be noted that the update policy does not resolve non-determinism completely. For example, let us assume that persons should be added to existing families as children. If we insert another person with family name `Smith` into the persons model depicted in Fig. 3, a corresponding member may be inserted either into family `f1` or into `f2`.

Furthermore, the update behavior may depend on the *order* of change operations. For example, if persons should be added to existing families as parents, and two male persons with family name `Miller` are added to the persons model, the first person will be added as a father and the second person as a son.

Let us now consider change operations on *persons*:

Creation: New persons can be created and must be added to the person register. A new family member with correct gender and name should be created in a suitable family in the family register. The update policy is consulted if it is possible to add the new family member to an existing family, and if the family member can be added as a parent or a child.

Deletion: Persons can be deleted. The corresponding family member should be deleted.

Update: Changes of birthdays do not affect the families model. The first name of a person can be changed; the name of the corresponding family member should be updated accordingly. The family name of a person can be changed; this change should not affect the current family and its members. The family preserves its name even if it does not contain any other members. The corresponding family member should instead be moved to another family, which may have to be created as required; the precise update behavior, if there are multiple possibilities, depends on the specified update policy for the particular test case.

Move: Persons cannot be moved because the persons model consists of a single, flat, unordered collection.

The update policy constitutes an example of *application-specific requirements*, as discussed at the end of Sect. 3.2.2. The rules for handling changes to family names are application-specific, as well. They are based on the underlying assumption that the person intends to leave their family (e.g., because of marriage). This kind of synchronization behavior may (arguably) be considered reasonable; we simply take it for granted, being required by an (imaginative) customer. However, in certain circumstances it may (arguably) violate principles such as least change (e.g., if the person was the single member of a family); we use these principles only as general guidelines, not as strict laws.

3.3 Challenges

The Families-to-Persons case includes a number of diverse *challenges* summarized in the following:

Heterogeneous metamodels: Solutions must establish a mapping between heterogeneous metamodels, where the same information is represented in different ways (concerning, e.g., names and genders).

Loss of information: The scenario is symmetric as the family structure is only present in the source, and birthdays are only present in the target.

No keys: There are no uniquely identifying (combinations of) properties for family members or persons, which makes synchronization difficult.

Non-determinism: The consistency relation is not deterministic: For a given families model, there can be multiple correct persons models (birthdays may be arbitrarily selected). Likewise, for a given persons model there can be multiple correct families models (due to different groupings into families and different roles in these families). Synchronization has to deal with this non-determinism (and resolve it as far as possible).

Configurability: The behavior of backward synchronization is controlled by an update policy determining roles of members and groupings of members into families. The update policy may be changed at runtime and should take effect only for future updates (there is to be no global reshuffling of the families model after the update policy has been changed).

Renaming and movement: Changes to be synchronized include not only creations and deletions, but also renamings and moves, which must not be reduced to deletions and creations so that changes can be synchronized in a minimally invasive way.

Order-dependent synchronization: Backward synchronization depends on the order in which change operations on the persons model are processed. For example, if two persons of the same gender are to be inserted into the same family as parents, only the first person can be inserted as a parent; the second person must be added as a child.

Specific least surprise requirements: The benchmark requires adherence to an application-specific definition of what "least surprise" is to mean. For example, if the family name of a person is changed, the corresponding family member should be moved to another family (rather than having the family name updated, with possible side effects on other

members). This definition of which change is "better" or "smaller" than another is arguably arbitrary and must be treated as an additional requirement.

4 Foundations: Bx tool architectures

In this section, we introduce the most important terms and notation required in the rest of the paper to describe and discuss the different bx approaches and solutions to our benchmark.

We first of all define the *input and output data* for bx tools in Sect. 4.1. Based on this, we identify *basic operations* applied by bx tools to maintain consistency in Sect. 4.2, and discuss how different combinations of expected input and produced output lead to a number of different *bx application scenarios* (Sect. 4.3). This finally allows us to describe and discuss a series of possible *bx tool architectures* in Sect. 4.4.

This overview of the complete range of possible architectures is used in Sect. 6 to assign the best fitting architecture to each solution and bx tool used to solve our benchmark. This helps to abstract from technical details and focus on the conceptual core strategy realized by each tool.

4.1 Input and output data

We refer to the input and output data expected and produced by a bx tool as *models* and *deltas*, representing the data to be kept consistent, and changes applied to models, respectively.

Definition 1 (Model)
A model, denoted by capital (primed) letters A, A', B, B', is a generic term used to refer to the data to be kept consistent by a bx tool.

We view the concrete representation of a model, e.g., a graph, a tree, a set, as being *internal* to a bx tool and thus not of primary relevance (for the goals and scope of this paper). Note that unprimed and primed letters such as A, A' indicate that these models are related in some way (e.g., are versions of the same original model derived by applying a sequence of changes).

Example 1 (Models)
An example of two models from the Families-to-Persons case is depicted in Fig. 3. These models are represented using typed, attributed graphs.

Definition 2 (Delta)
A delta, denoted by an arrow $\delta : A \to A'$ going from a "before" to an "after" model, is a generic term used to refer to the relation between two models, where these models are to be interpreted as being two versions (before and after) of the same model.

Again the concrete representation of a delta, e.g., a span of mappings or a change log, is internal to a bx tool and not of primary relevance. In order to precisely classify our tests in the benchmark, however, we distinguish between delta representations that record the *order* in which changes were made, and delta representations that only provide a structural mapping between two versions of a model:

Definition 3 (Operational delta, o-delta)
An operational delta, denoted by $(\delta) : A \overset{@}{\to} A'$, is a delta that represents the changes made to a model as an ordered sequence of applied change operations.

While the exact set of available change operations may vary depending on the chosen technological space, we assume the following set of change operations, which is typical in an MDE (graph-based) setting: (i) addition of elements (objects and links in a model), (ii) deletion of elements, (iii) attribute changes, and (iv) movement of objects. Note that our interpretation of "movement" corresponds to a change of the container relation (as defined by Ecore) of an object.

Definition 4 (Structural delta, s-delta)
A structural delta, denoted by $\langle\delta\rangle : A \Rightarrow A'$, is a delta that represents the changes made to a model as a purely structural mapping between the model and the version of the model after applying the changes.

As it is impossible to discern from an s-delta $\langle\delta\rangle$ the exact order in which change operations were applied, it can be represented as various, in this sense equivalent o-deltas $(\delta), (\delta'), ...$

Finally, test cases in the benchmark are implemented as a series of model *edits*, representing changes *to be applied*. It is helpful for conceptual clarity to differentiate edits from deltas. Edits can produce deltas when applied to models, but can also fail:

Definition 5 (Edit)
An edit, denoted by $@ : M \to \Delta$ is a partial function $@$ from a set of models M to a set of deltas Δ. If an edit is applicable to a model A, it can be applied to yield a delta δ, i.e., $@ : A \longmapsto \delta : A \to A'$.

Example 2 (Deltas and edits)
Given the persons model to the right of Fig. 3, an s-delta would be:

> *Added two female persons* `(Miller, Anne)` *and* `(Miller, Sandra)` to the persons register.

Representing the same change as an o-delta requires fixing the order in which these female persons were added to the person register, for instance:

> *Added two female persons to the register in the order* `[(Miller, Anne), (Miller, Sandra)]`.

As explained in Sect. 3, this distinction is required to pass some of the tests in the benchmark. In contrast, an edit would be, for example:

> *Attempt to delete the female person* `(Miller, Anne)` *from the register.*

This edit is not applicable to the register depicted to the right of Fig. 3, but would be applicable to the resulting register referenced by the deltas above.

Models and deltas are typically grouped into *domains* or *kinds*. We use the term model space to refer to this grouping:

Definition 6 (Model space)
A model space, denoted by $\mathcal{M} = (\Delta, M)$, consists of a set of deltas Δ between models taken from a set of models M. In the binary case, we denote one of the model spaces as the source model space: $\mathcal{M}_S = (\Delta_S, M_S)$, and the other as the target model space: $\mathcal{M}_T = (\Delta_T, M_T)$.

While deltas represent relations between models in the same model space, *correspondences* represent relations between models in different model spaces:

Definition 7 (Correspondence, corr)
A correspondence (or just corr), denoted by a bidirectional arrow $\sigma : A \leftrightarrow B$, refers to a relation between two models typically in different model spaces.

In order to support an intuitive understanding of the diagrams used to discuss the tool architectures in Sect. 4.2, deltas are often depicted visually as *vertical* single-headed arrows, while corrs are depicted visually as *horizontal* double-headed arrows. The models connected by a corr are typically interpreted as being in the same version, with respect to the relevant deltas being discussed. In many cases, it is conceptually helpful to regard some correspondences as being *diagonal* in the sense that they represent correspondences between models (in different model spaces) that are in different versions with respect to the relevant deltas being discussed:

Definition 8 (Diagonal, diag)
A diagonal (or just diag), denoted by $\sigma : A' \leftrightarrow B$, is a corr that is "diagonal" in the sense that it connects two models in different versions (A' is in an updated, new version, while B is in an old version) with respect to a set of deltas being discussed.

While arbitrary pairs of models can be connected in some way or another with a corr (or diag), not all corrs make sense for a given consistency management scenario. This is expressed by assuming an underlying *consistency relation* that can be used to classify corrs as being either consistent or not. This is summarized together with all previous definitions in the following:

Definition 9 (Triple space)
A triple space, denoted by the triple $(\mathcal{M}_S, \mathcal{M}_T, C \subseteq R)$, consists of a source model space $\mathcal{M}_S$, a target model space $\mathcal{M}_T$, a set of corrs R connecting models in the source and target model spaces, and a consistent subset of corrs $C \subseteq R$. A corr $c \in R$ is said to be consistent if and only if $c \in C$.

Model spaces can be formalized as *categories* with models as objects and deltas as arrows. If deltas are formalized as spans of structure-preserving mappings, then delta composition can be formalized as a pullback of spans. Similarly to deltas, corrs and diags can also be formalized as spans of structure-preserving mappings. The reader interested in a more rigorous handling of these terms is referred to e.g., Diskin et al. [16] and Anjorin [1] for further details.

Example 3 (Model spaces, corrs, and triple space)
The source (target) model space for the Families-to-Persons case consists of all models that are valid instances of the families (persons) metamodel depicted to the left (right) of Fig. 2, and all deltas between the models in each model space representing all possible combinations of the basic change operations (add/delete elements, attribute changes, object movements).

For the Families-to-Persons case, corrs can be represented by mappings between family members and persons. Consistent corrs are bijections that additionally satisfy the two constraints given in Sect. 3.1. For the pair of models depicted in Fig. 3, for example, a *consistent* corr would consist of mappings $fm\langle n \rangle \leftrightarrow p\langle n \rangle$ for $n \in \{1, ..., 6\}$. The resulting triple space for the benchmark consists of these two model spaces, the set R of all possible corrs (mappings between family members and persons), and the consistent subset $C \in R$ of bijections that satisfy the two additional constraints.

We use triple spaces to abstract from exactly how the underlying consistency relation is specified or checked by concrete bx tools. To achieve a high-level but still useful abstraction for *consistency management* strategies, we now discuss a few basic consistency management operations in the following.

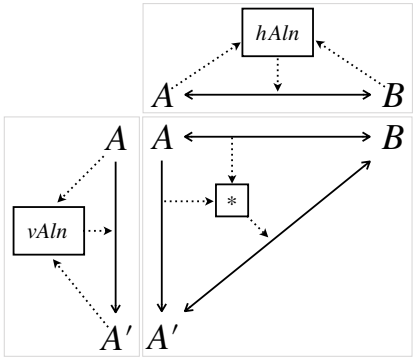


Fig. 4 Overview of auxiliary operations

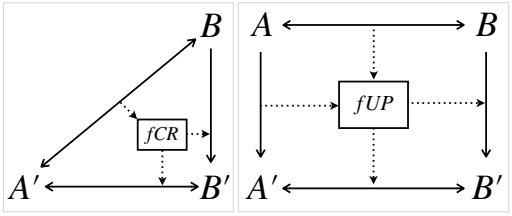


Fig. 5 Overview of basic operations

4.2 Basic operations

While we do not aim to describe and discuss in detail exactly how bx tools internally manage consistency, we claim that there is actually only a small set of basic operations, which are applied and combined in different ways to realize different bx tool *architectures*. These architectures can be used to classify bx tools and characterize fundamental differences and similarities.

Figure 4 provides an overview of three auxiliary operations showing their arity and indicating input and output parameters. Using the same notation as Def. 1, 2, and 7, models are depicted as (primed) capital letters, deltas as vertical single-headed arrows, and corrs (diags) as horizontal (diagonal) double-headed arrows. The three auxiliary operations are depicted as rectangles, connected to their input via incoming, and to their output via outgoing dotted single-headed arrows. These operations are now explained in the following by providing concrete examples taken from our Families-to-Persons benchmark:

Horizontal Alignment (hAln) takes two models A and B in different model spaces and determines a corr $A \leftrightarrow B$ between them. It is in general impossible to recover unique corrs and *hAln* is often heuristic-based. For the Families-to-Persons case, a simple strategy would be to pair female(male) persons and family members with the right gender and same family name and first name according to convention. The pair of models depicted in Fig. 3, however, shows that this is not sufficient to obtain *the* unique correspondence when there are multiple persons with the same name in the person register.

Vertical Alignment (vAln) takes two models A and A' in the same model space and determines a delta $A \rightarrow A'$. This is also non-deterministic in general as there can be multiple deltas going from A to A'. Consider, for example, renaming a person `Smith, Mary` to `Smith, Anne`, as opposed to deleting `Smith, Mary` and adding a new person `Smith, Anne` with the same birthday. It is impossible to decide what *the* delta is in this case, and *vAln* must apply suitable heuristics to make a choice.

Re-Alignment (∗) takes a coinciding corr and delta and produces a diag by combining them. The idea is that two models A and B were previously aligned via $A \leftrightarrow B$. Then A was changed to A' yielding the delta $A \rightarrow A'$. The task of $*$ is now to "re-align" the new A' with B. Re-alignment is often a straightforward task of updating the corr, removing obsolete (invalid) mappings due to deleted (changed) elements in A, as well as adding any new mappings from added (changed) elements in A to existing elements in B. Although we introduce $*$ as depicted in Fig. 4, we shall also use variants of $*$ with different input/output roles for the parameters, e.g., taking the diag as input and producing corr and delta as output. For simplicity, we shall also refer to these operations as re-alignment, indicating clearly in the diagrams which parameters are taken as input and which are produced as output.

Figure 5 depicts two basic operations that can be used to maintain consistency: either by *restoring* consistency by suitably manipulating the dependent model (left), or by *propagating* deltas from the master to the dependent model (right). Appropriately combined with the auxiliary operations discussed previously, a wide range of different application scenarios can be addressed by these two basic operations that are discussed in the following:

Forward Consistency Restoration (fCR) takes as input a diag $A' \leftrightarrow B$ and determines a delta $B \rightarrow B'$ and a *consistent* corr $A' \leftrightarrow B'$; backward consistency restoration (*bCR*) is defined analogously. This operation obviously requires some knowledge of the underlying consistency relation. The input diag is typically not yet consistent and some strategy of how to restore consistency by changing B in some appropriate manner must be implemented. For example, if an unmapped family member in A' and an unmapped person in B are found, a strategy to restore consistency would be to rename the person

in B' so that a consistent corr can be reestablished. Consistency restoration is often non-deterministic, hence the need for laws to characterize "desirable" behavior.

Forward Update Propagation (fUP) differs from *fCR*: Instead of restoring consistency to an already inconsistent diag, *fUP* takes as input a (typically consistent) corr $A \leftrightarrow B$ and a delta $A \rightarrow A'$. To decide how to maintain consistency, *fUP* inspects the input delta $A \rightarrow A'$ and *propagates* the delta to B, yielding an output delta $B \rightarrow B'$ as well as a consistent corr $A' \leftrightarrow B'$. For example, adding a new family member to A to yield A' can be propagated to adding a suitably named new person to B to yield B', and adding a mapping between the new family member and person in $A' \leftrightarrow B'$. Backward update propagation (*bUP*) is defined analogously. Similar to consistency restoration, update propagation is also often non-deterministic.

4.3 Input-based application scenarios

In this section, we now discuss 11 different application scenarios according to the provided input data in each case. If certain input data, e.g., the input delta, is available or not is often an intrinsic part of an application scenario. Consider, for example, an application scenario in which models are manipulated without any means of tracking exactly which changes were made. If this is part of the requirements (perhaps due to an existing process workflow at a company) and cannot be influenced, then input deltas will not be available for consistency management. To classify all possible application scenarios, Table 1 depicts a grid distinguishing horizontal and vertical input data to yield 11 possible combinations.

The possibilities for vertical input data range from just the changed source model A' (initial-), to the old and changed source models A, A' (state-), to the input delta $A \rightarrow A'$ (delta-). While the terms *state* and *delta* should be natural, *initial* deserves some explanation. If a bx tool is supplied the changed model A' as vertical input, it can only reasonably assume that the implied input delta is $\emptyset \rightarrow A'$, i.e., the model A' has been created from scratch (even if this is not the case). The empty model $\emptyset$ is often referred to as the *initial object* and the unique existence of $\emptyset \rightarrow A'$ is required.

The possibilities for horizontal input data range from nothing (batch-), to just the previous target model B (state-), to the corr between previous source and target models $A \leftrightarrow B$, to the diag between the changed source model and previous target model $A' \leftrightarrow B$. All

Table 1 Overview of bx application scenarios

		batch-	state-	corr-	diag-
		Nothing	B	$A \longleftrightarrow B$	$A' \longleftrightarrow B$
initial-	A'	initial-batch-based	initial-state-based		initial-diag-based
state-	A, A'	state-batch-based	state-state-based	state-corr-based	state-diag-based
delta-	$A \downarrow A'$	delta-batch-based	delta-state-based	delta-corr-based	delta-diag-based

combinations of vertical and horizontal input are theoretically possible, apart from initial-corr-based, which would be contradictory (the corr requires the presence of A, which is not available). Compared to the commonly used but imprecise terms *state-based* or *delta-based*, our classification schema allows for a fine-granular distinction between, e.g., *state-state-based*, *delta-state-based*, and *state-corr-based*.

From an inspection of all bx tools we are aware of, it is our observation that tool architectures tend to be either restoration-based (i.e., using basic operations *fCR*, *bCR*) or propagation-based (i.e., using basic operations *fUP* and *bUP*), but not both. A plausible explanation is that implementing both *fCR* and *fUP* would be redundant as both serve the same purpose of consistency maintenance. All other basic operations are auxiliary in nature and help compensate a mismatch between application scenario and tool architecture.

Such a mismatch stems from the fact that bx tool developers choose a certain bx tool architecture corresponding to one of the 11 application scenarios, i.e., to their expectations concerning what input data is available and what not. If a bx tool expects more input data than is available, it can only be used in combination with external alignment operations, which are required to recover the missing data. This situation can be problematic as alignment can be imprecise, inefficient, and difficult to implement. Faced with this overfitting mismatch, users of such a bx tool tend to view the tool as being heavy-weight, having unrealistic (utopic) expectations, and thus not being worth the effort if one has to implement all required additional auxiliary alignment operations.

If a bx tool requires less input data than is available, the additional input data is simply ignored. While the required alignment is either completely internalized, e.g., via the use of in-built heuristics and optimization, or is well-supported with an appropriate language or API, our benchmark shows that this situation is still undesirable as it leads to unexpected results in general. Faced with this underfitting mismatch, users of such a

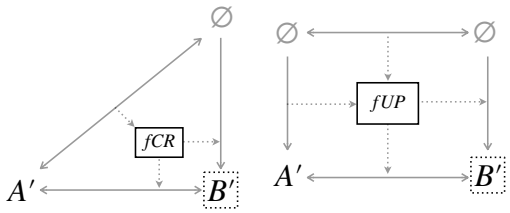


Fig. 6 Initial-batch-based architectures

bx tool tend to view the tool as being a lightweight, but often unpredictable, unconfigurable black-box.

Identifying the underlying architecture of a bx tool helps to understand its limitations and compare it to other bx tools. To support this architectural classification, we now discuss possible restoration-based and propagation-based tool architectures for each of our 11 application scenarios.

