Design Space of Heterogeneous Synchronization

Michał Antkiewicz and Krzysztof Czarnecki

University of Waterloo
Generative Software Development Lab
http://gsd.uwaterloo.ca
{mantkiew, k2czarne}@uwaterloo.ca

Abstract. This tutorial explores the design space of *heterogeneous synchronization*, which is concerned with establishing consistency among artifacts that conform to different schemas or are expressed in different languages. Our main application scenario is synchronization of software artifacts, such as code, models, and configuration files. We classify heterogeneous synchronizers according to the cardinality of the relation that they enforce between artifacts, their directionality, their incrementality, and whether they support reconciliation of concurrent updates. We then provide a framework of artifact operators that describes different ways of building heterogeneous synchronizers, such as synchronizers based on artifact or update translation. The design decisions within the framework are described using feature models. We present 16 concrete instances of the framework, discuss tradeoffs among them, and identify sample implementations for some of them. We also explore additional design decisions such as representation of updates, establishing correspondence among model elements, and strategies for selecting a single synchronization result from a set of alternatives. Finally, we discuss related fields including data synchronization, inconsistency management in software engineering, model management, and model transformation.

1 Introduction

The sheer complexity of today's software-intensive systems can only be conquered by incremental and evolutionary development. As Brooks points out [1], "teams can grow much more complex entities in four months than they can *build*," where "build" refers to the traditional engineering approach of specifying structures accurately and completely before they are constructed. However, despite important advances in software methods and technology, such as agile development and object orientation, evolving software to conform to a changed set of requirements is notoriously hard. Evolution is hard because it requires keeping multiple software artifacts such as specifications, code, configuration files, and tests, consistent. A simple change in one artifact may require multiple changes in many artifacts and current development tools offer little help in identifying the artifacts and their parts that need to be changed and performing the changes.

Synchronization is the process of enforcing consistency among a set of artifacts and *synchronizers* are procedures that automate—fully or in part—the synchronization process. *Heterogeneous synchronizers* synchronize artifacts that conform to different schemas or are expressed in different languages. Many processes in software engineering can be viewed as heterogeneous synchronization. Examples include reverse engineering models from code using code queries, compiling programs to object code, generating program code from models, round-trip engineering between models and code, and maintaining consistency among models expressed in different modeling languages.

While many approaches to synchronization of heterogeneous software artifacts exist, it is not clear how they differ and how to choose among them. The purpose of this tutorial is to address this problem. We explore the design space of heterogeneous synchronizers. We cover both the simpler synchronization scenarios where some artifacts are never edited directly but are re-generated from other artifacts and the more complex scenarios where several artifacts that can be modified directly need to be synchronized. Both kinds of scenarios occur in software development. Example of the simpler scenario is generation of object code from source code. The need for synchronizing multiple heterogeneous artifacts that are edited directly arises in *multi-view development* [2, 3], where each stakeholder can understand and change the system through an appropriate view. The motivation for providing different views is that certain changes may be most conveniently expressed in a particular view, e.g., because of conciseness of expression or the familiarity of the a stakeholder with a particular view.

The tutorial is organized as follows. In Section 2, we present kinds of relations among software artifacts and concrete examples of such relations. In Section 3, we introduce kinds of synchronizers that can be used for reestablishing the consistency among artifacts. We classify heterogeneous synchronizers according to the cardinality of the relation that they enforce between artifacts, their directionality, their incrementality, and whether they support reconciliation of concurrent updates in Sections 4-6. The need for reconciliation arises in the context of concurrent development, where developers need to concurrently modify multiple related artifacts. We provide a framework of artifact operators that describes different ways of building heterogeneous synchronizers, such as synchronizers based on artifact or update translation. The operator-based approach is inspired by the manifesto for model merging by Brunet et al. [4]. The design decisions within the framework are described using feature models. We present 16 concrete instances of the framework, discuss their properties, and identify sample implementations for some of them. We summarize the synchronizers and discuss the tradeoffs among the synchronizers in Section 7. In Section 8, we explore additional design decisions such as representation of updates, establishing correspondence among model elements, and strategies for selecting a single synchronization result from a set of alternatives. Finally, we discuss related fields including data synchronization, inconsistency management in software engineering, model management, and model transformation in Section 9. We conclude in Section 10.

Purpose and Approach The purpose of the tutorial is to present a wide family of scenarios that require heterogeneous synchronization and the different solutions that can be applied in each scenario. The solutions are characterized by the scenarios they support, such as unidirectional or bi-directional synchronization, and the different design choices that can be made when constructing a synchronizer. The discussion of the scenarios and design choices is made more precise by considering the properties of the relations that are to be maintained among sets of artifacts and formulating the synchronizers using a set of artifact operators. The formalization does not consider the structure of the artifacts or their semantics. Whereas such a treatment would allow more precision in the analysis of choices, it would introduce a considerable amount of additional complexity and detail. We leave this endeavor for future work.