4.4 Bx tool architectures

Figure 6 depicts bx tool architectures for the simplest application scenario, namely *initial-batch-based*. As this application scenario can actually be handled with a standard unidirectional model transformation, the architectures are not necessarily practical, but show how a bx tool can handle simple cases as degenerate versions of the more general case of consistency maintenance. First some words on the notation used in all architecture diagrams: the same notation for models, deltas, corrs, diags, and basic operations is used as in Fig. 4 and 5. In addition, all objects that are internally computed by the bx tool and typically not presented to the user are grayed out. Finally, output data is distinguished from input data via a dotted border. To improve readability, the restoration-based architectures for every application scenario are depicted to the left of the figures, while the propagation-based architectures are depicted to the right.

To apply *fCR* in an *initial-batch-based* application scenario, the empty diag is assumed and provided as additional input to produce $\emptyset \rightarrow B'$ and $A' \leftrightarrow B'$. In this scenario, only B' is typically required as output, although a bx tool would (internally) compute a corr as well. Similarly, *fUP* can be applied by assuming and providing the empty corr $\emptyset \leftrightarrow \emptyset$ and $\emptyset \rightarrow A'$. Note that both architectures do not require any auxiliary operations as the implied empty diag, corr, and delta are typically trivial to compute. In practice, however, (mis)using a bx tool for this simple application scenario can be much more inefficient than a standard transformation, and we are thus not aware of any dedicated bx tool that addresses solely this application scenario.

Figure 7 depicts three architectures addressing the more common bx application scenario *initial-state-based*. In this scenario, the previous output model B is additionally provided as input, so that the bx tool has the chance of preserving as much information as possible in B while maintaining consistency. The conceptually simplest architecture makes use of *fCR* by computing a diag between A' and B using *hAln*. If both *bCR* and *fCR* are available, *vAln* can be used in place of *hAln*, by first determining $A \leftrightarrow B$, then $A \rightarrow A'$ via *vAln*, and then the required diag $A' \leftrightarrow B$ with $*$. While this might sound more complex, it can make sense if all necessary operations are available, while *hAln* is not. A propagation-based architecture for this scenario (depicted to the right of Fig. 7) can use *bUP* to determine $A \leftrightarrow B$ and *vAln* to establish the required input delta $A \rightarrow A'$ for *fUP*. If *bUP* or *vAln* is unavailable, an alternative propagation-based architecture (not shown explicitly as a diagram) can use *hAln* to compute the diag $A' \leftrightarrow B$, then the inverse of $*$ to determine the corr $A \leftrightarrow B$ and delta $A \rightarrow A'$ for *fUP*.

Figure 8 depicts architectures addressing the *initial-diag-based* application scenario. This represents a perfect fit for a restoration-based architecture, as *fCR* can be directly applied without requiring any auxiliary operations. It is also of practical relevance, as A is often destructively changed to A' in many application scenarios, leaving $A' \leftrightarrow B$ naturally as the left-over corr. Note that the typical output for this scenario is the updated, consistent corr $A' \leftrightarrow B'$. While it is theoretically possible to address initial-diag-based scenarios with a propagation-based architecture, Fig. 8 indicates why this is awkward in comparison: to apply *fUP*, one would have to determine $A \leftrightarrow B$ and $A \rightarrow A'$ from

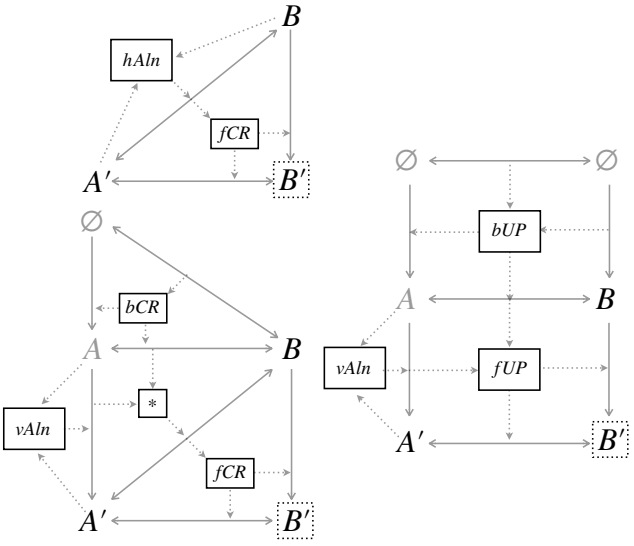


Fig. 7 Initial-state-based architectures

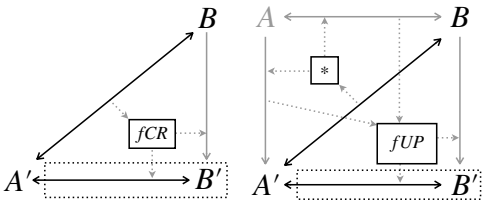


Fig. 8 Initial-diag-based architectures

the diag $A' \leftrightarrow B$, a task in itself already comparable in complexity to *fCR/bCR*.

Figure 9 depicts architectures addressing the *state-batch-based* application scenario. In all cases, A has to be extended to $A \leftrightarrow B$. *fCR* can then be used either by combining *vAln* and $*$, or by using *hAln* to directly determine $A' \leftrightarrow B$. The propagation-based architecture uses *vAln* to determine the missing input delta for *fUP*. The complexity of all three architectures indicates that a state-batch-based application scenario can be considered relatively unfavorable compared to the initial-state-based case.

Figure 10 depicts architectures addressing the *state-state-based* application scenario. In both cases, *hAln* and *vAln* are required to recover the corr $A \leftrightarrow B$ and input delta $A \rightarrow A'$. While this already constitutes the input for *fUP*, the restoration-based architecture additionally requires $*$ to determine the diag for *fCR*.

The architectures depicted in Fig. 11 for the *state-corr-based* application scenario are fairly analogous to the state-state-based case. The only difference and simplification is that the required corr is supplied as input and *hAln* is therefore not needed.

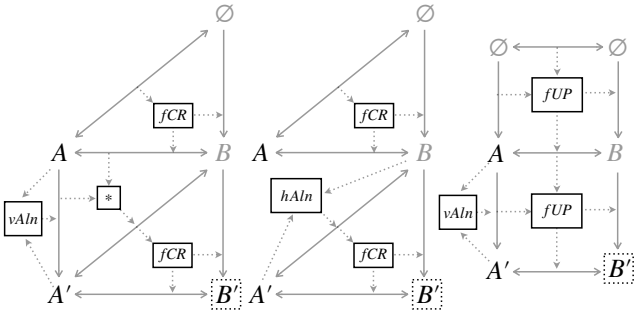


Fig. 9 State-batch-based architectures

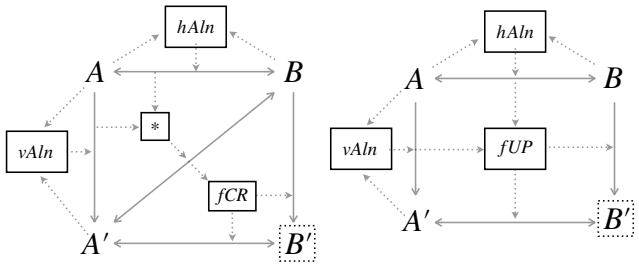


Fig. 10 State-state-based architectures

Figure 12 depicts architectures for *state-diag-based* and for *delta-diag-based* application scenarios. As the required input diag for *fCR* is directly supplied, providing extra input does not seem reasonable and we thus do not suggest any restoration-based architectures. Even the propagation-based architectures appear awkward: To make use of all provided input in the state-diag-based case, either the delta $A \rightarrow A'$ must be determined with *vAln* (depicted to the left of Fig. 12), or the corr $A \leftrightarrow B$ with *hAln* (not depicted in the figure). Taking this delta (or corr) and the supplied diag as input, $*$ can then be used to compute the required corr (or delta), so that *fUP* can be used. The propagation-based architecture for the delta-diag-based case is analogous, only simplified as the input delta is supplied and does not need to be computed with *vAln*.

Figure 13 depicts the architectures for the *delta-batch-based* application scenario. These are simplified versions of architectures for the state-batch-based case, as the required input delta is now provided and does not need to be computed via *vAln*. Note that both

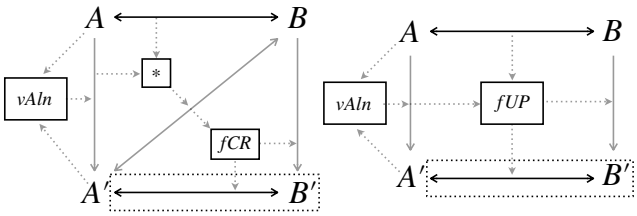


Fig. 11 State-corr-based architectures

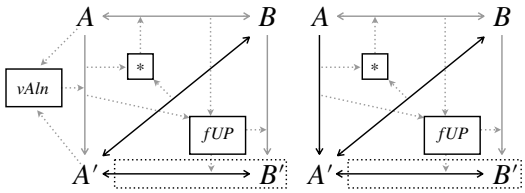


Fig. 12 State- and delta-diag-based architectures

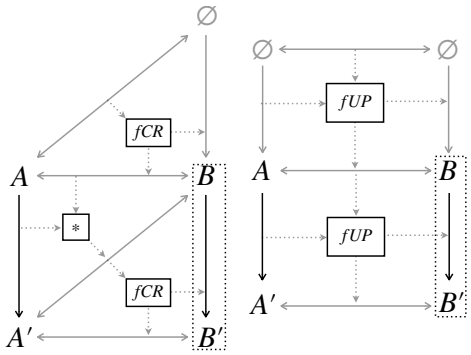


Fig. 13 Delta-batch-based architectures

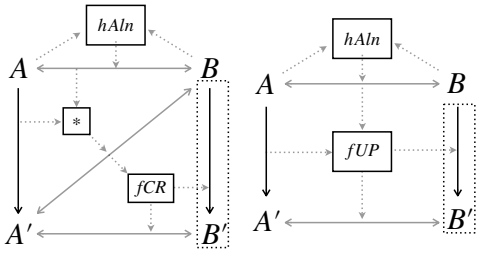


Fig. 14 Delta-state-based architectures

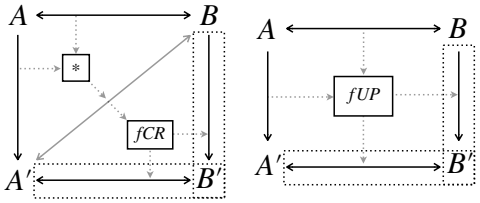


Fig. 15 Delta-corr-based architectures

architectures neither require auxiliary alignment, nor *bCR* (*bUP*) for consistency restoration (update propagation).

The architectures depicted in Fig. 14 are for the *delta-state-based* application scenario. As the previous target model B is provided in this scenario, the architectures can use *hAln* to determine the corr $A \leftrightarrow B$ and can make do without a second application of *fCR* (*fUP*) as in the delta-batch-based case.

Finally, Fig. 15 depicts the architectures for the *delta-corr-based* application scenario. The diagrams indicate that this is the ideal case for applying *fUP* as exactly what is required is available as input. While the restoration-based architecture requires realignment via $*$, this is usually a simple operation in this case.

4.5 Classifying bx tools

Reflecting the 11 application scenarios and possible bx tool architectures discussed in detail in previous paragraphs, the variability of bx tool architectures is represented as a feature model in Fig. 16, extended to cover some additional features of bx tools. Our goal in this paper is not to suggest a further exhaustive feature model for bx approaches such as Hidaka et al. [26], but more to make intensive use of a minimal set of core features to classify the different bx tools used to solve our bx benchmark.

Using standard feature model notation, features are denoted as rectangles; a gray fill denotes *abstract* features that are only used to group other features. Circles with a black fill indicate mandatory children, circles without a fill optional children. Children connected with an angle without a fill are exclusively ored, while children connected with an angle with a black fill are ored. Features that exclude each other are connected with a dashed, double-headed (red) arrow; A dashed, single-headed (green) arrow from a feature f to another feature g denotes that f requires (implies) g.

A bx tool must have a bx tool architecture, developed in a specific style (either restoration-based using *fCR/bCR* or propagation-based using *fUP/bUP*), and addressing a specific application scenario according to expected input and provided output. Concerning horizontal input, a scenario can exclusively require no input at all, the previous output model (state-based), a diag (diag-based), or a corr (corr-based) as input. In contrast, vertical input is a mandatory feature, forcing an exclusive choice between requiring just the input model (initial-based), the previous and changed input models (state-based), and an input delta (delta-based). In practice, it is also informative to know if a bx tool expects an s-delta or o-delta. Finally, as explained previously, requiring a corr and only the input model (initial) is contradictory, hence the exclusion between these two features.

A major goal of bx languages and tools is often to provide some added value compared to implementing synchronization using standard unidirectional transformation languages. While this goal can be realized as an attempt to increase productivity, it is often also realized as an attempt to increase the *quality* of implemented synchronization solutions. In Fig. 16, therefore, we include an optional feature *well-behavedness guarantees*, which can include a combination of correctness, hippocraticness, round-trip laws, termination, and completeness (totality) guarantees.

An important, hence mandatory feature of a bx tool is its underlying *consistency relation*. This has to be specified in some manner, either *implicitly*, e.g., by providing implementations of *fCR/bCR* (*fUP/bUP*) combined with round-trip laws, or *explicitly*, by specifying a set of constraints, providing a grammar (language membership as consistency check), implementing a consistency checker in some suitable (programming) language, or via a combination of these (partial) specifications, e.g., grammar-based for structural parts of the model and constraint-based for attribute values. Correctness can only be guaranteed with respect to the underlying consistency relation, so this only makes sense as a feature if the relation is explicitly provided to the tool, hence the requirement between these features.

A final, core feature of a bx tool is synchronization and we have observed that this can be either controlled *implicitly*, e.g., based on the underlying consistency relation and some governing laws the bx tool automatically derives *fCR/bCR* or *fUP/bUP* as required,

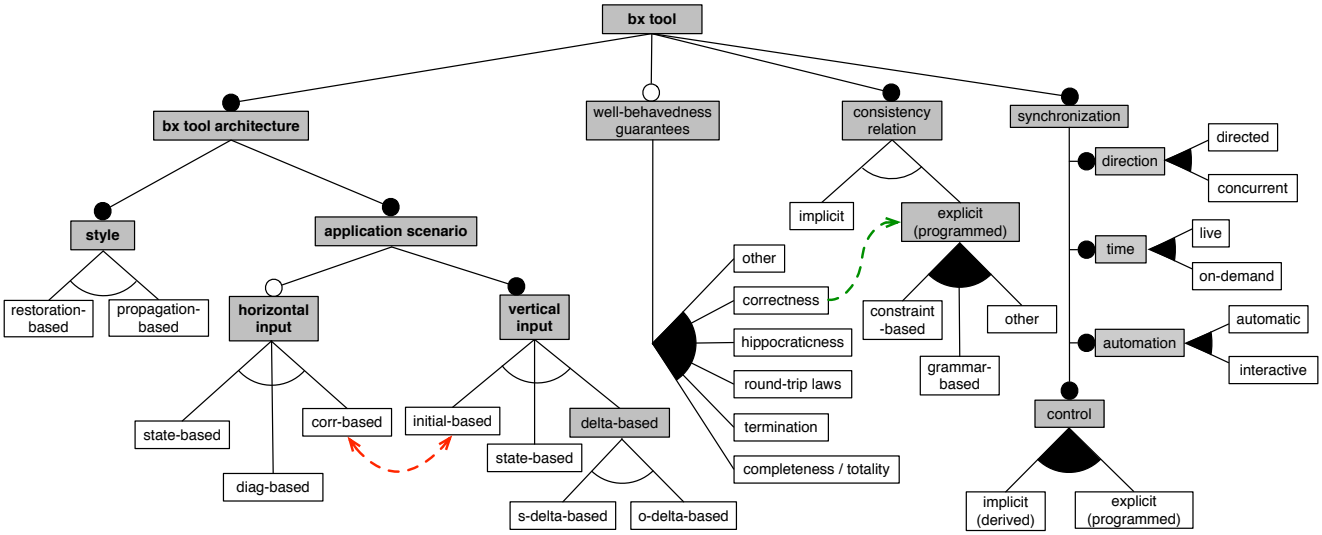


Fig. 16 Bx tool variability as a feature model

or *explicitly*, e.g., the bx tool requires implementations of *fCR/bCR* or *fUP/bUP*, and perhaps performs some checking for compatibility of these implementations. Some bx tools allow a mixture of implicit and explicit, e.g., requiring an implementation of *fCR* (*fUP*) and automatically deriving a compatible *bCR* (*bUP*), or vice-versa. Although most bx tools are currently limited to directed synchronization, a few tools already attempt to address the more general task of *concurrent* synchronization. A bx tool can be designed for *on-demand* synchronization, allowing a user (or arbitrary component) to decide exactly when to synchronize, or for *live* synchronization deciding itself when "atomic" deltas must be propagated. Synchronization with a bx tool is *interactive* (*automatic*) if the tool is (not) specially designed to incorporate user interaction at runtime.

5 The Benchmarx framework

The Benchmarx framework is a component-based framework that enables a comparison of bx tools. A major challenge addressed by the framework is that different bx tools may require different input data. The solution adopted by the Benchmarx framework is to provide a unifying design space, in which different bx tool architectures can be placed, classified, and evaluated.

While the general conceptual design of the Benchmarx framework can be transferred to any technological space, we provide a reference implementation based on Eclipse, the Eclipse Modeling Framework (EMF), JUnit as a unit testing framework, and Java. To ensure that non-EMF and even non-JVM-based bx tools can be integrated with no restrictions on the input data of the tools, we use a string representation of produced and expected models established by convention for each benchmark example. The current solutions to the Families-to-Persons case show that it is indeed possible to integrate diverse bx tools with reasonable effort.

5.1 Design of a Benchmarx test suite

Figure 17 depicts a feature model for Benchmarx test cases. Every Benchmarx test case must state the relevant application scenario (cf. Fig. 16), its direction to be (exclusively) *fwd* (forward), *bwd* (backward) or a mix of both, i.e., a *round trip*, if the test case requires *runtime configuration* or not, and the combination of different change types applied in the test. The set of possible change types can be extended in the future to accommodate more expressive frameworks.

Benchmarx is designed as a generic framework for benchmarking bx tools; we thus strive to minimize requirements regarding the typically tool-specific representation of deltas and corrs. Instead of establishing some standard data structure for deltas and corrs, test cases are designed as *synchronization dialogs*. A synchronization dialog always starts from the same agreed

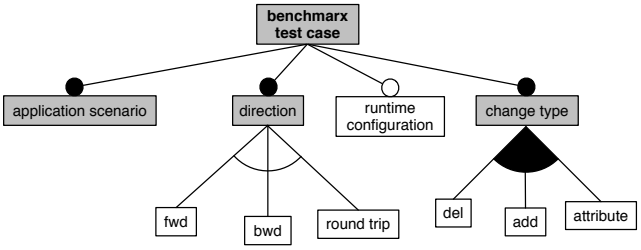


Fig. 17 Variability of Benchmarx test cases

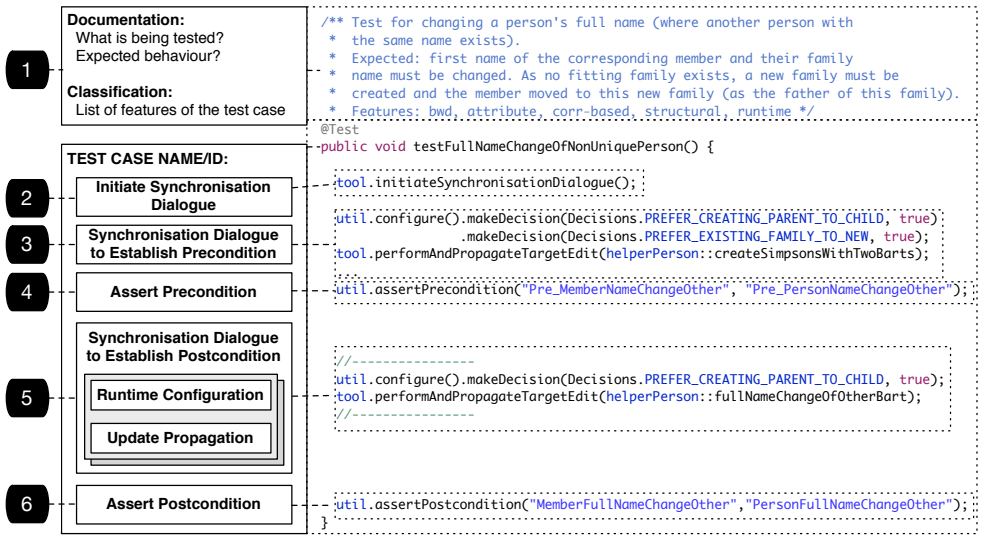


Fig. 18 A Benchmarx test case as a synchronization dialog

upon initial consistent state, and then applies a sequence of edits to the source and target models. Only the resulting models are directly asserted by comparing them with expected versions. In this manner, each bx tool is free to maintain arbitrary internal state, e.g., recording deltas during edit application, or maintaining corrs in whatever format is required.

A Benchmarx test case is depicted schematically to the left of Fig. 18, with a concrete test case for our example depicted to the right of Fig. 18, following the proposed schema using JavaDoc, Java, and JUnit as implementation technologies. Each test case contains a documentation (cf. Label 1 in Fig. 18) stating (i) what is being tested, (ii) the expected behavior, and (iii) a list of the concrete features of the test case taken from Fig. 17 to clarify at a glance if a given bx tool can be expected to pass the test or not, i.e., if the relevant application scenario for the test case matches the addressed application scenario of the (implemented solution with the) bx tool.

The test case itself starts by initializing the supplied bx tool (Label 2); the agreed upon starting state is hereby established (e.g., for the Families-to-Persons benchmark this comprises a single empty family register and a corresponding single empty person register), and all necessary internal (auxiliary) tool-specific data structures can be created at this point.

The subsequent part of a test case (Label 3) consists of a series of propagation steps, used to establish the precondition of the test. Although this creates a dependency on other tests (asserting exactly this precondition), it allows us to abstract from any tool-specific corr representation, as every bx tool can build up whatever internal state it requires along with the precondition. In this manner, the input corr is "passed" implicitly to the bx tool via a series of preparatory propagation steps. Each propagation step is specified in the test case as a source or target edit (realized as a Java lambda expression), which is passed to the bx tool and is to be performed on the source or target model. In this manner, the bx tool can record[7] whatever it wants when applying the edit to produce, e.g., a tool-specific representation of an o-delta or s-delta.

The precondition is finally asserted (Label 4), representing a well-defined starting point for the test. If the test requires a runtime update policy (cf. Sect. 1), this is configured (Label 5) just before propagating the relevant input edit. A sequence of such (optional) configuration and edit propagation steps form the primary synchronization dialog used to establish the postcondition (Label 5). The final 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 test case depicted to the right of Fig. 18, a number of persons are created in the person register and then propagated backward 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 propagated backward with a runtime update policy set to prefer creating parents (if possible) to creating children. Note that in this particular test, two persons with the name *Bart Simpson* are created as part of the precondition (indicated by the target edit `createSimpsonsWithTwoBarts`). As a consequence, this test can only be reliably passed if corrs

[7] EMF supports this via a notification framework.

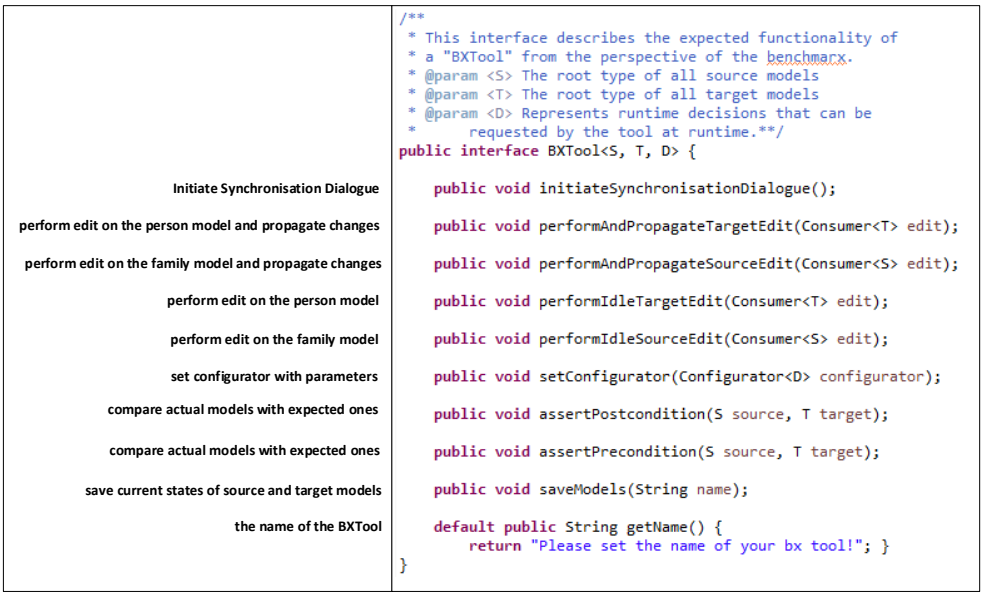


Fig. 19 The BXTool interface

are exploited as explicit input (indicated by the feature `corr-based` in the classification of the test).

The current test suite provided for the Families-to-Persons case consists of such test cases as in Fig. 18 separated into two broad categories: (1) *batch* test cases providing only the input model and no horizontal input (initial-batch-based), and (2) *alignment-based* test cases providing an input delta and input corr (delta-corr-based). More tests covering the other bx application scenarios identified in Table 1 can be added in the future. Each category comprises test cases for each transformation direction.[8]

While the batch test cases are basic tests used to check if a given source model is transformed into a new target model correctly, different input source deltas are handled by the alignment-based tests. In particular, renaming, deleting, and moving persons and family members are currently addressed in these test cases. To keep the number of test cases manageable, only the batch category contains separate test cases for each combination of configuration parameters. In the alignment-based category, the parameters (preferences represented by the update policy) are changed dynamically during test case execution.

5.2 Implementing a benchmark with a bx tool

In order to use a specific bx tool with the Benchmarx framework, a single interface `BXTool`, depicted in Fig. 19, needs to be implemented. For EMF-based tools, we provide an abstract class `BXToolForEMF` that contains implementations for both `assert` methods and can be subclassed to simplify the integration in the benchmark.

The method `initiateSynchronisationDialogue` is invoked before each test case run and is used to establish the agreed upon common starting state for a given benchmark. For the Families-to-Persons case, this consists of a single empty family register and its corresponding single and empty person register, plus all internal, tool-specific internal data structures.

The two methods `performAndPropagateSourceEdit` and `performAndPropagateTargetEdit` are called from the test cases when corresponding edits should be performed and propagated on the corresponding models. In contrast, the methods `performIdleSourceEdit` and `performIdleTargetEdit` are used to modify source and target models, respectively, without propagating the change. These methods should be used whenever a change in the respective models does not affect the opposite model, e.g., when the birthday date of a person is changed, or the role of a family member in its containing family.

Multiple benchmarks including Families-to-Persons, together with solutions using numerous bx tools are maintained in the Benchmarx GitHub repository[9] together with documentation on how to add solutions to existing benchmarks as well as add entirely new benchmarks to the collection.

[8] Recall from Sect. 1 that the *forward* direction is from the families model to the persons model, while *backward* is from the persons model to the families model.

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

Table 2 Summary of classification of all tools and solutions

	BiGUL	BXtend	eMoflon	EVL+Strace	JTL	NMF	SDMLib
Style (tool)	restoration-based	restoration-based	propagation-based	restoration-based	restoration-based	propagation-based	restoration-based
Scenario (solution)	initial-state-based (with *hAln*)	initial-diag-based	s-delta-corr-based	initial-diag-based	initial-diag-based	o-delta-corr-based	initial-diag-based
Guarantees (tool)	round-trip laws	none	correctness, completeness	none	correctness, hippocraticness	correctness, hippocraticness	none
Consistency (tool)	implicit	implicit	explicit (grammar-based)	explicit (constraint-based)	explicit (constraint-based)	explicit (constraint-based)	implicit
Control (tool)	explicit and implicit	explicit	implicit	explicit	implicit	explicit and implicit	explicit
Direction (tool)	directed	directed	directed	concurrent	directed	directed	directed
Time (solution)	on-demand	on-demand	on-demand	on-demand	on-demand	live	on-demand
Automation (solution)	automatic	automatic	automatic	interactive	interactive	automatic	automatic

6 Description of benchmark solutions

This section presents solutions to the Families-to-Persons benchmark submitted to the Transformation Tool Contest 2017 [2]. The solutions have been selected to cover a wide spectrum of bx approaches, implemented in different tools and languages. Some of them (BiGUL, BXtend, and eMoflon) were provided as reference solutions to the TTC 2017 participants. The solutions in EVL+Strace, NMF Synchronizations, and SDMLib were submitted to TTC 2017 and were published in the TTC proceedings [28,47,56]. The JTL solution was prepared after the TTC.

A summary of a classification of these seven solutions based on our feature model is provided in Table 2. In the following subsections, one for each tool/solution in alphabetical order, we shall refer to this table and explain each tool's classification in detail.

6.1 BiGUL

BiGUL [35], short for *the Bidirectional Generic Update Language*, is the current result of a long line of research on *bidirectional programming* [22] predominantly performed by the programming language community. Bidirectional programming languages typically share two main ideas in common: (i) the task of programming a bx can be reduced by automatically deriving one direction of synchronization from the other direction that is explicitly programmed, and (ii) well-behavedness properties are guaranteed by providing a small set of well-behaved primitive functions, and combinators to allow bx programmers to compose complex, well-behaved bidirectional programs from these primitives.

Most bidirectional programming languages address *asymmetric* consistency relations, where one of the models (the *view*) is fully determined by the other model (the *source*). Instead of forward and backward synchronization, the terms *put* (the source is dependent, view is master) and *get* (the view is dependent, source is master) are used instead. In this asymmetric setting, *get* simplifies to a function that takes a source and produces a view, i.e., the old view is not necessary.

As a bidirectional programming language, BiGUL is unique in the sense that it provides a programming language (primitives and combinators) for programming *put* instead of *get*. Such a *putback-based* bidirectional programming language has the advantage that *get* can be fully derived from *put* (the inverse is not true in general), for the price that *put* is often more complex (and thus requires more effort to program) than *get*.

6.1.1 Classification

A summary of the features of BiGUL according to our common feature model for bx tools is provided in the first column of Table 2. These features will now be discussed in detail in the following.

BiGUL's architecture clearly follows a *restoration-based* style, i.e., the bx programmer has the job of programming *put* as *fCR* and thinks in terms of how to restore the consistency of both models by comparing them and making suitable changes to the dependent model. The exact delta between the previous and current master model is thus not of primary interest.

The main application scenario addressed by BiGUL is *initial-state-based*. Referring to Fig. 7, the exact tool architecture of BiGUL is the top-left restoration-based combination of *fCR* and *hAln*. Specific to BiGUL, *put* programs tend to be a recursive, flexible mix of intertwined "bits and pieces" of *fCR* and *hAln*, rather than clearly separated functions as Fig. 7 appears to suggest. A further point is that BiGUL is flexible enough to be used for other application scenarios, e.g., by encoding corrs, diags, and deltas as part of the models passed to the tool. When programming *hAln*, for example, one could then access this extra information and also update it if necessary. While this is indeed possible, it is also clear that the language was specifically designed for initial-state-based scenarios, which is how it was also applied to solve the benchmark.

BiGUL is formally founded and was originally developed in the dependently typed programming language Agda [42] so as to formally verify its well-behavedness guarantees; for practical usage a Haskell port of BiGUL is provided.[10] BiGUL guarantees basic *round-trip laws* for the programmed *put* and automatically derived *get* functions. These laws (*putget* and *getput*) are closely related to correctness and hippocraticness; we refer to Ko et al. for further details [35].

The underlying consistency relation is never specified explicitly when working with BiGUL. It is *implicitly* implied by the provided *put* program together with the guaranteed round-trip laws, which fix the corresponding derived *get* program.

BiGUL represents an interesting mix of *explicit* and *implicit* control over consistency restoration: *put* is explicitly programmed, while *get* is automatically derived, i.e., implicitly programmed. Well-behavedness guarantees that the derived *get*, if it exists, is unique for the provided *put*.

Finally, BiGUL currently only supports *directed* synchronization, performed *on-demand* by executing *put* or *get* as required. While support for (user) interaction can be implemented as required, there is no direct support for this; BiGUL is designed more for *automatic* consistency maintenance.

6.1.2 Benchmark solution with BiGUL

In this section we provide a top-down, high-level, and intentionally incomplete description of the solution to the Families-to-Persons benchmark with BiGUL. Our aim is not to explain all details, but rather to impart an intuition for the basic structure of the solution. We refer the reader interested in further details to the BiGUL solution available on GitHub.[11]

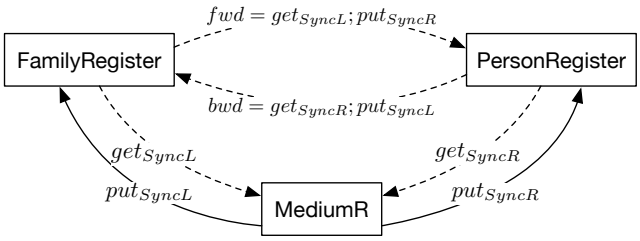


Fig. 20 Handling a symmetric bx with BiGUL

As BiGUL only directly supports asymmetric bx, the first challenge when implementing the Families-to-Persons benchmark is to decompose the symmetric bx into two asymmetric bx. The decomposition applied in the proposed BiGUL solution is depicted in Fig. 20. Bold arrows represent functions to be programmed by the bx developer, while dashed arrows represent automatically derived functions. In this decomposition, an additional data structure `MediumR` is introduced and can be thought of intuitively as the intersection of both data structures, i.e., it contains exactly the concepts that are present in both the source and target model spaces and that must be kept consistent. All the bx developer now has to do is program how `MediumR` models can be *put* into `PersonRegister` models via put_{SyncR}, and into `FamilyRegister` models via put_{SyncL}. The required *fwd* and *bwd* transformations can then be computed as depicted in Fig. 20 by composing programmed *put* and derived *get* arrows as required.[12]

To provide a feeling for actual BiGUL code, Listing 1 depicts the BiGUL program for put_{SyncR}. Recall that the architecture for this solution is restoration-based (with *hAln*) for an initial-state-based scenario. So this code represents *fCR* and *hAln* (see Fig. 7).

As both models are essentially lists (of persons and $(name, gender)$ pairs), *hAln* can be implemented with the auxiliary, higher-order function `align` (Line 4) that takes a matching condition and uses it to lift a BiGUL program on elements to lists. The strategy is to traverse the list of view elements and to search for the first source element that matches according to the condition (here simply a comparison of name and gender on Lines 7–8). If a match is found, then the first part of the *fCR* code (Lines 10–25) is executed (it does nothing as the elements are already consistent); if no match can be found, then a new source element is created using the second part of the *fCR* code (Lines 27–30). If un-

[10] `http://hackage.haskell.org/package/BiGUL`

[11] `http://bit.ly/bigul-f2p-benchmarx`

[12] The actual solution is a bit more complex as *SyncL* is further decomposed into two arrows.

Listing 1 Structure of a BiGUL program

```
1  syncR :: BiGUL PersonRegister MediumR
2  syncR =
3    $(rearrS [| \(PersonRegister ps) -> ps |])$
4      align (const True)
5        (\p (name, gender) ->
6          splitFullName
7            (getFullName p) == name &&
8            isMale p == gender)
9        (Case [
10         $(normalSV
11           [p| Male _ _ |]
12           [p| _ |]
13           [p| Male _ _ |])
14             ==>$(update
15                  [p| Male name _ |]
16                  [p| (name, True) |]
17                  [d| name = Skip splitFullName |]),
18         $(normalSV
19           [p| Female _ _ |]
20           [p| _ |]
21           [p| Female _ _ |])
22             ==>$(update
23                  [p| Female name _ |]
24                  [p| (name, False) |]
25                  [d| name = Skip splitFullName |])
26         ])
27       (\((familyName, firstName), gender) ->
28          (if gender then Male else Female)
29          (familyName ++ ", " ++ firstName)
30          defaultDate)
31       (const Nothing)
```

matched source elements remain after all view elements have been matched, they are deleted by `align`.

Finally, although the code might appear cryptic without looking up all details, the used BiGUL primitives are highlighted as keywords in Listing 1. The point here is that the entire program is actually a composition of primitives, but is designed in a way that it appears to be written in an imperative programming language [34].

6.2 BXtend

BXtend [6], an acronym for *Bidirectional Xtend*, is a pragmatic approach to programming bx, developed to address problems encountered in the practical application of existing bx languages and tools [7].

BXtend provides an *internal DSL* based on Xtend, a multi-paradigm language supporting object-oriented, procedural and functional programming, as well as a seamless integration with the Eclipse IDE and the mainstream programming language Java.

To realize a bidirectional transformation, the transformation developer defines a *correspondence model*, resembling the correspondence graph of a *triple graph grammar* [48]. In contrast to triple graph grammars, however, transformation rules are defined in a procedural rather than declarative way and are attached to correspondences. Both transformation directions are programmed separately; the transformation developer is responsible for programming both directions in a mutually consistent way. Altogether, BXtend provides a light-weight framework in which developers can structure their bx solutions in a straightforward manner.

6.2.1 Classification

BXtend's architectural style is *restoration-based*, addressing *initial-diag-based* application scenarios (see Table 2). The restoration-based architecture for initial-diag-based application scenarios is depicted to the left of Fig. 8: the programmer takes a diag and has the task of manipulating the dependent model until both models are consistent again. The corr between the models should also be updated in the process and returned.

With BXtend, the program representing *fCR/bCR* is decomposed into multiple "rules" consisting of restoration logic separated into forward and backward direction. Each rule defines flexibly, e.g., based on a certain type, if it is applicable and can be used to check and fix a specific inconsistency or not. Similarly, each rule checks first by exploiting the supplied diag if existing structure in the dependent model can be reused, suitably changed, or must be deleted and created as required. To perform *fCR/bCR* on the top-most level, an orchestration component is implemented to decide the order in which individual rules are applied and combined to realize *fCR/bCR*. This global component can also perform a final clean up if required. The delta implicitly induced as output (see Fig. 8) is mixed in the sense that each rule can freely perform deletion and creation as required in no fixed order.

BXtend provides no formal guarantees: both synchronization directions are implemented separately and are free to contradict each other. This can be viewed positively: the bx programmer is free to do whatever is required to solve the current task. There is no explicitly specified notion of consistency, i.e., this is *implicitly* given by the implemented pair of *fCR/bCR*. The transformation developer has full *explicit* control over consistency restoration, i.e., implements both *fCR* and *bCR* explicitly in Xtend. BXtend is currently designed to support *directed* synchronization, performed *on-demand*, in an *automatic* manner.

6.2.2 Benchmark solution with BXtend

To provide an impression of the benchmark solution with BXtend, Listing 2 depicts two classes: `Family2PersonTransformation` representing the orchestrating component, and `MotherDaughter2Female` representing a BXtend rule. For the Families-to-Persons benchmark, the same ordering of rules can be used in both directions

Listing 2 Rule orchestration and a specific rule with BXtend

```
1  class Family2PersonTransformation {
2    ...
3    def private void addRules() {
4      ...
5      rules.add(new Register2Register(
6        sourceModel, targetModel,
7        corrModel, decision))
8      rules.add(new MotherDaughter2Female(
9        sourceModel, targetModel,
10       corrModel, decision))
11     rules.add(new FatherSon2Male(
12       sourceModel, targetModel,
13       corrModel, decision))
14   }
15   ...
16 }
17
18 class MotherDaughter2Female
19   extends FamilyMember2Person {
20   ...
21   override sourceToTarget(String s) {
22     sourceModel.allContents
23       .filter(typeof(Family))
24       .forEach [ family |
25         val corr = family.eContainer
26                          .getCorrModelElem
27         val females = ECollections.newBasicEList
28         females.addAll(family.daughters)
29         if (family.mother != null)
30           females.add(family.mother)
31         females.forEach [ member |
32           val corrFemale =
33             member.getOrCreateCorrModelElement(s)
34           val female = getOrCreatePersonElement(
35                 family.name + ", " + member.name,
36                 corrFemale, ...)
37           (corr.personElement as PersonRegister)
38             .getPersons().add(female)
39         ]
40       ]
41   }
42
43   override targetToSource(String s) {
44     targetModel.allContents
45                .filter(typeof(Female))
46                .forEach [ p | ...]
47   }
48 }
```

(Lines 5–13). This order is specified in `addRules()`; three rules are created and added following the composition hierarchy of the metamodels (containers first). While the bx developer is free to program arbitrarily complex rule orchestration, current experience indicates that such a fixed order along the composition hierarchy is sufficient in most cases.