The intended audience is primarily those interested in building heterogeneous synchronizers. This audience can learn about the different design choices, the tradeoffs among the choices, and examples of systems implementing particular kinds of synchronizers. Furthermore, the operator-based formalization of the different kinds of synchronizers may also be of interest to researchers studying the semantics of model transformations.

2 Relations Among Software Artifacts

Modern software development involves a multitude of artifacts of different types, such as requirements and design models, program code, tests, XML configuration files, and documentation. Since the artifacts describe the same software system, they are related to each other in various ways. For example, a design model and its implementation code should be related by *refinement*. Furthermore, both the code and its XML configuration files have to use *consistent* names and identifiers. Also, the design model should *conform* to the *metamodel* defining the abstract syntax of the language in which the model is expressed.

In this tutorial, we usually consider software artifacts simply as typed values. An *artifact type* is a set of artifacts and it may be viewed as an extensional definition of a language. For example, assuming that $\mathcal{J}$ denotes the Java language, we write $P \in \mathcal{J}$ in order to denote that the artifact P is a Java program. Alternatively, we may also indicate the type of an artifact using a subscript, e.g., $P_{\mathcal{J}}$. On few occasions, we also consider the internal structure of an artifact, in which case we view an artifact as a collection of elements with attributes and links among the elements.

When an artifact is modified, related artifacts need to be updated in order to reestablish the relations. For example, when the design model is changed, the implementation code may need to be updated, and vice versa. The general problem of identifying relations among artifacts, detecting inconsistencies, handling of inconsistencies, and establishing relations among artifacts is referred to as *consistency management*. Furthermore, the update of related artifacts in order to re-establish consistency after changes to some of these artifacts is known as *syn-*

chronization, *change propagation*, or *co-evolution*. We refer to synchronization as *heterogeneous* if the artifacts being synchronized are of different types.

Definition 1. Consistent Artifacts. *We say that two artifacts $S_{\mathcal{S}}$ and $T_{\mathcal{T}}$ are* consistent *or* synchronized *with respect to the relation $R \subseteq \mathcal{S} \times \mathcal{T}$ iff $(S_{\mathcal{S}}, T_{\mathcal{T}}) \in R$.*

In general, two or more artifacts need not be consistent at all times [2, 5]. For example, the implementation code may be out of sync with its design model while several changes are being applied to the model. In this case, the inconsistency between the code and the design is desirable and should be tolerated. Only after the changes are completed, the code is updated and the consistency re-established. Consequently, some authors use the term *inconsistency management* [6–8].

The relations among software artifacts may have different properties. For a binary relation $R \subseteq \mathcal{S} \times \mathcal{T}$, we distinguish among the following three interesting cases:

1. R is a *bijection*. This is the *one-to-one* case where each artifact in $\mathcal{S}$ corresponds to exactly one artifact in $\mathcal{T}$ and vice versa.
2. R is a *total and surjective function*. This is the *many-to-one* case where each artifact in $\mathcal{S}$ corresponds to exactly one artifact in $\mathcal{T}$ and each artifact in $\mathcal{T}$ corresponds to at least one artifact in $\mathcal{S}$.
3. R is a *total relation*. This is the *many-to-many* case where each artifact in $\mathcal{S}$ corresponds to at least one artifact in $\mathcal{T}$ and each artifact in $\mathcal{T}$ corresponds to at least one artifact in $\mathcal{S}$.

Note that all of the above cases assume total binary relations. In practice, cases where R covers $\mathcal{S}$ or $\mathcal{T}$ only partially can be handled, e.g., by making these sets smaller using additional well-formedness constraints or by introducing a special value representing an error. For example, a source artifact that has no proper translation into the target type would be mapped to such an error element in the target type. Furthermore, the above cases are distinguished only in regard to the correspondence between whole artifacts. In practice, the artifact relations also need to establish correspondence between the structures within the artifacts, i.e., the correspondence between the elements and links in one artifact and the elements and links in another artifact. We will explicitly refer to this structural correspondence whenever necessary. Finally, the artifact relations need not be binary, but could be relating three or more sets of artifacts.

Examples. Let us look at some examples of relations among different kinds of artifacts.

Example 1. Simple class diagrams and KM3.

KM3 [9] is a textual notation that can be used for the specification of simple class diagrams. The relation between graphical class diagrams and their textual specifications is a *bijection*. In this example, assuming that the layout of diagrams

and text is irrelevant, artifacts expressed in one language can be translated into the other language without any loss of information.