The specific rule `MotherDaughter2Female` checks for and handles, as its name suggests, inconsistencies between mothers/daughters in the source model and females in the target model. The two methods in this class `sourceToTarget` and `targetToSource` indicate that both synchronization directions (*fCR* and *bCR*) are explicitly specified. The first step in `sourceToTarget` is to filter for all families, and to accumulate all "female" family members, i.e., a mother if present, and all daughters. On Lines 32–33, the provided diag is accessed and used to retrieve the person connected to each female family member. With a helper method `getOrCreatePersonElement`, this person is either created if necessary, or updated as required.

6.3 eMoflon

eMoflon [38] applies triple graph grammars in the technological space of EMF. Models are viewed as graphs, where objects and links are represented by nodes and edges, respectively. A correspondence graph connecting the source and the target graph is maintained to keep track of the relationships between source and target elements. Consistency is specified declaratively by providing a *triple graph grammar* (TGG) [48]. A TGG consists of a *schema*, which defines types of correspondences to be maintained in the correspondence graph, and a set of *(triple) rules*. Each rule describes a synchronous extension of the whole triple graph. Rules are monotonic, i.e., they add nodes and edges, but they do not delete them. Together with a start graph (which is assumed to be empty in eMoflon), triple rules constitute a graph grammar which generates consistent triple graphs, i.e., triples of mutually consistent source, correspondence, and target graphs.

From synchronous triple rules, *directed rules* are automatically derived. A directed rule assumes that a local extension has been performed on the master graph, and propagates this extension to the correspondence graph and the dependent graph. To perform a directed synchronization, either forward or backward rules are applied, depending on the requested transformation direction. To restore consistency, all rule applications that were invalidated by changes made to the master graph (the graph that has been modified last) are revoked in a first "rollback" phase. In a second "translation" phase, directed rules are applied such that the resulting triple graph is consistent according to the TGG, i.e., the triple graph could also have been generated from the start graph by applying synchronous TGG rules.

6.3.1 Classification

eMoflon operates in a *propagation-based* style: As input, the tool thus expects old versions of the master, dependent, and correspondence graphs, as well as a structural delta between the old and the new versions of the master graph. The tool architecture is thus classified as *s-delta-corr-based* (see the right-hand side of Fig. 15). eMoflon derives both *fUP* and *bUP* automatically from the TGG, and uses these functions to propagate the structural delta from the master to the correspondence and dependent models. Each TGG rule can be viewed

as a connected pair of a source and target edit, describing an infinite set of paired source and target deltas produced by these edits. With this in mind, deriving *fUP* from a TGG is fairly straightforward: every input source delta is decomposed into a sequence of smaller source deltas that can be induced by a corresponding sequence of source edits from TGG rules. Once this sequence of required source edits has been computed, each source edit is propagated to the target edit specified by the corresponding TGG rule. The derived sequence of target edits is applied to the target model to produce a target delta as output of *fUP* (deriving *bUP* works analogously).

eMoflon guarantees *correctness*: After a directed synchronization, a triple graph is obtained that is guaranteed to be a member of the language generated by the TGG. Under certain conditions that can be mapped to required properties of the underlying TGG, eMoflon also guarantees *completeness*: Every structural delta that results in a master model for which a consistent triple exists, can be successfully propagated to the correspondence and dependent model. While such totality guarantees are often ignored in formal frameworks, practical instantiations of these frameworks tend to implement *partial* consistency restoration. For instance, although the *put* and *get* functions for BiGUL can fail in general, this is not regarded as a violation of the round-trip laws.

In eMoflon, the *consistency relation* is defined *explicitly* by a *grammar* that generates consistent triple graphs. *Synchronization* is controlled *implicitly*, as directed rules for consistency restoration are derived under the hood. eMoflon currently supports only *directed* synchronization. Finally, eMoflon is designed mainly for *on-demand* and *automatic* synchronization.

6.3.2 Benchmark solution with eMoflon

The benchmark solution is composed of a graph schema and a set of triple rules. The *graph schema* (not shown) defines two types of correspondence nodes for relationships between family and person registers, as well as between family members and persons, respectively. Each correspondence node is linked to exactly one node in the families graph and one node in the persons graph, respectively. The *rule set* consists of one rule for creating and relating family and person registers, and a set of rules for creating family members and persons. In principle, eight rules could be specified independently from each other for covering all cases with respect to the roles of family members, and the creation of a new or the reuse of an existing family. As this would result in highly redundant rules with many copies of rule elements, however, eMoflon supports *rule inheritance* that can be used to factor out common rule elements into abstract superrules.

Listing 3 Mapping family members to persons

```
1  #abstract #rule FamilyMember2Person
2            #with FamiliesToPersons
3
4  #source {
5    families : FamilyRegister {
6      ++ -families->f
7    }
8    ++ f : Family
9    ++ fm : FamilyMember
10 }
11
12 #target {
13   persons : PersonRegister {
14     ++ -persons->p
15   }
16   ++ p : Person
17 }
18
19 #correspondence {
20   families2persons : FamiliesToPersonsCorr {
21     #src->families
22     #trg->persons
23   }
24   ++ member2Persons : FamilyMemberToPerson {
25     #src->fm
26     #trg->p
27   }
28 }
29
30 #attributeConditions {
31   concat(", ", f.name, fm.name, p.name)
32 }
```

Listing 3 depicts an *abstract rule* which serves as root of the inheritance hierarchy. As TGGs are often regarded as a predominantly *visual* language, eMoflon also supports a read-only visualization for TGGs rules. To simplify comparison to all other bx languages evaluated in this paper, however, a *textual* concrete syntax is used here to represent and discuss TGG rules.

A rule which is designated as abstract is not applied as such, but (eventually) needs to be refined by *concrete* rules. The `with` clause on Line 2 refers to the underlying TGG schema. The triple rule is composed of three parts, indicated by the keywords `source`, `target`, and `correspondence`, respectively. The rule requires the presence of a family register (Line 5), a person register (Line 13), and a correspondence (Line 20) which is linked to the family register in the source graph and the person register in the target graph. Nodes and edges to be created are decorated with a `++` qualifier. In the source graph, a family is created along with an incoming edge from the family register; in addition, a family member is created. In the target graph, a person is created and linked to the person register. The family member and the person are connected by a correspondence to be added to the correspondence graph (Line 24). Finally, the attribute condition (Line 31) specifies that

the name of the person is composed from the family name and the first name of the family member.

Listing 4 Mapping members in existing families to persons

```
1  #abstract #rule ExistingFamily2Person
2            #extends FamilyMember2Person
3            #with FamiliesToPersons
4
5  #source {
6    families : FamilyRegister {
7      -families->f
8    }
9    f : Family
10 }
```

The abstract rule in Listing 3 creates a family together with a family member. This rule is refined by another abstract rule which assumes that a suitable family already exists (Listing 4). The subrule inherits all elements from its superrule, but redefines the binding state (removes the `++`) of the family node (Line 9) and its incoming edge (Line 7).

Listing 5 Mapping daughters to female persons

```
1  #rule DaughterToFemale
2  #extends FamilyMember2Person
3  #with FamiliesToPersons
4
5  #source {
6    ++ f : Family {
7      ++ -daughters->fm
8    }
9    ++ fm : FamilyMember
10 }
11
12 #target {
13   ++ p : Female
14 }
```

Listing 5 shows one out of four concrete rules for mapping family members in new families to persons. The rule adds the edge between the family and the member, which has been missing so far (Line 7), and redefines the type of the person from `Person` to `Female`.

Listing 6 Mapping daughters in existing families to females

```
1  #rule DaughterOfExistingFamilyToFemale
2  #extends ExistingFamily2Person, DaughterToFemale
3  #with FamiliesToPersons
```

Listing 6 demonstrates the use of *multiple inheritance*: The rule for creating a daughter in an existing family as well as a female person is obtained by extending both the abstract rule for mapping members in existing families to persons (Listing 4), and the concrete rule for mapping daughters in new families to females (Listing 5). Conflicts are resolved by favoring "context over create". This means that subrules are allowed to extend the precondition of their superrules but never reduce it. The family node `f` in `DaughterOfExistingFamilyToFemale` thus obtains its binding state from the rule `ExistingFamily2Person`.

As presented so far, the backward transformation would behave *non-deterministically*. To obtain a *configurable backward transformation*, controlled by configuration parameters, the set of applicable alternatives for a given match in the persons graph is constrained dynamically according to the current setting of configuration parameters. For example, if children are preferred in new families, mappings of a person to a parent as well as to a child in an existing family are disabled. As the TGG language does not offer any control structures, the configurability of the backward transformation is realized as an additional Java program that communicates with the transformation engine via an API.

6.4 EVL+Strace

The bx tool *EVL+Strace* [46] is based on EMF as well as the *Epsilon* framework [36], which provides tool support for a variety of DSLs for model transformation. In EVL+Strace, a transformation definition consists of a trace metamodel, EOL operations, and EVL constraints (Fig. 21).

The *trace metamodel* defines domain-specific types of links and link ends. Trace models contain copies of all relevant elements of source and target models, as well as links connecting these elements. By means of a rich trace model, it is possible to detect any category of changes to the participating models, including creation and deletion of objects and links, as well as modification of attribute values.

The behavior of the synchronizer is defined by *EOL operations*, i.e., operations for queries and updates written in the Epsilon Object Language, and *EVL constraints*, i.e., checks augmented with repair actions written in the Epsilon Validation Language. Each constraint is directed and checks an inconsistency between the trace model and one of the participating models which is caused by a change to this model. The corresponding

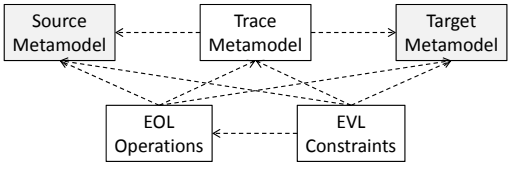


Fig. 21 EVL+Strace artifacts and their dependencies

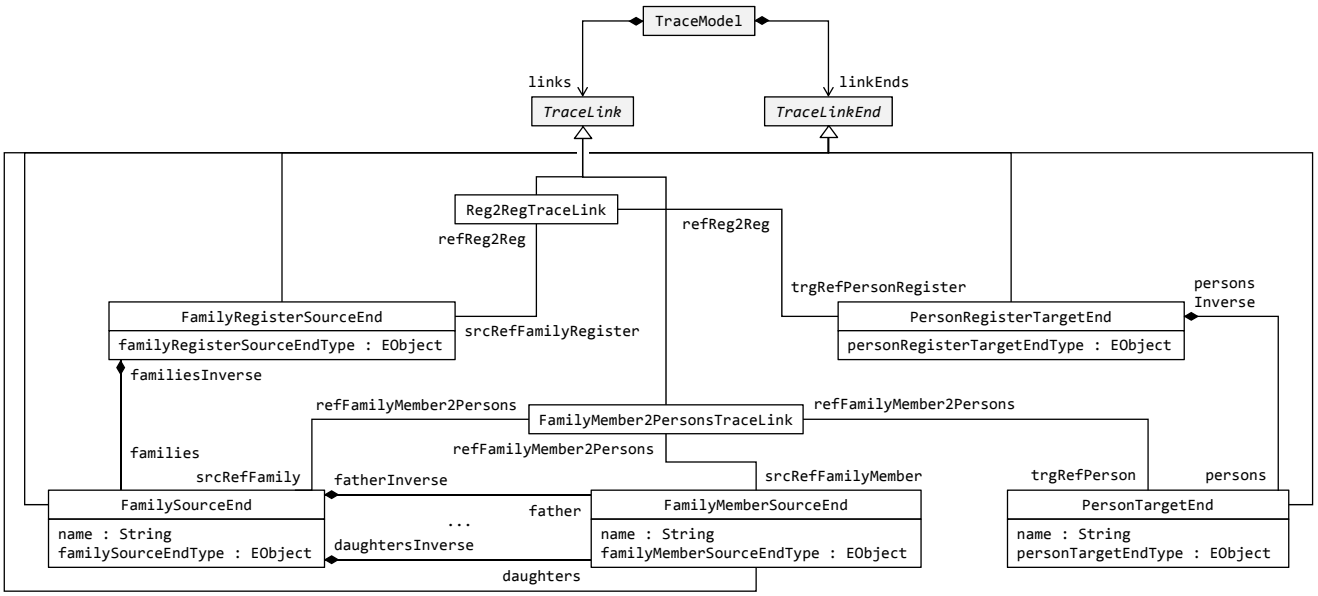


Fig. 22 Trace metamodel for the Families-to-Persons benchmark (without multiplicities)

repair action propagates this change to the trace model and the opposite model.

6.4.1 Classification

EVL+Strace is *restoration-based*: The transformation developer specifies how to detect and repair inconsistencies by explicitly programming *fCR* and *bCR*. With respect to horizontal and vertical inputs, EVL+Strace is classified as *initial-diag-based* and realizes the tool architecture displayed in Fig. 8 on the left-hand side.

As the transformation developer is free to implement both consistency checks and repairs, adherence to any formal property cannot be guaranteed in general. For example, an update that changes the trace model but preserves consistency can be responded to by an – according to hippocraticness unnecessary – repair action. Correctness is also not guaranteed: repair actions are specified in a procedural way and may fail to restore consistency or can contradict the checks for consistency.

In EVL+Strace, the *consistency relation* is defined *explicitly* as a set of *constraints*. The overall consistency relation is composed of two sets of directed constraints; the transformation developer is responsible for the mutual consistency of forward and backward constraints. Constraint restoration is also *explicitly* specified and *controlled* by programmed repair operations.

EVL+Strace is designed primarily for *concurrent synchronization*, i.e., consolidating changes applied to both participating models. Directed synchronization is included as a special case: If only one of the participating models has been modified, constraints may be violated only between this model and the trace model.

Synchronization is performed *on-demand* and *interactively*: Each constraint violation is reported to the user, who has to confirm (or reject) the respective repair action. Automatic synchronization is possible, but requires rewriting of EVL constraints (explained in the following section).

6.4.2 Benchmark solution with EVL+Strace

The following description is based on the solution in EVL+Strace submitted to TTC 2017 [47]. Figure 22 displays the *trace metamodel* for the Families-to-Persons benchmark. Specific types of trace links and trace link ends are defined by subclassing built-in abstract superclasses `TraceLink` and `TraceLinkEnd`, respectively. In the case of the Families-to-Persons benchmark, the trace metamodel defines two types of trace links for relating family and person registers as well as for relating family members along with their families with persons. Each *trace link* references a set of trace link ends, which may be considered proxies of source or target model objects. *Trace link ends* store links to source or target model objects with the help of attributes of type `EObject`. In addition, trace link ends may store attribute values and may be connected by links. In this way, the trace model may shadow the relevant parts of the source and target models.

Listing 7 gives an example of an *EVL* constraint along with its repair action. The constraint refers to family members in the families models (Line 1). As indicated by the name of the constraint (Line 2), it

Listing 7 Example of an EVL constraint

```
1  context Source!FamilyMember{
2    constraint isNewMale{
3      guard: self.isMale()
4      check: not self.isNew()
5      message:self+' is a new inserted element
6        in the source model'
7      fix{
8        title:'Insert its corresponding elements
9          in the trace and target models'
10       do {
11         var familyMemberSourceEnd
12           = addFamilyMemberSourceEnd(self);
13         var family;
14         if(self.isFather())
15           family = self.fatherInverse;
16         else
17           family = self.sonsInverse;
18         family.satisfies("isNew");
19         var familySourceEnd
20           = family.getTraceLinkEnd();
21         if(self.isFather())
22           familyMemberSourceEnd
23             .setFatherInverse(familySourceEnd);
24         else
25           familyMemberSourceEnd
26             .setSonsInverse(familySourceEnd);
27         var person = insertMale(family,self);
28         var personTargetEnd
29           = addPersonTargetEnd(person);
30         addFamilyMember2PersonsTraceLink(
31           familySourceEnd,
32           familyMemberSourceEnd,
33           personTargetEnd);
34         copySrc2Trg();
35       }
36     }
37   }
38 }
```

handles the creation of a new male member. Its guard (Line 3) ensures that it may be applied only to male family members. The check (Line 4) defines that the member must not be new, i.e., there must be a corresponding person in the persons model. If this check fails, a meaningful message (Lines 5–6) is created and displayed in the user interface. The fix (Lines 7–36) has a title (Lines 8–9) explaining the repair action to the user, as well as a `do` part (starting at Line 10) which defines all operations to restore consistency in a procedural way. Note that these operations refer both to the persons model and the trace model.

The Families-to-Persons benchmark deviates from the primary synchronization scenario targeted by EVL+Strace in two respects. First, the benchmark comprises only alternating direct rather than concurrent changes, i.e., only one model is changed, and the opposite model is updated to restore consistency. Second, the test suite is executed automatically, without any user interaction. As EVL+Strace is designed to support the more general case of concurrent synchronization, it can of course also handle the simplified case of directed synchronization. For automatic synchronization, the constraint definitions for interactive synchronization (as shown in Listing 7) have to be modified by moving the `do` part into the `check` part, as well as eliminating the `message` and the `fix` parts. These mechanical transformations were applied throughout the EVL code to execute the Families-to-Persons benchmark without user interaction.

6.5 JTL

JTL (Janus Transformation Language) [11,20] is a model transformation language specifically tailored to support bidirectionality and non-determinism. JTL is EMF compatible and provides tool support via an Eclipse plugin.

A transformation developer using JTL provides a set of constraints specifying a consistency relation. Together with the metamodels, these constraints are transformed to an Answer Set Programming (ASP) [24] problem, which an ASP solver can use to enable consistency restoration.

JTL reuses QVT-R syntax [43], but deviates from its semantics deliberately and significantly. In particular, JTL maintains a persistent trace, in contrast to the purely state-based design of QVT-R. Second, JTL takes non-determinism into account by generating all possible models which may be obtained by selecting one of multiple rules being applicable to a given match. From these candidates, the user has to select the desired result, which is then used in further synchronizations.

6.5.1 Classification

A summary of the features of JTL is provided in Table 2. JTL is *restoration-based*: the underlying constraint solver takes an inconsistent state of all models and figures out how to restore consistency. The exact delta applied is irrelevant for this strategy. With respect to horizontal and vertical inputs, therefore, JTL realizes an *initial-diag-based* architecture, handling correspondences as additional constraints. JTL guarantees correctness with respect to the specified set of constraints, and hippocraticness as the current state of all models can always be taken as a solution if it already fulfills all constraints (and is thus already consistent). JTL requires an *explicit* consistency relation defined as a set of constraints (Prolog-like "rules"). Consistency restoration is *implicit*: the underlying constraint solver automatically derives *fCR* and *bCR* based on the provided constraints. JTL currently supports *directed*, *on-demand*, and *interactive* synchronization (which reduces to automatic synchronization in the case of deterministic transformations).

Listing 8 The Families-to-Persons transformation in JTL

```
1  transformation Families2Persons(
2    family: Families, person: Persons) {
3    top relation FamilyRegister2PersonRegister {
4      enforce domain family
5        fr : Families::FamilyRegister { };
6      enforce domain person
7        pr : Persons::PersonRegister { };
8      where {
9        Father2Male(fr,pr);
10       Mother2Female(fr,pr);
11       Son2Male(fr,pr);
12       Daughter2Female(fr,pr); }
13   }
14   relation Father2Male {
15     n: String;
16     sn: String;
17     enforce domain family
18       fr : Families::FamilyRegister {
19         families = f : Families::Family {
20           name = sn,
21           father = m : Families::FamilyMember {
22             name = n
23           }
24         }
25       };
26     enforce domain person
27       pr : Persons::PersonRegister {
28         persons = m : Persons::Male {
29           name = n + sn
30         }
31       };
32   }
33   relation Mother2Female { ... };
34   relation Son2Male { ... }
35   relation Daughter2Female { ... }
36 }
```

6.5.2 Benchmark solution with JTL

The Families-to-Persons case implemented in JTL is depicted in Listing 8. JTL adopts a QVT-R [43] like textual concrete syntax. On Line 2, variables `family` and `person` are declared to match models conforming to the Families and Persons metamodels, respectively. When the top relation `FamilyRegister2PersonRegister` holds, then the two models are consistent. In the `where` block, further relations can be specified as required conditions for the top relation to hold. In this case, the top relation holds when the relations `Father2Male`, `Mother2Male`, `Son2Male`, and `Daughter2Male`, all hold. Variables bound in one relation, such as `fr` and `pr`, can be passed to invoked relations. In addition to structural constraints represented as object patterns in the relations, it is possible to specify attribute constraints, e.g., on Line 29.

Per default, JTL handles non-determinism by generating all possible models and allowing the user to iterate through them one by one. To implement the benchmark, which requires runtime configuration to make the tests deterministic, additional constraints have to be provided so the solver can determine the desired solution. Currently, these additional constraints have to be specified on a comparably low level of abstraction as direct input for the underlying constraint solver, and not with the normal JTL concrete syntax.

6.6 NMF Synchronizations

NMF Synchronizations [29] is a bx language realized as an internal domain-specific language with C# as a host language. To program a bx with NMF Synchronizations, a transformation developer specifies a set of consistency relations between the elements of the source and target models. Starting with a top-most relation, representing the consistency of the entire models, each relation can "invoke" other relations, which must hold for this relation to be satisfied. Leaf relations can be, for example, identity of primitive types such as strings.

This coupling of relations is specified with so called *synchronization blocks*, one of which is depicted schematically in Fig. 23. The relations are represented by horizontal arrows: Φ_{A-C} between source elements of type A and target elements of type C, and Φ_{B-D} between source elements of type B and target elements of type D. The *base relation* Φ_{A-C} holds only if the *target relation* Φ_{B-D} holds for certain pairs of (B, D) elements, determined by the *intra-model lenses* f and g (depicted as vertical arrows in Fig. 23).

A target relation may occur as base relation in another synchronization block, resulting in nested synchronization blocks. Furthermore, a relation may play the role of a base relation in multiple synchronization blocks, meaning that multiple consistency relations are required for making a pair of elements in the base relation consistent.

The intra-model lenses f and g are each composed of a *get* and a *put* function: $f.get : A \to B, g.get : C \to D$, and $f.put : A \times B \to A, g.put : C \times D \to C$.

As described above, the *get* function is a simple model query indicating how the base and target relations are coupled, i.e., for which elements the target relation must hold with respect to a given pair of elements in the base relation. In the simplest cases, this can be a getter navigating to the properties or contained children of an object.

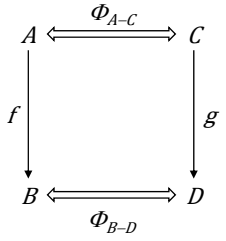


Fig. 23 Schematic view of a synchronization block

The *put* functions implement updates, i.e., a way of propagating changes of elements in the target relation back to elements in the base relation. As synchronization blocks pair such updates, they can be viewed as an implementation of update propagation (*fUP* and *bUP*).

NMF Synchronizations uses the incrementalization system *NMF Expressions* [31] to obtain notifications when the result of a *get* function changes. Therefore, it does not have to traverse the model to identify a change.

6.6.1 Classification

As synchronization blocks can be viewed as pairs of updates (and queries), the style of specification is probably closest to *propagation-based*. The addressed application scenario is *o-delta-corr-based*: The NMF solution accepts o-deltas and a corr as vertical and horizontal input, respectively. Hinkel et al. [29] provide a proof for *correctness* and *hippocraticness* under the assumption that the transformation developer ensures that *get-put* and *put-get* laws are actually satisfied for all intra-model lenses. The consistency relation between two models is specified *explicitly* by a set of coupled synchronization blocks, using only the *get* operations. This combination of relations coupled by queries is similar to a JTL program and can be best viewed as a *constraint-based* specification of consistency.

Consistency restoration is implemented by providing corresponding *put* operations for each synchronization block. An *explicit* specification of *put* satisfying the round-trip laws of intra-model lenses must be provided by the transformation developer in the general case. In simple situations, however, a *put* operation may be automatically derived from the *get* operation. The transformation developer can also reuse *put* implementations from a library of intra-model lenses. Furthermore, many details such as the order in which synchronization blocks are inspected and used to propagate parts of an update are fully automated by the framework and in this sense *implicit*.

While NMF Synchronization can be used to support many other cases [29], its sweet spot is for *directed*, *live* and *automatic* synchronization. This is also how the solution to the Families-to-Persons case is implemented.

6.6.2 Benchmark solution with NMF

The following description of the NMF Synchronizations solution to the Families-to-Persons benchmark is based on the corresponding TTC 2017 paper [28]. Two synchronization blocks are required for the transformation and are depicted in Fig. 24: (i) a block with the top-level relation *Reg2Reg* as base, and *Mem2Mem* as target, and (ii) a block with *Mem2Mem* as base and the identity relation on strings as target.

Listing 9 depicts fragments of the solution in C#. The two relations are implemented as *synchronization rules* subclassing the generic class `SynchronizationRule` (Lines 2–15). The actual synchronization blocks are defined by overriding the method `DeclareSynchronization` (Lines 4 and 12). This is basically just a textual representation of Fig. 24.

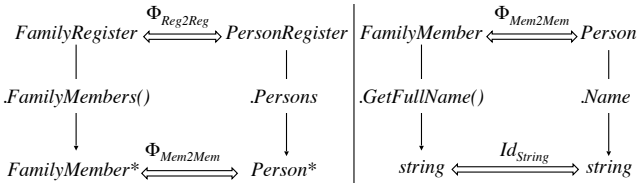


Fig. 24 Synchronization blocks for benchmark solution

Listing 9 Solution in NMF Synchronizations

```
1  ...
2  public class Reg2Reg:SynchronizationRule<
3   FamilyRegister, PersonRegister>{
4   public override void DeclareSynchronization(){
5    SynchronizeMany(SyncRule<Mem2Mem>(),
6     fam=>new FamilyMemberCollection(fam),
7     persons=>persons.Persons);
8   }
9  }
10 public class Mem2Mem:SynchronizationRule<
11  IFamilyMember, IPerson>{
12  public override void DeclareSynchronization(){
13   Synchronize(m=>m.GetFullName(), p=>p.Name);
14  }
15 }
16 ...
17
18 private class FamilyMemberCollection
19  :CustomCollection<IFamilyMember>{
20  ...
21  public override void Add(IFamilyMember item){
22   ...
23  }
24  ...
25 }
26
27 [LensPut(typeof(Helpers),"SetFullName")]
28 [ObservableProxy(typeof(Helpers),"GetFNameInc")]
29 public static string
30  GetFullName(this IFamilyMember  member){
31   return fullName.Evaluate(member);
32 }
33 public static INotifyValue<string>
34  GetFNameInc(this IFamilyMember member){
35   return fullName.Observe(member);
36 }
37 public static void SetFullName(
38  this IFamilyMember member,
39  string newName){
40  ...
41 }
42 private static ObservingFunc<
43  IFamilyMember,string> fullName =
44   new ObservingFunc<IFamilyMember,string>
45    (m=>m.Name == null ? null:
46     ((IFamily)m.Parent).Name + ", " + m.Name);
47 ...
```

Concerning the first synchronization block `Reg2Reg`, the target intra-model lens `.Persons` is simple enough (the *get* function merely collects the elements obtained via a multi-valued reference) so updates (the *put* function) can be inferred automatically. In contrast, a *custom collection* `FamilyMemberCollection` is required for the source intra-model lens, implemented on Lines 18–25. This helper class encapsulates the logic for determining the elements of the collection (traversing references to families and members sequentially), inserting an already created family member to the collection (Line 21) at the appropriate location in the families model, depending on the configuration parameters `PreferExistingToNewFamily` and `PreferParentToChild`.

Concerning the second synchronization block `Mem2Mem`, the situation is similar: the target intra-model lens `.Name` can be handled automatically, while the source intra-model lens `.GetFullName()` has to be implemented manually on Lines 27–46. On Line 27, an annotation is used to provide `SetFullName` as the *put* implementation for `GetFullName`. The custom method `SetFullName` sets the full name of a family member, potentially moving the member to a different family.

Finally, the annotation on Line 28 and the implementation of the helper method `fullName` on Lines 42–46 indicate that the NMF Synchronization framework provides hooks into infrastructure for an *incrementalization* of the intra-model lens. The reader is referred to Hinkel et al. [30] for further details.

6.7 SDMLib

SDMLib, short for *Story Driven Modeling Library*,[13] is a Java library to support Story Driven Modeling [41], a formalism based on graph transformations [17]. SDMLib provides an internal DSL with Java as a host language. Although SDMLib is not a bx tool, in the sense that it does not provide any extra support especially for developing bx, it is nonetheless included here as a "general purpose" model transformation tool against which bx tools should also be compared.

As models can be viewed as attributed, typed *graphs*, model transformations can also be regarded as a problem in the domain of graph transformations. A *graph rewrite rule* specifies the replacement of a graph pattern (left-hand side) by a subgraph to be embedded into the overall host graph. Graph rewrite rules can be used to specify not only in-place model transformations, but also model-to-model transformations by applying the rules to multiple graphs. Graph rewrite rules are *declarative* in the sense that the exact order in which to traverse the host graph and check for or replace patterns is not specified.

[13] http://www.sdmlib.org

6.7.1 Classification

The solution to the benchmark with SDMLib follows a *restoration-based* architectural style; *fCR* and *bCR* are implemented with graph rewrite rules.

The solution addresses the *initial-diag-based* application scenario (Fig. 8). A diag is taken as input, and the dependent model is manipulated until both models are consistent again.

No formal guarantees are provided by SDMLib as *fCR* and *bCR* are specified separately and independently. It is left up to the transformation developer to ensure that the implementations are not contradictory. As with the BXtend solution, this can be viewed as an advantage; the bx programmer may freely decide whatever is necessary to solve the current task.

The SDMLib solution has no explicit notion of *consistency* as it is *implicitly* given by the implemented pair of *fCR/bCR*. Accordingly, synchronization *control* is *explicit*, and programmed by providing suitable graph rewrite rules. The SDMLib solution was developed with the benchmark in mind and thus supports *directed*, *on-demand*, *automatic* synchronization.

6.7.2 Benchmark solution with SDMLib

The following description is based on the SDMLib solution for the Families-to-Persons benchmark at TTC 2017 [56]. As SDMLib is designed for in-place transformations on a single host graph, the solution uses a single metamodel as depicted in Fig. 25. To support incrementality efficiently, additional unidirectional references between `FamilyRegister` and `FamilyMember`, and `PersonRegister` and `Person` are used to detect changed elements. Finally, the correspondences are represented using two bidirectional references. The SDMLib solution thus operates on a single graph comprising the contents of both models as well as explicit correspondence links between respective elements.

The core of the SDMLib solution consists of two graph rewrite rules, one for each direction. While graph rewrite rules are typically presented using a visual concrete syntax, we use the actual textual concrete syntax provided by the tool to enable and simplify a comparison with all other solutions.

Listing 10 depicts the graph rewrite rule for the forward transformation using SDMLib's internal DSL, embedded into Java. The method for the graph rewrite rule uses code generated from the graph metamodel depicted in Fig. 25.

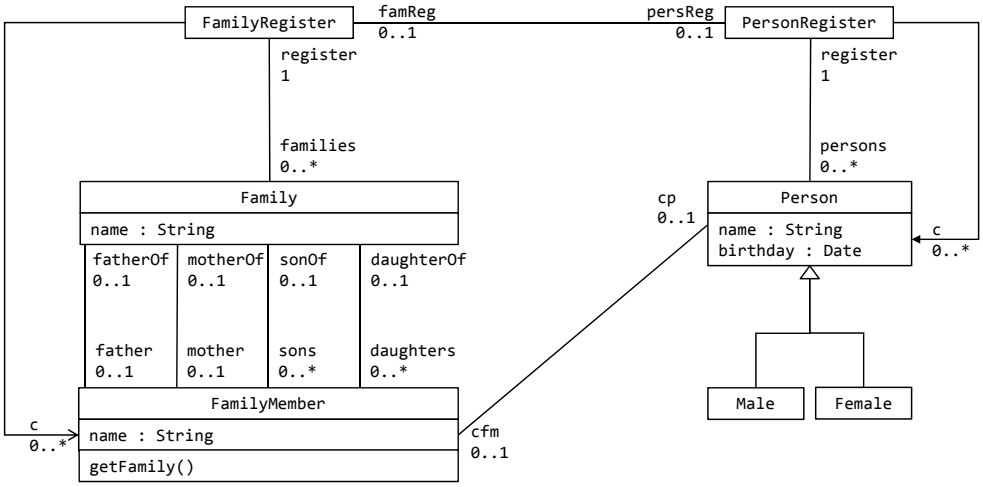


Fig. 25 SDMLib model used for the benchmark

The statements on Lines 2–9 define story pattern objects for the family register, the person register, and a family member, which need to be matched in the families model; the links connecting these objects are added implicitly to the story pattern by the invoked methods (e.g., `createCPO()`).

Listing 10 Forward transformation in SDMLib

```
1  private void transformForward() {
2   familyRegisterPO = new FamilyRegisterPO()
3     .withPatternObjectName("fr");
4   PersonRegisterPO personRegisterPO =
5    familyRegisterPO.createPersonRegisterPO()
6                    .withPatternObjectName("pr");
7   FamilyMemberPO memberPO = familyRegisterPO
8     .createCPO()
9     .withPatternObjectName("fm");
10
11  // there is an old corresponding person
12  memberPO.startSubPattern();
13  PersonPO oldPersonPO = memberPO.createCpPO()
14    .withName("oldP");
15  oldPersonPO.createCondition(p ->
16    ensureNameAndGender(p));
17  memberPO.endSubPattern();
18
19  // no corresponding person
20  memberPO.startNAC();
21  memberPO.createCpPO()
22    .withPatternObjectName("noOldP");
23  memberPO.endNAC();
24  MalePO personPO = memberPO
25    .createCpMalePO(Pattern.CREATE)
26    .withPatternObjectName("newP");
27  personPO.createRegisterLink(
28    personRegisterPO,
29    Pattern.CREATE);
30  personPO.createCondition(p ->
31    ensureNameAndGender(p));
32
33  familyRegisterPO.createLink(
34    memberPO,
35    Pattern.DESTROY);
36
37  familyRegisterPO.rebind(familyRegister);
38  familyRegisterPO.doAllMatches();
39 }
```

Lines 12-17 handle the case that a corresponding person object already exists. This case is realized with an optional subpattern, which is started at Line 12 and is terminated at Line 17. The pattern is composed of an old person object, connected to the family member object by a correspondence link. If pattern matching succeeds, the method `ensureNameAndGender` is invoked to restore consistency. Lines 20–31 deal with the case where there is no corresponding person object. This is handled with a *negative application condition*, defined on Lines 20–23. On Lines 24–31, elements of story patterns are created that define the actions to be performed when the negative application condition holds: A person object has to be created and linked to the person register; the method `ensureNameAndGender` is then called to establish consistency with the family object. Finally, the temporary link between the family register and the family member — which is set in the course of changes to the families model — is matched and then removed (Lines 33–35), the story pattern object for the family register is bound (Line 37), and the pattern is applied to all matches (Line 38).

7 Evaluation of benchmark solutions

After presenting seven solutions, we now evaluate them with respect to three measurable properties: *conciseness*, i.e., the size of the transformation definition measured in terms of lines of code (Sect. 7.1); *conformance* to requirements, measured in terms of passed and failed test cases (Sect. 7.2); and *performance*, measured in terms of runtime (Sect. 7.3). We leave qualitative properties such as level of abstraction or cognitive complexity to future work. We conclude the evaluation by discussing threats to validity in Sect. 7.4.

Table 3 Size of the transformation definitions of all solutions

	BiGUL	BXtend	eMoflon*	EVL+Strace	JTL†	NMF	SDMLib
Lines of code	176	211	192 + 25	1299	168 + 59	279	236
Number of words	1010	565	256 + 81	2878	260 + 513	607	427
Number of characters	6197	7571	3195 + 1382	50109	4338 + 4415	7215	6761

*The two numbers in each row indicate the amount of native eMoflon code + additional Java code.
†The two numbers in each row indicate the amount of native JTL code + additional code for the constraint solver.

7.1 Size of transformation definitions

As all the discussed solutions support a textual concrete syntax with which transformation definitions can be specified, a quantitative impression of the size of the transformation definitions can be obtained by counting the number of *lines of code* (excluding empty lines and comments), the *number of words* (character strings separated by whitespace) in these lines, and the *number of characters* in these words. The values obtained for these metrics are depicted in Table 3:

- All solutions apart from EVL+Strace require approximately the same number of lines of code. The EVL+Strace solution is composed of a trace metamodel, EVL constraints, and EOL operations. The solution is quite verbose, but can be generated partially from a declarative specification [46].
- Since the number of words approximates the number of lexical units, it is considered as a more accurate measure of the solution size than the number of lines. With respect to this measure, the declarative parts of the solutions in eMoflon and JTL are very concise; however, additional code is required to control the configurable backward transformation. SDMLib, BXtend, and NMF allow for compact solutions, as well, although both transformation directions have to be specified explicitly in SDMLib and BXtend. Notably, the size of the BiGUL solution exceeds the size of these solutions although one transformation direction (*get*) is derived from its opposite (*put*). This is probably due to the code required for horizontal alignment (*hAln*), as BiGUL is the only tool that does not require (and thus cannot exploit) provided corrs.
- Although the number of characters depends heavily on the length of identifiers and is thus a very questionable metric, it still yields almost the same ranking as for number of words. In the case of BXtend vs. NMF, however, it indicates that fundamental conclusions cannot be drawn from minor differences in number of words.

7.2 Conformance to requirements

As far as the benchmark is concerned, a solution is considered to *conform to requirements* if it passes all test cases. The current test suite for the Families-to-Persons case consists of 34 *test cases*, which are classified according to the taxonomy introduced in Sect. 5.1 (Fig. 17). In Table 4, the test cases are grouped into four categories, according to the direction (*forward* or *backward*) and granularity of synchronization (*batch* or *incremental*). For more detailed information (regarding not only the test suite, but the test results for all solutions), a read-only Google spreadsheet is available.[14]

Please note that with respect to the feature model of Fig. 17, the test suite achieves almost full *feature coverage*, i.e., each feature (but one) is covered by at least one test case. The feature round trip constitutes the only exception. As argued earlier (Sect. 3.2.3), round-trip laws are not explicitly tested because the required round-trip behavior is implied if directed synchronization is correct and hippocratic (which is tested by directed test cases).

With respect to results, we distinguish four rather than two types of results: (1) tests that pass and can be expected to pass as the features of the test match the features of the solution (*expected passes*), (2) tests that fail and can be expected to fail as the features of the test do not match the features of the solution (*expected fails*), (3) tests that pass even though the features of the test do not match the features of the solution (*unexpected passes*), and finally (4) tests that fail even though the features of the test match the features of the solution (*unexpected fails*).

Table 5 summarizes the test results, grouped into the same categories as the test cases shown in Table 4. The bottom part aggregates the numbers of the different categories. Before comparing the test results of different solutions, we analyze some global metrics derived from the table.