Example 2. Java and type hierarchy.

A type hierarchy of a Java program is a graph in which *classes* and *interfaces* are nodes and *extends* and *implements* relations are edges. Such a type hierarchy is an abstraction of a Java program because it contains a subset of the information contained in the program and it does not contain any additional information that does not exist in the program. Furthermore, many different Java programs may have the same type hierarchy. Therefore, the relation between a Java program and its type hierarchy is a *function*.

Example 3. Java and XML and Struts Framework-Specific Modeling Language (FSML).

Struts FSML [10] is a modeling language that can be used for describing how Struts' concepts *actions*, *forms*, and *forwards* are implemented in an application consisting of Java code and XML configuration files. A model expressed in the Struts FSML is an abstraction of the code and it can be fully recreated from the code. Actions, forms, and forwards can be implemented in the code in various ways, some of which are equivalent with respect to the model. For example, a Java class is represented in the model as an action if it is a direct or indirect subclass of the Struts' `Action` class. The relation between the code and the model expressed in Struts FSML is a *function*: parts of the code do not have any representation in the model and equivalent ways of implementing actions, forms, and forwards are represented the same way in the model.

Example 4. UML class diagrams and RDBMS.

This example considers UML class diagrams and relational database schemas. The relation between the two languages is a *general relation* because inheritance and associations in class diagrams can be represented in many different ways in database schemas and every database schema can be represented using different class diagrams with or without inheritance [11]. For example, each single class can be mapped to a separate table or an entire class hierarchy can be mapped to a single table. Furthermore, different class hierarchies may still be translated into the same table structure.

Example 5. Statecharts and sequence diagrams.

The relation between statecharts and sequence diagrams is a *general relation* because a statechart can be synthesized from multiple sequence diagrams and a given sequence diagram can be produced by different statecharts.

Example 6. Metamodels and models.

In model-driven software development [12], the syntax of a modeling language is often specified as a class model, which is referred to as a *metamodel*. A metamodel defines all syntactically correct models and a model is syntactically correct if it conforms to its metamodel. As any other software artifacts, metamodels evolve over time. Some changes to the metamodels may break the conformance of existing models, in which case the models need to be updated [13].

The relation between a metamodel and a model is a *general relation* because many models can conform to a single metamodel and a single model can conform to many metamodels. As an example of the latter situation, consider two metamodels representing the same set of models, but one using abstract and concrete classes and the other using concrete classes only.

3 Mappings, Transforms, Transformations, Synchronizers, and Synchronizations

We refer to the specifications of relations among artifacts as *mappings*. Furthermore, we refer to programs that implement mappings as *transforms* and executions of those programs as *transformations*. In this tutorial, we focus on *synchronizers*, which are transforms used for (re-)establishing consistency among related artifacts. Consequently, we refer to the execution of a synchronizer as *synchronization*. Note that not every transform is a synchronizer. For example, *refactorings*, which change the structure of an artifact while preserving the artifact's semantics are transforms, but they are not synchronizers.

Transforms are executable programs, which may be interactive. For example, they may seek additional inputs from the user to decide among possible alternative results. In this tutorial, we model transforms as *computable functions*, where any additional inputs are given to the functions up-front as arguments. In particular, we represent interactive choices as *decision functions* that are passed as parameters to the transforms.

In the following sections we present various kinds of heterogeneous synchronizers that can be used to synchronize two artifacts, which we refer to as *source* and *target*. At the highest level, a synchronizer falls into one of the three distinct categories: *unidirectional*, *bidirectional*, and *bidirectional with reconciliation* (cf. Figure 1). The three alternatives are represented as a *feature model* [14, 15]. A feature model is a hierarchy of common and variable features characterizing the set of instances of a concept that is represented by the root of the hierarchy. In this tutorial, the features provide a terminology and a representation of the design choices for heterogeneous synchronizers. The subset of the feature model notation used in this tutorial is explained in Table 1. The three categories of synchronizers are modeled in Figure 1 as a group of three alternative features. Each of these alternative features is actually a reference to a more refined feature model that is presented later.