[14] `http://bit.ly/2RXABuw`

Table 4 Classification of test cases according to the taxonomy

Category	Name of test	Direction	Horizontal input	Vertical input	Change type	Runtime config
Batch FWD	*testInitialiseSynchronisation*	fwd	none	initial-based	-	no
	testFamilyNameChangeOfEmpty	fwd	none	initial-based	attribute	no
	testCreateFamily	fwd	none	initial-based	add	no
	testCreateFamilyMember	fwd	none	initial-based	add	no
	testDuplicateFamilyMemberNames	fwd	none	initial-based	add	no
	testNewDuplicateFamilyNames	fwd	none	initial-based	add	no
	testNewFamilyWithMultiMembers	fwd	none	initial-based	add	no
Batch BWD	*testCreateFamilyMembers-InExistingFamilyAsParent*	bwd	none	initial-based	add	yes
	testCreateMalePersonAsSon-InNewFamily	bwd	none	initial-based	add	yes
	testCreateMalePersonAsSon-InExistingFamily	bwd	none	initial-based	add	yes
	testCreateFamilyMembersIn-ExistingFamilyAsChildren	bwd	none	initial-based	add	yes
	testCreateDuplicateFamilyMembers-InExistingFamilyAsChildren	bwd	none	initial-based	add	yes
	testCreateMalePersonAsParent-InNewFamily	bwd	none	initial-based	add	yes
	testCreateMalePersonAsParent-InExistingFamily	bwd	none	initial-based	add	yes
	testCreateFamilyMembersInNew-FamilyAsParents	bwd	none	initial-based	add	yes
	testCreateDuplicateFamilyMembers-InNewFamilyAsParents	bwd	none	initial-based	add	yes
	testCreateFamilyMembersInNew-FamilyAsChildren	bwd	none	initial-based	add	yes
	testCreateDuplicateFamilyMembers-InNewFamilyAsChildren	bwd	none	initial-based	add	yes
Incr. FWD	*testIncrementalInserts*	fwd	state-based	state-based	add	no
	testIncrementalDeletions	fwd	corr-based	s-delta-based	del	no
	testIncrementalRename	fwd	corr-based	s-delta-based	attribute	no
	testIncrementalMove	fwd	state-based	s-delta-based	move (del+add)	no
	testIncrementalMixed	fwd	state-based	s-delta-based	add, del	no
	testIncrementalRoleChange	fwd	state-based	s-delta-based	add, del	no
	testStability	fwd	state-based	state-based	-	no
	testHippocraticness	fwd	state-based	state-based	attribute	no
Incr. BWD	*testIncrementalInsertsFixedConfig*	bwd	state-based	state-based	add	yes
	testIncrementalInsertsDynamic-Config	bwd	state-based	state-based	add	yes
	testIncrementalDeletions	bwd	state-based	state-based	del	yes
	testIncrementalRenamingDynamic	bwd	corr-based	s-delta-based	attribute	yes
	testIncrementalMixedDynamic	bwd	state-based	s-delta-based	add, del	yes
	testIncrementalOperational	bwd	state-based	o-delta-based	add	yes
	testStability	bwd	state-based	state-based	-	yes
	testHippocraticness	bwd	state-based	state-based	attribute	yes

Table 5 Aggregate test results, grouped into categories and classified as expected/unexpected passes/fails

Category	Result	BiGUL	BXtend	eMoflon	EVL+Strace	JTL	NMF	SDMLib
Batch FWD	expected pass	7	7	7	6	7	7	7
	expected fail	0	0	0	0	0	0	0
	unexpected pass	0	0	0	0	0	0	0
	unexpected fail	0	0	0	1	0	0	0
Batch BWD	expected pass	11	11	11	7	11	11	11
	expected fail	0	0	0	0	0	0	0
	unexpected pass	0	0	0	0	0	0	0
	unexpected fail	0	0	0	4	0	0	0
Incr. FWD	expected pass	3	8	5	8	2	6	8
	expected fail	5	0	0	0	0	0	0
	unexpected pass	0	0	0	0	0	0	0
	unexpected fail	0	0	3	0	6	2	0
Incr. BWD	expected pass	4	7	5	4	4	7	5
	expected fail	3	0	1	0	1	0	1
	unexpected pass	0	1	0	1	0	0	0
	unexpected fail	1	0	2	3	3	1	2
Total	expected pass	25	33	28	25	24	31	31
	expected fail	8	0	1	0	1	0	1
	unexpected pass	0	1	0	1	0	0	0
	unexpected fail	1	0	5	8	9	3	2

7.2.1 Global analysis

Let tt_{all} and pt_{all} denote the total number of all and all passed test cases, respectively. The *global success rate* sr_{all} is then defined as follows:

$$sr_{all} = \frac{pt_{all}}{tt_{all}} \tag{1}$$

From Table 5, we obtain $sr_{all} = \frac{199}{7\times34} \approx 0.84$. Thus, about 16% of all test cases fail, even though the solution authors tried hard to satisfy all test cases. This number demonstrates that the Families-to-Persons benchmark is indeed challenging, as claimed in Sect. 3.3.

A refinement of the preceding analysis reveals a significant difference between batch and incremental cases. For batch test cases, we obtain $sr_{bat} = \frac{121}{7\times18} \approx 0.96$; in contrast, $sr_{inc} = \frac{78}{7\times16} \approx 0.70$. Thus, incremental test cases are much more difficult to pass than batch test cases. Batch cases are designed to test basic functionality, which the solution authors should address first, before proceeding to deal with incremental behavior.

Similarly, we may compare forward and backward test cases. For forward cases, the success rate amounts to $sr_{fwd} = \frac{88}{7\times15} \approx 0.84$. For backward cases, we obtain essentially the same success rate: $sr_{bwd} = \frac{111}{7\times19} \approx 0.84$. For a more detailed analysis, we can compare forward and backward test cases for batch and incremental test cases separately and obtain: $sr_{fwd-batch} = \frac{48}{7\times7} \approx 0.98$ vs. $sr_{bwd-batch} = \frac{73}{7\times11} \approx 0.95$ and $sr_{fwd-incr} = \frac{40}{7\times8} \approx 0.71$ vs. $sr_{bwd-incr} = \frac{38}{7\times8} \approx 0.68$. While these values now indicate that the success rate in the backward direction is indeed slightly lower, this still does not provide strong support for our expectation that the configurable backward transformation should be *considerably* more difficult to implement than the forward transformation.

Finally, we examine some metrics concerning our distinction between expected and unexpected results. The *prediction quality* is defined by the relative number of expected results. As introduced already above, let tt_{all} denote the total number of test cases. Furthermore, let et_{all} denote the number of test cases with expected results. The *prediction quality* pq_{all} is defined as the quotient of these numbers:

$$pq_{all} = \frac{et_{all}}{tt_{all}} \tag{2}$$

From the table, we obtain the value $pq_{all} = \frac{208}{7\times34} \approx 0.87$. Thus, the prediction rate, based on the comparison of required and provided architectural features, is quite high. 28 out of 30 unexpected results are unexpected failures. Due to bugs in the solutions or technical limitations of the respective bx tools not covered by our feature model, it is not realistic to expect a prediction value close to 100%.

To conclude this subsection, we analyze the *unexpected pass ratio*, i.e., the relative number of test cases

yielding unexpected passes. Let up_{all} denote the total number of unexpected passes. We define the unexpected pass ratio upr_{all} as follows:

$$upr_{all} = \frac{up_{all}}{tt_{all}} \tag{3}$$

For this metric, we obtain the value $upr_{all} = \frac{2}{7 \times 34} \approx 0.01$, i.e., only 1% of the test cases are passed even though the respective tool does not possess the required features. This indicates a high quality of the test cases, which should provoke failures as often as possible.

Unexpected passes occur only in a test case which requires o-deltas (because the order in which persons are inserted affects the resulting families model). BXtend and EVL+Strace pass this test case unexpectedly. Although these two solutions do not exploit o-deltas, they are still able to pass the test by exploiting the fact that the EMF collections used are ordered. In the future, we plan to improve order-sensitive tests by, e.g., scrambling collections so this assumption fails.

7.2.2 Analysis of solutions

To compare the solutions, we employ the values of the success rate metric defined in Equation 1 (Table 6). As mentioned earlier, this metric does not take into account which test cases a solution may be expected to pass, based on the features of the respective tool on one hand and the features required by the test cases on the other hand. Therefore, we define an additional metric which is based only on those test cases which the solution is expected to pass. This metric is called *normalized success rate*:

$$nsr_{all} = \frac{ep_{all}}{tt_{all} - (ef_{all} + up_{all})} \tag{4}$$

For the normalized success rate, we count only the number ep_{all} of expected passes. From the number of all test cases tt_{all}, the number of all test cases is subtracted in which the solution exhibits either an expected fail (ef_{all}) or an unexpected pass (up_{all}). The normalized success rate is the quotient of these numbers.

Applied to our test results from Table 5, we obtain values for the success rates and the normalized success rates which are roughly the same for all but one solution: For BiGUL, the normalized success rate exceeds the success rate considerably. This is because the solution in BiGUL is the only one that does not maintain correspondences, resulting in a high number of expected failures. BiGUL has been designed for a different use case (view updates with round-trip properties) in which maintenance of correspondences is not feasible. The solution could be extended with correspondences which, however, must be maintained "manually".

For the remaining solutions, slight differences between the success rate and the normalized success rate stem from the fact that the corresponding tools are expected to fail if o-deltas are required. NMF is the only tool which supports o-deltas, and thus the only tool where both variants of success rates should always coincide. For the other tools, only slight differences are observed (if any) because there is only a single test case requiring o-deltas.

Tools such as eMoflon and JTL provide (different kinds of) well-behavedness guarantees. These tools trade well-behavedness guarantees for expressiveness: In the Families-to-Persons benchmark, the required behavior of the backward transformation differs from the behavior of the forward transformation considerably. eMoflon and JTL are essentially based on symmetric specifications of consistency relations and thus have difficulties in achieving conformance. For this reason, external mechanisms (Java and ASP programs) are exploited to improve synchronization behavior.

NMF pursues a compromise between well-behavedness guarantees and expressiveness: Certain parts of bidirectional transformations may be specified declaratively, other parts have to be programmed. While this hybrid approach improves expressiveness, well-behavedness guarantees hold only under certain assumptions (intra-model lenses), which must be proved manually.

Finally, in BXtend, SDMLib, and EVL+Strace, both transformation directions have to be specified explicitly. On the one hand, this implies that no well-behavedness guarantees may be provided at all. On the other hand, high conformance to the requirements may be achieved, because of the flexibility gained by separate specifications of forward and backward transformations. Nevertheless, EVL+Strace achieves only a rather low success rate (note, however, the remarks in Sect. 7.4.3).

7.3 Performance

To assess and compare the runtime scalability with respect to increasing model size for each solution, two experiments were conducted in both forward and backward directions (yielding four sets of measurements): (i) initial-batch-based transformations in forward and backward directions, and (ii) delta-corr-based transformations in forward and backward directions.

The batch transformations test how the solutions scale when creating the dependent model from scratch from a master model of increasing size (10^3 up to 10^6 model elements, i.e., nodes and edges). The incremental (delta-corr-based) transformations apply the exact

Table 6 Success rates of solutions

Metric	BiGUL	BXtend	eMoflon	EVL+Strace	JTL	NMF	SDMLib
sr_{all}	0.74	1.00	0.82	0.76	0.71	0.91	0.91
nsr_{all}	0.96	1.00	0.85	0.76	0.73	0.91	0.94

same edit (adding a single family member/person) to master models of increasing size, for which consistency can be theoretically maintained by applying *the same* corresponding edit (adding a single person/family member) to the dependent model in each case. This means that we test to what extent incremental consistency maintenance is decoupled from model size for each solution.

7.3.1 Experiment setup

All tests were performed on the same machine and in isolation for each solution. A standard machine with an Intel Core i7-4770 CPU was used, running at 3.40 GHz, with 16 GB of DDR3 RAM and with Microsoft Windows 10 64-bit as operating system. Java 1.8.0_191, Eclipse Neon (4.6.3), and EMF version 2.12.0 were used to compile and execute the Java code for the scalability test suite. As there were no substantial differences between the mean and median, each test was repeated just three times and the median of the measured time was computed. For all test runs, we used a threshold of 600s (10 minutes); if a solution took longer, we registered a time-out and terminated the test. The EVL+Strace solution could not be fully automated so we had to omit it and restrict our performance evaluation to the six remaining solutions.

To obtain models of increasing size, we generated synthetic models consisting of an increasing number of families with 2 parents and 3 children (and corresponding female and male persons).

7.3.2 Results and observations

Our four measurement results are depicted in Fig. 26, 27, 28, and 29. Each figure consists of two plots: a plot with a linear/linear scale to the left, and a plot with a log/log scale to the right. The linear plot is meant to provide a realistic impression for the actual complexity curve of each solution (i.e., linear, polynomial, exponential). The logarithmic plots help zoom into finer details for smaller models (practically invisible in the linear plot), and zoom out for larger models so even large differences in runtime can still be presented qualitatively.

To prevent distorting the plots, we omit solutions with poor scalability (JTL for batch and JTL+BiGUL for incremental transformations) from the linear plots (indicated for each case in the legend) as the logarithmic plots can better capture the complete picture. In all plots, the x-axis denotes the number of elements (nodes and edges) in the respective models in steps of 2k for small models (< 5k) and then steps of 20k up to 10^6.

Figure 26 depicts the obtained results for the forward batch transformation. In comparison to the other solutions, the JTL solution does not scale, already requiring over 300s for models with 5k elements (all other tools only require milliseconds). We were unable to obtain results for the JTL solution for models larger than 5k. This is due to the fact that JTL uses a generic constraint solver to derive the required consistency restoration behavior.

All other solutions scale reasonably well (linear with model size) up to about 500k, with the BXtend solution being the fastest. As from this point, the JVM-based solutions start breaking down: SDMLib already at 500k with a substantial and sudden increase in runtime, BXtend with a garbage collector exception at about 650k, and eMoflon with a sharp increase in runtime at about 850k. Only the BiGUL (Haskell) and NMF (C#) solutions scale linearly up to the maximum size.

Figure 27 depicts the obtained results for the forward incremental transformation. The NMF solution is the fastest here (below our measurement precision), and appears to be completely decoupled from model size. This is due to the fact that NMF relies on o-deltas, which are propagated from the master to the dependent model without any need for global analysis.

All other tools scale reasonably well with a moderate increase in runtime, until BXtend breaks down again at 650k for the same reason as for the batch transformation. The eMoflon solution is the only EMF-based solution that makes it until the maximum size, but starts showing problems as from 900k due to memory management problems.

The logarithmic plot contains all solutions and shows clearly that both BiGUL and JTL do not attain any speed up in the incremental case, taking just as long as for the batch transformation. Since BiGUL exploits neither correspondences nor deltas, a global re-alignment is required even after small incremental changes. JTL is based on a constraint solver which does not operate incrementally.

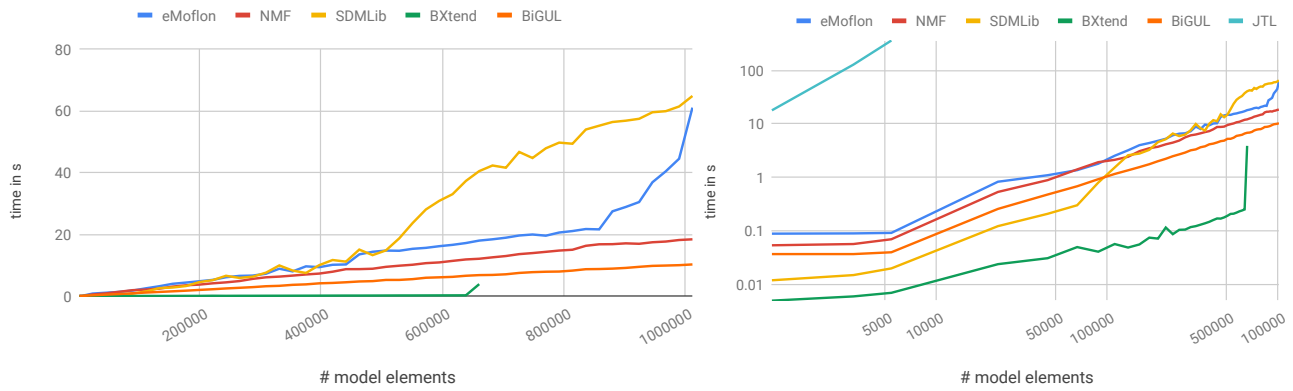


Fig. 26 Forward batch transformation: Linear/linear scale (left) and log/log scale (right)

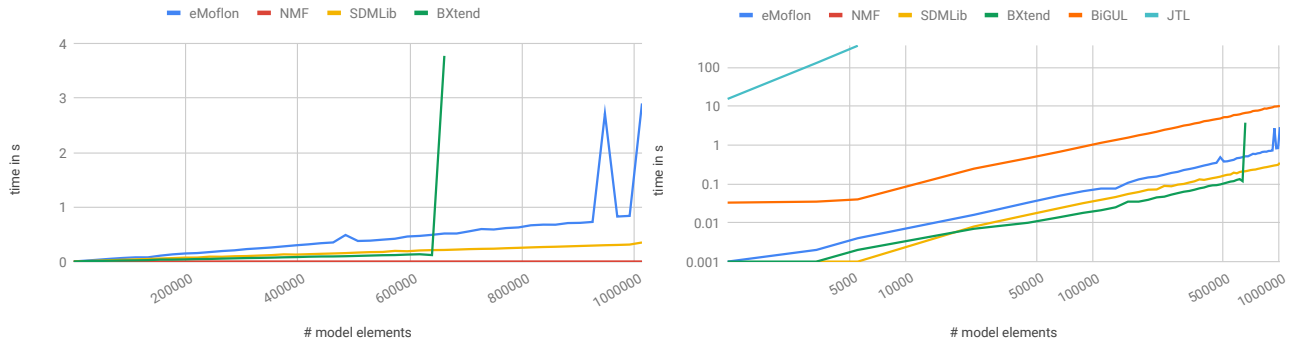


Fig. 27 Forward incremental transformation: Linear/linear scale (left) and log/log (right)

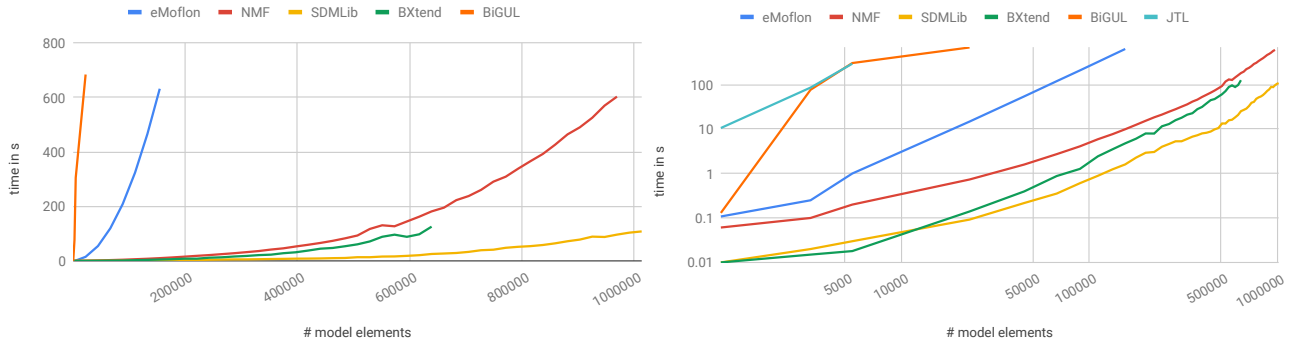


Fig. 28 Backward batch transformation: Linear/linear scale (left) and log/log scale (right)

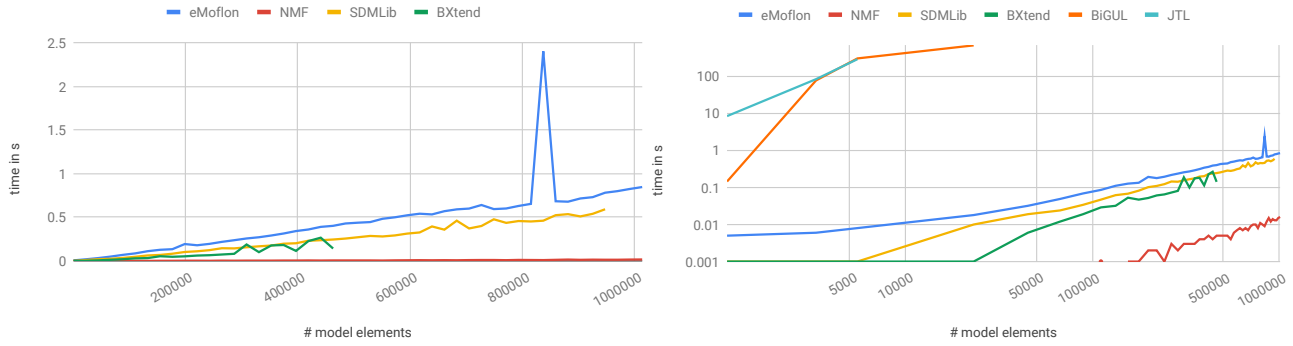


Fig. 29 Backward incremental transformation: Linear/linear scale (left) and log/log scale (right)

The results for the backward batch and incremental transformations are depicted in Fig. 28 and 29, respectively. Recall that the backward transformation requires a configurable update policy. This seems to pose a greater challenge (from a runtime scalability perspective) than the forward transformation for all solutions.

Only the BXtend, NMF, and SDMLib solutions scale with BXtend breaking down again at about 650k, and NMF timing out at about 960k. In this case, the SDMLib solution scales best. The solutions in JTL (up to 5k), BiGUL (up to 67k), and eMoflon (up to 155k) don't scale. While this might not be surprising for the solver-based JTL solution, the problem with the eMoflon and BiGUL solutions is probably related to how the configuration is handled. eMoflon, for example, always computes *all possible matches* and provides these to the update policy so it can make a choice supported by inspecting all alternatives. While this strategy and corresponding API can be advantageous for some use cases, it explodes for a large number of possibilities. A better strategy (at least for our limited scalability test) would be to compute alternatives *on demand* and consult the update policy after computing the next alternative.

The measurements for the backward incremental transformations reveal nothing new and are fairly analogous to the results in the forward direction.

7.4 Threats to validity

In the following, the main threats to validity of the study and corresponding mitigations are discussed. In particular, the threats suggested by Wohlin et al. [55] are considered and applied to the quantitative comparison of the different solutions.

7.4.1 Internal Validity

Internal validity refers to the level of influence that extraneous variables may have on the design of the study. As the Benchmarx framework is EMF-based, while not all the used bx tools are EMF-based, it is possible that this affects the evaluation. This threat has been mitigated both before and during the analysis of solutions by experimenting with all solutions and measuring the overhead posed by a required conversion to and from EMF data structures: The NMF solution requires a *very* costly conversion (EMF to C#) [28] so we were forced to omit this overhead for the measurements. The result, however, is now slightly biased in favor of the NMF solution as the measurements now exclude all overhead posed by the framework. The BiGUL solution requires a similar conversion (EMF to Haskell) but experiments show that this overhead was in contrast negligible and could be ignored. The SDMLib (JVM-based) solution uses a rewrite of edits to solve the compatibility problem. All other tools are EMF conform and work directly with the actual models. By and large, we succeeded in taking the heterogeneity of tool implementations into account. We have, however, interpreted performance results with care, focusing on fundamental differences concerning scalability.

Another threat concerns the quality of the solutions and the expertise of the different solution developers. This is an important point as bx tools and languages tend to be rather exotic and not very mature from a tooling perspective. To mitigate this threat and to make sure that we are comparing fundamental differences and not tool maturity, we ensured that all solutions were implemented by the tool developers themselves. This implies that the solutions are close to as best as they could be for each bx tool and approach.

7.4.2 Conclusion Validity

A threat to conclusion validity is constituted by the solution data extraction and analysis processes adopted to gather the results. Our strategy was discussed in advance with all co-authors, applied for a workshop paper [4], and refined for the TTC2017 [2].

In a quantitative study, the reliability of the measures refers to the reproducibility of the results. To ensure that all results are reproducible, we repeated each measurement three times and made sure that there were no substantial differences between the measured values. As the Benchmarx framework is supplied to solution developers via a GitHub repository, however, they had to install and execute the framework on their local machine to check conformance. The conformance results reported by the solution provider of EVL+Strace differed from the results obtained by the benchmark providers; this is mentioned in the respective sections, with a detailed reported on possible reasons in the additional material provided online (see Sect. 10). In all other cases, the results were the same, even on different platforms.

7.4.3 Construct Validity

Construct validity concerns the validity of our results with respect to the measured metrics.

Concerning the size of solutions: we are well aware of the fact that lines of code is an old but notoriously contentious metric (see e.g., Rosenberg [45] for a discussion of typical misconceptions). For measuring the size of the solutions, a simple tool was used to count the numbers of lines of code, words, and characters. These

numbers must be taken with a grain of salt. In particular, they refer to transformation definitions written in considerably different languages. Furthermore, they are sensitive to layout conventions and programming practices. For these reasons, the values from the size metrics were interpreted cautiously and primarily used to identify outliers (see Sect. 7.1). As there is too much noise in the data, it is, however, impossible to assert assumptions such as the following: "Solutions where both transformation directions are specified explicitly are two times larger than solutions in dedicated bx languages."

Concerning measuring conformance: developing a test suite for the Families-to-Persons benchmark proved challenging. As explained in Sect. 3.2.3, synchronization behavior cannot be derived uniquely from the consistency relation between the respective models. As a mitigation, all test cases were discussed and reviewed together with three co-authors (all bx researchers and different solution/tool developers) to decide what is to be "reasonable" synchronization behavior. All test cases were again discussed and reviewed as part of the TTC2017 (using GitHub as a discussion platform) with feedback, questions, and criticism from solution experts. We are confident that our test suite asserts reasonable behavior. As explained in Sect. 7.2, our test suite provides for feature coverage with respect to the feature model for classifying test cases (Fig. 17). In addition, the test suite provides for model coverage with respect to the types of model elements defined in the metamodels for families and persons. However, it should be noted that the test suite is used for benchmarking, not for detecting each and every error in the benchmark solutions. Consequently, the number of test cases, which have to be implemented manually by benchmark providers and understood by and discussed with solution providers, was confined to a bare minimum.

Concerning performance: runtime results have been interpreted with care, focusing on fundamental differences concerning scalability as the solutions differ substantially regarding their execution platforms (Haskell, C#, JVM-based, EMF-based).

7.4.4 External Validity

External validity refers to the generalizability of obtained results beyond the scope of the evaluation of the benchmark solutions.

The Benchmarx framework has been designed for comparing heterogeneous bx tools. The work presented in this paper demonstrates that this design goal has been achieved: Benchmarks may be executed by bx tools with considerably different architectures.

The framework is, however, biased towards a specific type of synchronization: directed synchronization by propagating changes from master to dependent models. Bx tools that are tailored towards this type of synchronization (e.g., NMF) are favored over bx tools which have been designed for different kinds of bx scenarios (e.g., BiGUL). We have been careful to take this fact into account when interpreting results and drawing possible conclusions. A generalization of the Benchmarx framework to other bx scenarios (e.g., concurrent synchronization) is left to future work.

The Families-to-Persons-benchmark might be considered as a toy example. However, it has been selected judiciously such that it can be implemented with acceptable effort. Simultaneously, this benchmark includes several challenges, which were summarized in Sect. 3.3. These challenges resulted in a significant failure rate: About 16% of all test cases failed, even though the solution authors made a serious attempt to provide best-effort solutions. Furthermore, we tried to choose an example that is amenable to a very wide range of tools. A consequence of this is that the data structures are simple lists (of lists). Benchmarks dealing with more complex structures such as graphs may yield different evaluation results.

Clearly, a spectrum of benchmarks would provide more data from which general conclusions may be drawn (even though the single case presented in this paper already provides some useful insights, which we expect to transfer to other transformation cases). Initial work in this direction has already been performed (see the end of Sect. 9.1).

So far, the performance evaluation is specific inasmuch as synthetic models are generated with a fixed strategy (many families with a fixed number of children) and the performance tests check incremental behavior only with respect to single edits. Further variation of the performance evaluation — with respect to both the generated models and the considered edits — is subject to future work. However, even the current evaluation shows considerable differences among the solutions implemented in heterogeneous bx tools (Sect. 7.3).

Finally, the compared solutions were provided by the tool developers themselves. While this positively affects internal validity, it also means that results concerning conformance and performance cannot be generalised to average developers; our results most probably represent an upper bound, with no guarantee that average developers can ever realise solutions of similar quality.

8 Assessment of the Benchmarx approach

In this section, we revisit the claims made in the introduction with respect to the usefulness of the Benchmarx approach. Each of the following subsections addresses one of these claims in turn. While the previous section focused on the evaluation of the solutions to the Families-to-Persons case, the current section targets the assessment of the Benchmarx approach itself.

8.1 Heterogeneity

The Benchmarx infrastructure has been designed for performing benchmarks in a wide variety of bx tools. This is achieved in particular by the concept of a synchronization dialog, introduced in Sect. 5. For implementing a specific benchmark case, a solution provider has to realize a pre-defined procedural interface with the bx tool at hand (Fig. 19). All the data maintained by the tool remains hidden behind this interface. The Benchmarx infrastructure thus abstracts from specific tool architectures, and case providers do not need to know in which ways bx tools store their data, nor how and when changes are propagated. A bx tool used for implementing a solution may have any tool architecture defined by the taxonomy displayed on the left-hand side of Fig. 16.

The solutions to the Families-to-Persons case demonstrate that the goal of supporting heterogeneity has been achieved. Table 2 shows that — apart from well-behavedness guarantees such as termination and completeness — all features from the bx tool feature model are covered by the bx tools used for implementing case solutions. This demonstrates that the bx tool interface may be implemented in considerably different ways. For example, while the state-based tool BiGUL merely updates model states on demand without maintaining any auxiliary data structures, NMF Synchronizations provides for live synchronization, relying on o-deltas and correspondences.

Heterogeneity is also supported with respect to technological spaces. While the Benchmarx infrastructure is EMF-based, bx tools are not required to be EMF-based. For example, BiGUL is based on the functional language Haskell, and NMF Synchronizations was implemented in C#, based on the .NET framework. In both tools, wrappers needed to be written for converting data or function calls. Providing wrappers proved to be a routine task which may be performed with acceptable effort.

8.2 Evaluation of solutions

In Sect. 7, we evaluated the solutions to the Families-to-Persons benchmark with respect to three categories: size of the transformation definition, conformance to requirements, and performance. While we did collect metrics for the purpose of evaluation, we also stressed repeatedly that we do not view a bx benchmark as a competition with the goal of identifying winners with respect to the collected metrics (as it is done in other domains).

We argue that the heterogeneity of bx tools should imply a qualitative evaluation approach, taking into account that the tools were built for different purposes with different technologies. In this way, we may provide for a more cautious and useful interpretation of benchmark results. In the sequel, we discuss this approach for the evaluation categories mentioned previously.

The size of the transformation definition was measured as an indicator of development effort. As has been discussed earlier, the validity of this metric is threatened for several reasons. Therefore, we used the size measurements merely to detect outliers. Altogether, size metrics played a minor role in our evaluation.

Conformance to requirements is usually measured only in terms of passed test cases. In contrast, we distinguish between expected and unexpected passes and failures. Whether a test result is classified as expected or unexpected, depends on the comparison of features required by the case and features provided by the bx tool. This comparison in turn relies on the conceptual framework introduced in Sect. 4. For example, a test case may require that correspondences between source and target models be maintained because they may not be reconstructed from the model states, or it may require that operations be propagated in a specific order (requiring o-deltas). The notion of normalized success rate takes these factors into account (Table 6).

Similarly, the feature model for the classification of bx tools (Fig. 16) assists in a balanced interpretation of performance results. For example, the performance tests for incremental transformations require to apply small changes to models of increasing size (Sect. 7.3). NMF Synchronizations, for instance, fits this use case perfectly because it performs live synchronization.

8.3 Understanding bx approaches

Finally, our work contributes to understanding the relationships between different bx approaches. To this end, Sect. 6 presented the solutions for the Families-to-Persons benchmark in a normalized format. All bx tools used for the solutions were classified with respect to our

taxonomy (Fig. 16). In addition, the solutions were described briefly, allowing to contrast them against each other. It is striking to see the wide spectrum of solution approaches: BiGUL applies putback-based programming in a functional language. In BXtend, forward and backward transformations are programmed separately in a procedural language. eMoflon defines consistency declaratively with the help of a graph grammar. In EVL+Strace, a correspondence model is defined along with declarative constraints and procedural repair actions. In JTL, consistency is defined by a set of declarative constraints which have to hold between source and target models. In NMF Synchronizations, consistency relations are defined declaratively as bijections; however, operations for updating models may have to be defined procedurally. Finally, in SDMLib forward and backward transformations are defined separately as graph transformations.

One of the lessons that might be learned from the Families-to-Persons benchmark concerns the trade-off between well-behavedness guarantees and expressiveness: Among the compared bx tools, correctness may be guaranteed only in eMoflon and JTL, both of which are based on a declarative, symmetric definition of the consistency relation between source and target models. However, in the Families-to-Persons benchmark the behavior of forward and backward transformations is not derived solely from the definition of the consistency relation, which would leave too many degrees of freedom. Rather, the behavior is specified explicitly — and separately for the forward and the backward direction. As a consequence, both eMoflon and JTL fail in certain test cases. In contrast, the BXtend solution is able to pass all test cases, but does not provide any guarantees.

9 Related work

In this section, we provide a discussion of related work divided into two broad groups: (i) existing results concerned with providing benchmark frameworks and examples in an MDE context, and (ii) previous work on classifying and comparing bx approaches. The restriction of (i) and (ii) to MDE and bx, respectively, is to keep the scope of our discussion manageable.

9.1 Benchmark frameworks and examples

To the best of our knowledge, Benchmarx is the first and only framework for developing and executing benchmarks for bidirectional transformations. The framework is based on initial conceptual work regarding the requirements bx benchmarks and bx benchmark frameworks should satisfy [3]. An implementation of these concepts was developed several years later and described in a paper for the BX 2017 workshop [4]. This article goes considerably beyond the preliminary work on Benchmarx as it presents the selected benchmark case and its challenges more accurately, includes a significantly extended section on the underlying conceptual framework regarding bx tool architectures, and presents and compares a broad spectrum of solutions to the selected case, demonstrating the applicability of the Benchmarx framework to heterogeneous bx tools.

The Transformation Tool Contest (TTC)[15] series has been organized to promote the comparison and evaluation of model transformation tools. Over the years the TTC has established conventions and guidelines for case descriptions and a systematic comparison of submitted solutions. While there have been bx case submissions to the TTC before – including our TTC case for the Families-to-Persons benchmark [2] – the TTC does not focus exclusively on bx and thus provides neither specialized infrastructure nor extra support for bx as the Benchmarx framework does. The TTC is also a *contest* typically with a ranking of solutions to identify winners and losers. Due to the heterogeneity of bx approaches and tools, our goal in this paper is more to understand fundamental differences and similarities without claiming which solution is "best".

The SHARE environment [25] (Sharing Hosted Autonomous Research Environments) provides general support for sharing research tools via virtual machines. SHARE has been used as a support environment in the TTC series to make benchmark solutions accessible without imposing any installation effort for inspecting and executing solutions. Solutions based on the Benchmarx framework may be distributed via SHARE or other virtual environments. Benchmarx and SHARE thus satisfy orthogonal needs.

There have been numerous proposals for benchmarking MDE technology. Varró et al. [52] suggest a suite of examples for graph transformation tools, chosen carefully to test various features related to graph pattern matching and rule-based model transformation. While there are bx approaches based on graph transformation, this benchmark cannot be directly applied to benchmarking bx solutions.

Bergmann et al. extend the graph transformation benchmark of Varró et al. by examples specifically focused on *incremental* graph pattern matching [5]. Although one of the examples in this extension is a bx problem, the benchmark itself covers features specific to incremental pattern matching and cannot be applied to benchmarking the broad spectrum of bx solutions.

[15] http://www.transformation-tool-contest.eu

With a special focus on evaluating and promoting the *scalability* of MDE tools, there have been several benchmark proposals from the BigMDE workshop series: Strüber et al. present a collection of examples and a conceptual framework for evaluating solutions [51]. Strüber et al. argue to evaluate not only the scalability of transformations (performance) but also the scalability of specifications (maintainability). While the examples and features are not specific to bx, we have included the "size" of specifications as a factor for our comparison of bx solutions in Sect. 7. Additional, better metrics for measuring the complexity of bx specifications can and should be explored in the future.

Also as part of the BigMDE workshop, Izs et al. present MONDO-SAM [32] as a framework to systematically assess MDE scalability. MONDO-SAM takes a very broad view on benchmarking MDE technology, covering numerous MDE tasks. While MONDO-SAM does not specifically cover bx benchmarking and its unique challenges, the Benchmarx framework could be aligned with MONDO-SAM in the future.

In recognition of the need to collect bx examples, the Bx Example Repository [10] was set up and is continuously being extended and maintained.[16] The repository includes a short description of the Families-to-Persons case, which we selected as an initial example for demonstrating the feasibility of the Benchmarx framework. This case, of which several variants exist, was originally proposed as part of the ATL [33] transformation zoo.[17] For its implementation with the Benchmarx framework, the case was refined into a bidirectional, incremental transformation with a configurable backward transformation, for which we developed a comprehensive set of test cases. The Benchmarx framework is meant to complement the Bx Example Repository. With time, the most promising examples in the repository can be chosen and suitably extended to establish them with the Benchmarx framework as bx benchmarks.

In order to enable a more comprehensive evaluation, bx tools should be compared with the help of a *spectrum* of benchmarks rather than with the help of just a single case (currently Families-to-Persons). In fact, more benchmarks are already available for evaluation with the Benchmarx framework, but have been implemented only in a few bx tools to date. Procuring further solutions is currently ongoing work. For example, all cases proposed by Westfechtel [53] have been implemented as bx benchmarks in the Benchmarx framework. While these cases were originally designed for evaluating the bx language QVT Relations (QVT-R [43]), they can be meaningfully applied to other bx languages and tools.

[16] `http://bx-community.wikidot.com/examples:home`

[17] `http://www.eclipse.org/atl/atlTransformations/#Families2Persons`

9.2 Classification and comparison of bx approaches

There has been a considerable amount of research done on classifying and comparing bx approaches. As an example of work towards a formal categorization of bx, our collection of bx tool architectures in Sect. 4 is inspired by the formal *tile* framework of Diskin; the interested reader is referred to his seminal work [14] for a more rigorous handling of synchronization operations and their composition.

Besides proposed formal frameworks for bx, there has also been work on comparing *similar* bx approaches: Foster et al. discuss and compare different and complementary approaches to bidirectional programming [22]. With similar goals, there have also been papers comparing numerous TGG tools [27,39]. In both cases, however, the comparison is restricted to relatively homogeneous variants of the same general bx approach. This allows for a detailed comparison but is orthogonal to our goal of comparing *diverse* bx approaches with the Benchmarx framework.

There have also been some attempts to compare rather different bx approaches, such as the comparison of JTL and some TGG tools provided by Eramo et al. [18]. This paper can be seen as a consequent development in the same direction of such ad-hoc comparisons, but with the crucial difference that we now provide both a conceptual and technical framework for establishing such bx frameworks in the future.

The Bx Community[18] has also been working towards establishing standard terminology and a classification schema for bx. Eramo et al. take first steps towards a taxonomy for bx [19] by providing a collection of basic definitions and requirements for bx approaches.

With similar goals, Hidaka et al. provide a comprehensive feature-based classification [26] of bx approaches. Compared to the feature model proposed by Hidaka et al., our feature model focuses on a relatively small set of "core features" and does not intend to provide a comprehensive classification of the bx landscape. Our feature model is not just a subset of Hidaka's feature model, but introduces different features and organizes them in a different way. Essentially, as far as the bx tool architecture is concerned (see the left-hand side of Fig. 16), our feature model precisely reflects the conceptual framework introduced in Sect. 4 and constitutes an original contribution; the remaining parts are cov-

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

ered by Hidaka's feature model but in much less detail due to the large number of features covered.