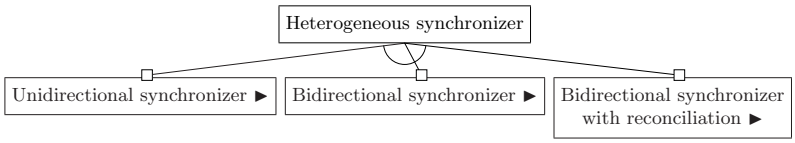

Fig. 1. Artifact synchronization synchronizers

Table 1. Feature modeling notation used in this tutorial

Symbol	Explanation
	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
	Solitary feature with cardinality [0..1], i.e., *optional* feature
	Reference to feature F
	XOR feature group (groups alternative features)
	Grouped feature (a feature under a feature group)

Unidirectional synchronizers synchronize the target artifact with the source artifact. *Bidirectional synchronizers* (without reconciliation) can be used to synchronize the target artifact with the source artifact and vice versa. They synchronize in one direction at a time, meaning that they are most useful if only one of the artifacts was changed since the last synchronization. Bidirectional synchronizers can also be used when both artifacts have changed since the last synchronization; however, they cannot be used to resolve conflicting changes to both artifacts, as one artifact acts as a slave and its changes may get overridden. Finally, *bidirectional synchronizers with reconciliation* can be used to synchronize both artifacts at the same time. Thus, these synchronizers are also applicable in situations where both artifacts were changed since the last synchronization and they can be used for conflict resolution in both directions.

4 Unidirectional Synchronizers

Unidirectional synchronization from $\mathcal{S}$ to $\mathcal{T}$ enforcing the relation $R \subseteq \mathcal{S} \times \mathcal{T}$ involves up to four artifacts (cf. Figure 2):

1. $S_{\mathcal{S}}$ is the *original source artifact*, i.e., the version of the source artifact before it was modified by the developer;
2. $T_{\mathcal{T}}$ is the *original target artifact*, i.e., the version of the target artifact that co-existed with the original source artifact;
3. $S'_{\mathcal{S}}$ is the *new source artifact*, i.e., the version of the source artifact after it was modified by the developer; and
4. $T'_{\mathcal{T}}$ is the *new target artifact*, i.e., the version of the target artifact after synchronization with the new source artifact.

The first three of these artifacts are the ones that typically exist before the synchronization occurs. However, the first two are optional since the new source could have been created from scratch and the original target might have not

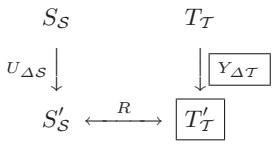

Fig. 2. Artifacts involved in unidirectional synchronization

been yet created. Note that we use the convention of marking new versions of artifacts by a prime.

The fourth artifact, $T'_{\mathcal{T}}$, is the new target that needs to be computed during the synchronization. The enclosing boxes in Figure 2 indicate elements that are computed during the synchronization. The arrows pointing downwards in the figure denote updates: $U_{\Delta\mathcal{S}}$ is the update applied to the original source and $Y_{\Delta\mathcal{T}}$ is the target update resulting from the synchronization. The double-headed arrow between the new artifacts indicates that they are consistent, i.e., $(S'_{\mathcal{S}}, T'_{\mathcal{T}}) \in R$. Note that, in general, the original artifacts $S_{\mathcal{S}}$ and $T_{\mathcal{T}}$ do not have to be consistent; however, some synchronizers might impose such a requirement.

A *unidirectional synchronizer* from $\mathcal{S}$ to $\mathcal{T}$ implementing the relation $R \subseteq \mathcal{S} \times \mathcal{T}$ computes the new target $T'_{\mathcal{T}}$ given the new source $S'_{\mathcal{S}}$, and optionally the original source $S_{\mathcal{S}}$ and the original target $T_{\mathcal{T}}$, as inputs, such that the new source and the new target are consistent, i.e., $(S'_{\mathcal{S}}, T'_{\mathcal{T}}) \in R$. Note that the new source can also be passed as input to the synchronizer indirectly by passing both the original source and the update of the source as inputs. Furthermore, some synchronizer variants require the original source and the original target to be consistent.

Unidirectional synchronizers can be implemented using different operators and the choices depend first and foremost on the cardinality of the end of the relation in the direction of which the synchronizers are executed. In particular, a synchronizer can be executed towards the cardinality of one, which we refer to as *to-one* case, and towards the cardinality of many, which we refer to as *to-many* case (cf. Figure 3).

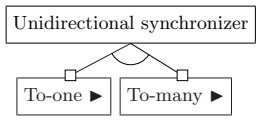

Fig. 3. Unidirectional synchronizers

The to-one case corresponds to the situation where the mapping between source and target is a function from source to target, which also covers the case of a bijection. The mapping clearly specifies a single target artifact $T'_{\mathcal{T}}$ that a synchronizer has to return for a given source artifact $S'_{\mathcal{S}}$. The to-many case

corresponds to the situation where the mapping between source and target is not a function in the source-to-target direction. In other words, the relation is either a function in the target-to-source direction or a general relation. Consequently, the mapping may specify several alternative target artifacts that a synchronizer could return for a given source artifact. Since all synchronizers are functions returning only a single synchronization result, to-many synchronizers will require a mechanism for selecting one target artifact from the set of possible alternatives.

4.1 Unidirectional synchronizers in to-one direction

The unidirectional to-one case could be described as computing a "disposable view", where the target $T'_{\mathcal{T}}$ is fully determined by the source $S'_{\mathcal{S}}$. In other words, the source-to-target mapping is a function and the target can be automatically re-computed whenever needed based on the source $S'_{\mathcal{S}}$ only.

In general, a disposable view can be computed in an *incremental* or a *non-incremental* fashion. The non-incremental approach implies that the view is completely re-computed whenever the source is modified, whereas the incremental approach involves computing only the necessary updates to the existing view and applying these updates. As a result, all incremental synchronizers take the original target as a parameter.

All to-one synchronizers are *original-target-independent*, meaning that the computed new target does not depend on the original target in a mathematical sense. Although the incremental to-one synchronizers take the original target as a parameter, the new target depends only on the new source because the relationship between the new source and the new target is a function. The original target is used by the synchronizer implementation purely to improve performance, which is achieved by reusing structures from the original target and avoiding recomputing these structures. We present examples of *original-target-dependent* synchronizers in Section 4.2.

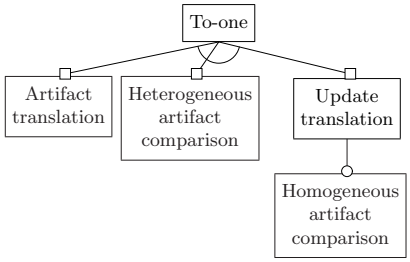

Fig. 4. Operators used in to-one unidirectional synchronizers

Depending on the operator that is used to translate between source and target artifact types, we distinguish among three fundamental ways of realizing unidirectional to-one synchronizers (cf. Figure 4). The first synchronizer variant

is non-incremental and it uses *artifact translation*, an operator that translates an entire source artifact into a consistent target artifact. The other two variants are incremental. The second variant uses *heterogeneous artifact comparison*, an operator that directly compares two artifacts of *different* types and produces an update that can be applied to the second artifact in order to make it consistent with the first artifact. The third variant uses *update translation*, an operator that translates an update to the source artifact into a consistent update of the target artifact. In addition, update translation expects the original source and the original target to be consistent. The artifacts involved in the synchronization using update translation are shown in Figure 5. The input artifacts are underlined. As an option, the transform may use *homogeneous artifact comparison* to compute the source update as a difference between the original and the new source.

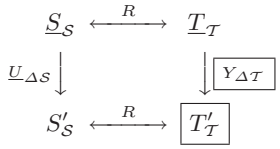

Fig. 5. Artifacts involved in unidirectional synchronization using update translation

Artifact translation. The non-incremental variant of the to-one synchronizer uses an operator that translates a source artifact into a consistent target artifact.

Operator 1 *Artifact translation:* $AT_{\mathcal{S},\mathcal{T}} : \mathcal{S} \rightarrow \mathcal{T}$. *For an artifact* $S_{\mathcal{S}}$*, the operator* $AT_{\mathcal{S},\mathcal{T}}(S_{\mathcal{S}})$ *computes* $S_{\mathcal{T}}$ *such that* $(S_{\mathcal{S}}, S_{\mathcal{T}}) \in R$.

In this tutorial, operators are defined generically over artifact types and the type parameters are specified as subscripts. For example, the operator $\mathrm{AT}_{\mathcal{S},\mathcal{T}}$ has the artifact type parameters $\mathcal{S}$ and $\mathcal{T}$ and these parameters are used in the operator's signature.

We are now ready to state the non-incremental to-one synchronizer. We present all synchronizers using the form *input+ precondition∗* $\Longrightarrow$ *computation* $\Longrightarrow$ *output+*, which makes the input artifact(s), the precondition(s) (if any), the computation steps, and the output artifact(s) explicit. This form may seem too verbose for the following simple synchronizer, but its advantages become apparent for more complex synchronizers.

Synchronizer 1 *Unidirectional, non-incremental, and to-one synchronizer using artifact translation:*
$S1_{\mathcal{S},\mathcal{T}} : \mathcal{S} \rightarrow \mathcal{T}$

$$S'_{\mathcal{S}} \quad \Longrightarrow \quad T'_{\mathcal{T}} = AT_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}}) \quad \Longrightarrow \quad T'_{\mathcal{T}}$$

In this non-incremental variant, the new source artifact is translated into the new target artifact, which then replaces the original target artifact.

Examples for Synchronizer 1.

Example 7. Type hierarchy.

Examples of Synchronizer 1 are type hierarchy extractors for object-oriented programs (cf. Example 2). Such extractors are offered by many integrated development environments (IDEs).