Other results towards classifying bx tasks and scenarios include work by Diskin et al. [15] on a taxonomy for bidirectional model synchronization, and Lämmel's megamodelling approach to characterizing different bx scenarios [37]. These results are complementary to our Benchmarx framework as their goal is to provide a *tool-independent* classification of bx problems, while we focus in this paper on classifying diverse bx *solution* strategies, tools, and approaches.

10 Conclusion and future work

We presented Benchmarx, an infrastructure for evaluating heterogeneous bx tools. Benchmarx is based on a conceptual framework for bx tool architectures, which are composed from basic operations for horizontal and vertical alignment, consistency restoration, and update propagation. The variability of bx tools is expressed with a feature model covering tool architectures, well-behavedness guarantees, the definition of consistency relations, and supported types of synchronization.

Benchmarks are executed with the help of test suites for evaluating both conformance to requirements and performance. Test cases are written as synchronization dialogs that start by establishing a consistent initial state and then proceed by propagating changes from the master model to the dependent model. The concept of a synchronization dialog does not make any assumptions regarding the bx tool architecture, allowing benchmarks to be implemented in heterogeneous bx tools.

For evaluating conformance, test cases are classified according to a taxonomy defined by a feature model. The taxonomy includes features for classifying tool architectures. In particular, the taxonomy covers application scenarios that define the inputs expected and exploited by bx tools. By comparing the required features of test cases against the provided features of bx tools, test results may be classified into expected and unexpected passes and failures. The heterogeneity of bx tools is thus taken into account when evaluating results.

The application of the Benchmarx infrastructure was demonstrated with the well-known Families-to-Persons benchmark, which is small yet challenging. The benchmark was implemented successfully in seven bx tools which differ considerably with respect to their technological spaces, tool architectures, and bx languages. A comprehensive evaluation with respect to the size of transformation definitions, conformance to requirements, and performance demonstrates that test results reveal significant differences among the respective solutions and underlying tools, and can be interpreted in a meaningful and balanced way - without any attempt to identify the "best" solution.

Altogether, the work presented in this paper underpins the claims made in the introduction: (1) The Benchmarx infrastructure is designed for heterogeneous tools. (2) Test results may be interpreted in a meaningful way, taking heterogeneity into account. (3) Comparison of solutions assists in understanding fundamental differences between different bx approaches.

This paper takes a decisive step towards achieving maturity of bx tools and enabling long-term industrial applications. In an industrial setting, the Benchmarx framework can be used not only to systematically compare and choose candidate bx tools, but also to simplify establishing a suitable architecture, and ensuring that the chosen bx tool can be replaced if it becomes obsolete, or if better alternatives become available. This reduces the dependency on a particular bx tool and thus lowers the risk of applying cutting-edge bx technology.

As future work, we plan to apply the Benchmarx infrastructure to other benchmark cases; initial work in this direction has already been performed by implementing the cases proposed by Westfechtel [53]. Only then will it be possible to draw more general conclusions concerning bx tools and languages. In addition, we will also consider generalizations of the Benchmarx infrastructure to other types of bx scenarios, e.g., concurrent synchronization.

Additional resources

Benchmarx on GitHub: The EMF-based implementation of the Benchmarx framework, all current benchmark examples, and all solutions to these benchmarks, is maintained as an open-source project on GitHub:

`https://github.com/eMoflon/benchmarx`

Conformance and Performance Data: All data used for our conformance and performance evaluation is available together with extra comments from the solution developers as a read-only Google sheet:

`http://bit.ly/2RXABuw`

A Glossary

Architecture (of a bx tool) External interface, defined by required inputs and provided outputs, and internal processing, defined by processing steps and their organization.

Automatic synchronization *Synchronization* that is not *interactive*.

Backward synchronization *Directed synchronization* in the direction of the *source model*.

Batch synchronization *Directed synchronization* that creates a new *dependent model*.

Benchmark A standardized test that serves as a basis for comparison or evaluation.
Bidirectional transformation (bx) A *transformation* that *synchronizes* a *source model* with a *target model*.
bx language A domain-specific language for defining *bidirectional transformations*.
bx law A condition that is satisfied by *bidirectional transformations*.
bx tool A tool for executing *bidirectional transformations*.
Change Any modification to the contents of a *model*.
Completeness The ability of a *bidirectional transformation* to process all *models* in the domain or range of the *consistency relation*.
Concurrent synchronization *Synchronization* in which both *source* and *target models* may be changed.
Conformance Satisfaction of requirements.
Consistency A condition on pairs of *source* and *target models* that ensures that both *models* agree on shared information.
Consistency relation A binary relation that includes all pairs of *source* and *target models* that are mutually *consistent*.
Correctness A *bx law* that demands *consistency* between the *source model* and the *target model*.
Delta Difference between two *versions* of the same *model*.
Dependent model The *model* that is created or changed in a directed synchronization.
Directed synchronization *Synchronization* from a *master model* to a *dependent model*.
Forward synchronization *Directed synchronization* in the direction of the *target model*.
Hippocraticness A *bx law* that excludes changes on *source* and *target models* that are already *consistent* before the execution of a *bidirectional transformation*.
Incremental synchronization *Synchronization* that modifies an already existing *model*.
Interactive synchronization *Synchronization* that is partially controlled by user interactions being performed during the synchronization.
Least change synchronization *Synchronization* that performs a minimal change with respect to a suitable metric to reestablish *consistency*.
Least surprise synchronization *Synchronization* which minimizes the surprise of the user of a *bx tool* and thus maximizes conformance to the user's expectations.
Live synchronization *Synchronization* that is performed immediately after each elementary *change*.
Master model The *model* that is read but not changed in a *directed synchronization*.
Metamodel Model that defines the structure of a set of *models*.
Model Abstraction of a *system* under study.
Operational delta A *delta* which is defined by a sequence of change operations from an old to a new *version* of a *model*.
Round-trip law A *bx law* that refers to a round trip of *directed synchronizations* being performed in sequence.
Source model A *model* that may act as first component of a pair in the *consistency relation* maintained by a *bidirectional transformation*.
Structural delta A *delta* which is defined in terms of structural elements being contained in both or only one of two *model versions*.
Symmetric synchronization *Synchronization* between two models, where neither *model* is a *view* of its opposite.
Synchronization Execution of a *bidirectional transformation* with the intent to establish or restore *consistency* between a *source model* and a *target model*.
Synchronization on-demand *Synchronization* that is performed only on explicit or implicit user request.
System Generic concept for designating a software application, software platform, or any other software artifact.
Target model A *model* that may act as second component of a pair in the *consistency relation* maintained by a *bidirectional transformation*.
Transformation A procedure that reads, creates, or changes a set of *models*.
Transformation definition The program that controls the transformation.
Termination An execution of a *transformation* which halts after a finite number of steps.
Version A state of an evolving *model* at a specific point in time, defined in terms of the model's contents at that time.
View An abstraction of a *model* that may be computed automatically from the model's content.
View-based synchronization *Synchronization* between a *model* and a *view* on this model.

References

1. Anthony Anjorin. An Introduction to Triple Graph Grammars as an Implementation of the Delta-Lens Framework. In Jeremy Gibbons and Perdita Stevens, editors, *Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures*, volume 9715 of *Lecture Notes in Computer Science*, pages 29–72. Springer, 2016.
2. Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. The Families to Persons case. In Garcia-Dominguez et al. [23], pages 27–34.
3. Anthony Anjorin, Alcino Cunha, Holger Giese, Frank Hermann, Arend Rensink, and Andy Schürr. Benchmarx. In Candan et al. [8], pages 82–86.
4. Anthony Anjorin, Zinovy Diskin, Frédéric Jouault, Hsiang-Shang Ko, Erhan Leblebici, and Bernhard Westfechtel. Benchmarx reloaded: A practical benchmark framework for bidirectional transformations. In Romina Eramo and Michael Johnson, editors, *Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, April 29, 2017.*, volume 1827 of *CEUR Workshop Proceedings*, pages 15–30. CEUR-WS.org, 2017.
5. Gábor Bergmann, Ákos Horváth, István Ráth, and Daniel Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2008.
6. Thomas Buchmann. BXtend — a framework for (bidirectional) model transformations. In Slimane Hamoudi, Luis Ferreira Pires, and Bran Selic, editors, *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD (MODELSWARD 2018)*, pages 336–345, Funchal, Madeira, January 2018. SciTePress.
7. 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.

8. K. Selçuk 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), Athens, Greece, March 28, 2014*, volume 1133 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.
9. James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Towards a principle of least surprise for bidirectional transformations. In Alcino Cunha and Ekkart Kindler, editors, *Proceedings of the 4th International Workshop on Bidirectional Transformations colocated with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015*, volume 1396 of *CEUR Workshop Proceedings*, pages 66–80. CEUR-WS.org, 2015.
10. James Cheney, James McKinna, Perdita Stevens, and Jeremy Gibbons. Towards a repository of bx examples. In Candan et al. [8], pages 87–91.
11. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: A bidirectional and change propagating transformation language. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *Lecture Notes of Computer Science*, pages 183–202, Eindhoven, The Netherlands, October 2010. Springer-Verlag.
12. 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 of Computer Science*, pages 260–283, Zurich, Switzerland, June 2009. Springer-Verlag.
13. Alberto Rodrigues da Silva. Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.
14. Zinovy Diskin. Model synchronization: Mappings, tiles, and categories. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, volume 6491 of *Lecture Notes in Computer Science*, pages 92–165. Springer, 2009.
15. Zinovy Diskin, Hamid Gholizadeh, Arif Wider, and Krzysztof Czarnecki. A three-dimensional taxonomy for bidirectional model synchronization. *Journal of Systems and Software*, 111:298–322, 2016.
16. Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *Journal of Object Technology*, 10:6: 1–25, 2011.
17. Karsten Ehrig, Esther Guerra, Juan De Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Gabriele Taentzer, Daniel Varró, and Szilvia Varro-Gyapay. Model transformation by graph transformation: A comparative study. In *Proceedings of the International Workshop on Model Transformations in Practice (MTiP 2005), Satellite Event of MoDELS 2005*, volume 3844 of *Lecture Notes of Computer Science*, pages 71–80, Montego Bay, Jamaica, 2005. Springer-Verlag.
18. Romina Eramo and Alessio Bucaioni. Understanding bidirectional transformations with TGGs and JTL. *ECEASST*, 57, 2013.
19. Romina Eramo, Romeo Marinelli, and Alfonso Pierantonio. Towards a taxonomy for bidirectional transformation. In Davide Di Ruscio and Vadim Zaytsev, editors, *Post-proceedings of the Seventh Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2014, L'Aquila, Italy, 9-11 July 2014*, volume 1354 of *CEUR Workshop Proceedings*, pages 122–131. CEUR-WS.org, 2014.
20. Romina Eramo, Alfonso Pierantonio, and Michele Tucci. Enhancing the JTL tool for bidirectional transformations. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, pages 36–41, 2018.
21. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17:1–17:65, May 2007.
22. Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three complementary approaches to bidirectional programming. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer, 2010.
23. Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors. *Proceedings of the 10th Transformation Tool Contest (TTC 2017)*, volume 2026 of *CEUR Workshop Proceedings*, Marburg, Germany, July 2017.
24. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.
25. Pieter Van Gorp and Steffen Mazanek. SHARE: a web portal for creating and sharing executable research papers. In Mitsuhisa Sato, Satoshi Matsuoka, Peter M. A. Sloot, G. Dick van Albada, and Jack J. Dongarra, editors, *Proceedings of the International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June, 2011*, volume 4 of *Procedia Computer Science*, pages 589–597. Elsevier, 2011.
26. Soichoro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-based classification of bidirectional transformation approaches. *Software and Systems Modeling*, 15(3):907–928, July 2016.
27. Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A survey of triple graph grammar tools. *ECEASST*, 57, 2013.
28. Georg Hinkel. An NMF solution to the Families to Persons case at the TTC 2017. In Garcia-Dominguez et al. [23], pages 35–39.
29. Georg Hinkel and Erik Burger. Change propagation and bidirectionality in internal transformation DSLs. *Soft-*

ware and Systems Modeling, 18(1):249–278, February 2019.
30. Georg Hinkel, Thomas Goldschmidt, Erik Burger, and Ralf H. Reussner. Using internal domain-specific languages to inherit tool support and modularity for model transformations. *Software and System Modeling*, 18(1):129–155, 2019.
31. Georg Hinkel, Robert Heinrich, and Ralf Reussner. An extensible approach to implicit incremental model analyses. *Software & Systems Modeling*, Jan 2019.
32. Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. MONDO-SAM: A framework to systematically assess MDE scalability. In Dimitris S. Kolovos, Davide Di Ruscio, Nicholas Drivalos Matragkas, Juan de Lara, István Ráth, and Massimo Tisi, editors, *Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering co-located with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, July 24, 2014.*, volume 1206 of *CEUR Workshop Proceedings*, pages 40–43. CEUR-WS.org, 2014.
33. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.
34. Hsiang-Shang Ko and Zhenjiang Hu. An axiomatic basis for bidirectional programming. *Proceedings of the ACM on Programming Languages*, 2(POPL):41:1–41:29, 2018.
35. Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. BiGUL: a formally verified core language for putback-based bidirectional programming. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 61–72. ACM, 2016.
36. Dimitris Kolovos, Louis Rose, Richard Paige, and Antonio Garcia-Dominguez. *The epsilon Book*, 2018. http://www.eclipse.org/epsilon.
37. Ralf Lämmel. Coupled software transformations revisited. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 239–252. ACM, 2016.
38. Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Developing eMoflon with eMoflon. In Davide Di Ruscio and Daniel 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.
39. Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A comparison of incremental triple graph grammar tools. *ECEASST*, 67, 2014.
40. Nuno Macedo and Alcino Cunha. Least-change bidirectional model transformation with QVT-R and ATL. *Software and Systems Modeling*, 15(3):783–810, July 2016.
41. Ulrich Norbisrath, Ruben Jubeh, and Albert Zündorf. *Story Driven Modeling*. CreateSpace Independent Publishing Platform, 2013.
42. Ulf Norell. Dependently typed programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.
43. Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.3*. Needham, MA, formal/2016-06-03 edition, February 2016.
44. Object Management Group. *OMG Meta Object Facility (MOF) Core Specification Version 2.5.1*. Needham, MA, formal/2016-11-01 edition, November 2016.
45. Jarrett Rosenberg. Some misconceptions about lines of code. In *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA*, page 137. IEEE Computer Society, 1997.
46. Leila Samimi-Dehkordi, Bahman Zamani, and Shekoufeh Kolahdouz-Rahimi. EVL+Strace: A novel bidirectional transformation approach. *Information and Software Technology*, 100:47–72, August 2018.
47. Leila Samimi-Dehkordi, Bahman Zamani, and Shekoufeh Kolahdouz Rahimi. Solving the Families to Persons case using EVL+Strace. In Garcia-Dominguez et al. [23], pages 54–62.
48. Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.
49. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition, 2009.
50. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Software and Systems Modeling*, 9(1):7–20, 2010.
51. Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. Scalability of model transformations: Position paper and benchmark set. In Dimitris S. Kolovos, Davide Di Ruscio, Nicholas Drivalos Matragkas, Jesús Sánchez Cuadrado, István Ráth, and Massimo Tisi, editors, *Proceedings of the 4th Workshop on Scalable Model Driven Engineering, Vienna, Austria, July 8, 2016*, volume 1652 of *CEUR Workshop Proceedings*, pages 21–30. CEUR-WS.org, 2016.
52. Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), 21-24 September 2005, Dallas, TX, USA*, pages 79–88. IEEE Computer Society, 2005.
53. Bernhard Westfechtel. Case-based exploration of bidirectional transformations in QVT Relations. *Software and Systems Modeling*, 17(3):989–1029, July 2018.
54. Bernhard Westfechtel. Incremental bidirectional transformations: Applying QVT Relations to the Families to Persons benchmark. In Ernesto Damiani, George Spanoudakis, and Leszek Maciaszek, editors, *Proceedings of the 13th International Conference on the Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 39–53, Funchal, Madeira, March 2018. SciTePress.
55. Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.
56. Albert Zündorf and Alexander Weidt. The SDMLib solution to the TTC 2017 Families 2 Persons case. In Garcia-Dominguez et al. [23], pages 41–45.

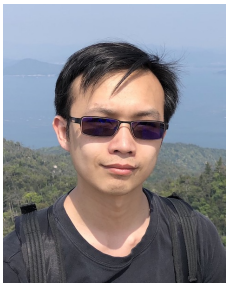
Anthony Anjorin received his B.Sc. and M.Sc. degrees in computational engineering from the Technische Universität Darmstadt (TUD), Germany in 2007 and 2010, respectively. He obtained his Ph.D. degree in computer science also from the TUD in 2014. He is currently a junior professor for model-based software development at Paderborn University. His research interests include triple graph grammars, graph transformations, bidirectional transformations, and model-driven engineering.

Thomas Buchmann received his diploma degree in mathematics from the University of Bayreuth in 2001. For the following three years he was employed as the manager of the software engineering department of a medium-sized local company. He obtained his doctoral as well as his habilitation degree (all in computer science) from the University of Bayreuth in 2010 and 2017, respectively. His research interests include model transformations, model-driven engineering, software product line engineering, domain-specific languages, and software architecture.

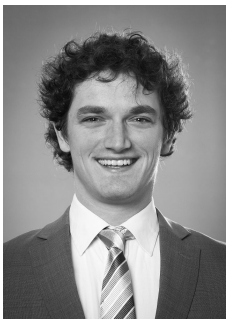
Bernhard Westfechtel received his diploma degree from University of Erlangen-Nuremberg in 1983 and his doctoral as well as his habilitation degree (all in computer science) from RWTH Aachen University in 1991 and 1999, respectively. Since 2004, he has been a full professor of computer science (in software engineering) at University of Bayreuth. His research interests include graph transformations, model-driven engineering, software product line engineering, software configuration management, software process modeling, software architecture, and re-engineering.

Zinovy Diskin is senior research fellow with McMaster University, Canada. His research interests include building mathematical models for software engineering including consistency management, assurance, and cyber-physical systems. He received Masters in mechanical engineering from Bryansk State Technical University, PhD in mathematics from Omsk State University and Dr. Math from the University of Latvia.

Hsiang-Shang 'Josh' Ko obtained his DPhil degree from the University of Oxford in 2014, and is now an Assistant Professor by Special Appointment at the National Institute of Informatics, Japan. His research interests include dependently typed programming, datatype-generic programming, bidirectional programming, Algebra of Programming/program calculation, and functional programming/type theory.



Romina Eramo obtained her Ph.D. degree in Computer Science in 2011 at the Department of Information Engineering, Computer Science and Mathematics (DISIM), University of L'Aquila. Currently, she is a Researcher in software engineering at University of L'Aquila. Her research interests include model-driven engineering, quality aspects in software engineering, domain-specific languages, software architecture, cyber-physical systems and embedded systems.



Georg Hinkel received his B. Sc. and M. Sc. degrees in computer science from the Karlsruhe Institute of Technology (KIT), in 2011 and 2014, respectively, and the B. Sc. degree in math in 2012. In 2017, he received his Ph.D. degree on implicit incremental model analyses and transformations from the KIT. Currently, he is a software technology engineer at Tecan Software Competence Center GmbH. His research interest covers model-driven engineering, incrementality and laboratory automation.

Leila Samimi-Dehkordi received her B.Sc. in Software Engineering from Iran University of Science and Technology (IUST) in 2008 and an M.Sc. in Algorithms and Computation from University of Tehran (UT) in 2011. She obtained her Ph.D. degree in Software engineering from University of Isfahan (UI) , Isfahan, Iran. Her Ph.D. research focussed on Model-Driven Development, Bidirectional Model Transformations, Traceability, and Change Propagations. She is a member of the Model Driven Software Engineering Research Group (MDSE) at University of Isfahan.

Albert Zündorf received his diploma degree from RWTH Aachen University in 1990 and his doctoral degree from RWTH Aachen University in 1996. He did his Habilitation at Paderborn University in 2002. He has been a full professor of software engineering at Kassel University since 2002. His research interests include graph transformations, model-driven engineering. He is known for his work on Progres, Fujaba, and SDMLib.