Example 8. Reverse engineering in FSMLs.

Another example of Synchronizer 1 is reverse engineering of framework-based Java code in FSMLs (cf. Example 3). The result of reverse engineering is a framework-specific model that describes how framework abstractions are implemented in the application code [10,16]. For any application code, a unique model is retrieved using code queries.

Example 9. Lenses in Harmony.

Synchronizer 1 corresponds to the *get* function in Lenses [17]. In Lenses, the source-to-target relationship is many-to-one and the target is also referred to as *view*. A *get* function takes the new source and creates the corresponding new view for it. A full lens, as shown later, is a bidirectional synchronizer and consists of two functions: *get* and *putback*.

Updates. Incremental synchronization can be achieved either by coercing the original target artifact into conformance with the new source artifact or by translating updates of the source artifact into the updates of the target artifact. Both variants require the notion of an *update*.

Definition 2. UPDATE. *An update* $U : \mathcal{S} \rightharpoonup \mathcal{S}$ *for artifact(s) of type* $\mathcal{S}$ *is a partial function that is defined for at least one artifact* $S_{\mathcal{S}}$*. Artifacts on which an update is defined are referred to as* reference artifacts *of that update.*

The intuition behind an update is that it connects an original version of an artifact with its new version, e.g., $S'_{\mathcal{S}} = U_{\Delta\mathcal{S}}(S_{\mathcal{S}})$. Note that we abbreviate the space of all partial functions $\mathcal{S} \rightharpoonup \mathcal{S}$ as $\Delta\mathcal{S}$ and we use this abbreviation to specify the type of an update.

The size of the set of reference artifacts of an update can vary. An extreme case is when an update is applicable to only a single artifact. A more practical solution is to implement updates so that they can be applied to a number of artifacts that share certain characteristics. For example, an update could be defined so that it applies to all artifacts that contain a certain structure that the update modifies.

In practice, we can think of an update as a program that takes the original version of an artifact and returns its new version. The update instructions, such as inserting or removing elements, could be recorded while the user edits the original artifact. The recorded sequence can then be applied to a reference artifact, e.g., the original artifact.

Alternatively, an update can be computed using a *homogeneous artifact comparison* operator, which takes an original version of an artifact and its new version and returns an update connecting the two. We refer to this comparison operator as *homogeneous* since it takes two artifacts of the same type.

Operator 2 *Homogeneous artifact comparison:* $AC_{\mathcal{S}} : \mathcal{S} \times \mathcal{S} \rightarrow \Delta\mathcal{S}$*. For artifacts* $S_{\mathcal{S}}$ *and* $S'_{\mathcal{S}}$*, the operator* $AC_{\mathcal{S}}(S_{\mathcal{S}}, S'_{\mathcal{S}})$ *computes* $U_{\Delta\mathcal{S}}$ *such that* $S'_{\mathcal{S}} = U_{\Delta\mathcal{S}}(S_{\mathcal{S}})$*.*

We further discuss the design choices for creating and representing updates in Section 8.1.

Heterogeneous artifact comparison. The first incremental synchronizer uses *heterogeneous artifact comparison*, an operator that directly compares two artifacts of *different* types and produces an update that can be applied to the second artifact in order to make it consistent with the first artifact.

Operator 3 *Heterogeneous artifact comparison:* $AC_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{T} \rightarrow \Delta\mathcal{T}$*. For artifacts* $S'_{\mathcal{S}}$ *and* $T_{\mathcal{T}}$*, the operator* $AC_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}}, T_{\mathcal{T}})$ *computes an update* $U_{\Delta\mathcal{T}}$ *such that* $(S'_{\mathcal{S}}, U_{\Delta\mathcal{T}}(T_{\mathcal{T}})) \in R$*.*

The incremental synchronizer using *heterogeneous artifact comparison* takes the original target in addition to the new source as an input and produces the new target.

Synchronizer 2 *Unidirectional, incremental, original-target-independent, and to-one synchronizer using heterogeneous artifact comparison:*
$S2_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{T}$

$$S'_{\mathcal{S}}, T_{\mathcal{T}} \implies \begin{array}{l} U_{\Delta\mathcal{T}} = AC_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}}, T_{\mathcal{T}}) \\ T'_{\mathcal{T}} = U_{\Delta\mathcal{T}}(T_{\mathcal{T}}) \end{array} \implies T'_{\mathcal{T}}$$

In general, the synchronizer needs to analyze the original target with respect to the new source, compute the updates, and apply the updates to the original target. Although the above formulation separates the update computation and application, all these actions could be performed in one pass over the existing target by synchronizing the target in place.

Note that the above operator and synchronizer assume the situation shown in Figure 2, where $S_{\mathcal{S}}$ and $T_{\mathcal{T}}$ do not have to be consistent. However, in cases where $S_{\mathcal{S}}$ and $T_{\mathcal{T}}$ are consistent and a small source update $U_{\Delta\mathcal{S}}$ corresponds to a small target update $Y_{\Delta\mathcal{T}}$, the performance savings from reusing $T_{\mathcal{T}}$ in the computation of $T'_{\mathcal{T}}$ are expected to be high.

Update translation. The second incremental synchronizer assumes that the original source $S_{\mathcal{S}}$ and the original target $T_{\mathcal{T}}$ are consistent (cf. Figure 5). The key idea behind this synchronizer is to translate the update of the source into a consistent update of the target.

Definition 3. Consistent Updates. *Two updates* $U_{\Delta\mathcal{S}}$ *and* $Y_{\Delta\mathcal{T}}$ *of two consistent reference artifacts* $S_{\mathcal{S}}$ *and* $T_{\mathcal{T}}$*, respectively, are consistent iff application of both updates results in consistent artifacts, i.e.,* $(U_{\Delta\mathcal{S}}(S_{\mathcal{S}}), Y_{\Delta\mathcal{T}}(T_{\mathcal{T}})) \in R$*.*

We can now define the *update translation* operator. The operator takes not only the update of the source artifact but also the original source and target artifacts as parameters. The reason is that consistent updates are defined with respect to these artifacts.

Operator 4 *Update translation:* $UT_{\mathcal{S},\mathcal{T}} : \Delta\mathcal{S} \times \mathcal{S} \times \mathcal{T} \to \Delta\mathcal{T}$*. For consistent artifacts* $S_\mathcal{S}$ *and* $T_\mathcal{T}$*, i.e.,* $(S_\mathcal{S}, T_\mathcal{T}) \in R$*, and an update* $U_{\Delta\mathcal{S}}$ *of the source artifact* $S_\mathcal{S}$*, the operator* $UT_{\mathcal{S},\mathcal{T}}(U_{\Delta\mathcal{S}}, S_\mathcal{S}, T_\mathcal{T})$ *computes an update* $U_{\Delta\mathcal{T}}$ *of the target artifact* $T_\mathcal{T}$ *such that* $U_{\Delta\mathcal{S}}$ *and* $U_{\Delta\mathcal{T}}$ *are consistent for* $S_\mathcal{S}$ *and* $T_\mathcal{T}$*, i.e.,* $(U_{\Delta\mathcal{S}}(S_\mathcal{S}), U_{\Delta\mathcal{T}}(T_\mathcal{T})) \in R$*.*

Using the update translation operator we can define the second incremental synchronizer as follows.

Synchronizer 3 *Unidirectional, incremental, original-target-independent, and to-one synchronizer using update translation:*
$S3_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \to \mathcal{T}$

$$
\begin{array}{lll}
S_\mathcal{S}, U_{\Delta\mathcal{S}}, T_\mathcal{T} & & \\
(S_\mathcal{S}, T_\mathcal{T}) \in R & \implies & U_{\Delta\mathcal{T}} = UT_{\mathcal{S},\mathcal{T}}(U_{\Delta\mathcal{S}}, S_\mathcal{S}, T_\mathcal{T}) \\
 & & T'_\mathcal{T} = U_{\Delta\mathcal{T}}(T_\mathcal{T}) \qquad \implies \quad T'_\mathcal{T}
\end{array}
$$

The synchronizer requires the original source and the original target, which have to be consistent, and an update to the original source.

Note that the update of the source artifact $U_{\Delta\mathcal{S}}$ can also be computed by comparing the new source against the original source using the homogeneous artifact comparison. This possibility allows us to rewrite Synchronizer 3 as follows.

Synchronizer 4 *Unidirectional, incremental, original-target-independent, and to-one synchronizer using homogeneous artifact comparison and update translation:*
$S4_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{S} \times \mathcal{T} \to \mathcal{T}$

$$
\begin{array}{lll}
S_\mathcal{S}, S'_\mathcal{S}, T_\mathcal{T} & & \\
(S_\mathcal{S}, T_\mathcal{T}) \in R & \implies & U_{\Delta\mathcal{S}} = AC_\mathcal{S}(S_\mathcal{S}, S'_\mathcal{S}) \\
 & & U_{\Delta\mathcal{T}} = UT_{\mathcal{S},\mathcal{T}}(U_{\Delta\mathcal{S}}, S_\mathcal{S}, T_\mathcal{T}) \\
 & & T'_\mathcal{T} = U_{\Delta\mathcal{T}}(T_\mathcal{T}) \qquad \implies \quad T'_\mathcal{T}
\end{array}
$$

An example for Synchronizer 3.

Example 10. Live Update.

An example implementation of Synchronizer 3 is *live update* [18]. In live update, a target artifact is first obtained by executing a transformation on the source artifact. The transformation execution context is preserved and later used for incremental update of the target artifact in response to an update of the source artifact. The update translation operator works by locating the points in the transformation execution context that are affected by the source update. Update application works by resuming the transformation from the identified points with the new values from the source.

4.2 Unidirectional synchronizers in to-many direction

The operators used in unidirectional to-many synchronizers are summarized in Figure 6. The feature diagram is similar to the diagram for the to-one case in Figure 4 except that each operator appears as a "with choice" variant. Furthermore, an additional variant using a special merge operator was added (on the bottom left in the diagram). The to-many case implies that a given source artifact may correspond to multiple target artifacts. Thus, each translating operator in its "with choice" variant produces a set of possible targets rather than a single target. Consequently, all to-many synchronizers need a decision function as an additional input that they use to select only one result from the set of possible targets.

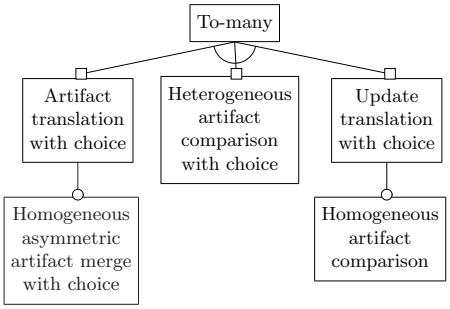

Fig. 6. Operators used in to-many unidirectional synchronizers

Like their to-one counterparts, the to-many synchronizers can be non-incremental or incremental. However, whereas all to-one synchronizers are original-target-independent, the to-many synchronizers have only one original-target-independent variant. The remaining ones are *original-target-dependent*, which means that values and structures from the original target are used in the computation of the new target and the resulting new target depends both on the new source and the original target.

The dependency on the original target is desirable for to-many synchronizers if the target can be edited by developers. The original-target-dependent synchronizers can preserve parts of the original target that have no representation in the source artifact type when the target is updated. These parts could be added to the target and edited by developers. Such edits should be preserved during the synchronization of the target in order to preserve developers' work.

The first two unidirectional to-many synchronizers are non-incremental and correspond to the left branch of the feature diagram in Figure 6. The first one is original-target-independent. It uses *artifact translation with choice* to translate the new source into a set of possible new targets and selects one target using a decision function. The second synchronizer is original-target-dependent. It also uses *artifact translation with choice* to translate the new source into a set of

possible new targets, but then it merges the selected new target with the original target. For this purpose, it uses *homogeneous asymmetric artifact merge with choice*, an operation which merges a slave artifact with a master artifact while preserving a certain property of the master artifact. As a result, some structures from the original target can be preserved.

The remaining synchronizers are incremental and operate similarly to their to-one counterparts. However, unlike the latter, they are original-target-dependent. The first incremental variant uses *heterogeneous artifact comparison with choice*. The other one uses *update translation with choice*. The source update may optionally be computed using *homogeneous artifact comparison* between the original source and the new source.

Artifact translation with choice. Let us first consider the first non-incremental variant. This variant requires an artifact translation operator that returns a set of possible results. Note that $\mathcal{P}^+(\mathcal{T})$ denotes the power set of the set $\mathcal{T}$ without the empty set. We mark all "with choice" variants of operators with $*$.

Operator 5 *Artifact translation with choice:* $AT^*_{\mathcal{S},\mathcal{T}} : \mathcal{S} \to \mathcal{P}^+(\mathcal{T})$. *For an artifact* $S'_\mathcal{S}$, *the operator* $AT^*_{\mathcal{S},\mathcal{T}}(S'_\mathcal{S})$ *computes* $\{S'_\mathcal{T} : (S'_\mathcal{S}, S'_\mathcal{T}) \in R\}$.

A single resulting artifact can be chosen using a *decision* function.

Definition 4. DECISION. *A decision for an artifact type* $\mathcal{T}$ *is a function* $D_{\mathcal{D}_\mathcal{T}} : \mathcal{P}^+(\mathcal{T}) \to \mathcal{T}$ *such that* $\forall X \in \mathcal{P}^+(\mathcal{T}) : D_{\mathcal{D}_\mathcal{T}}(X) \in X$. *We denote a set of all decision functions for an artifact type* $\mathcal{T}$ *as* $\mathcal{D}_\mathcal{T}$.

Intuitively, a decision function chooses one artifact out of a set of artifacts of a given type. It models both the situation where the user makes a choice interactively or the situation where a choice is made based on some predefined criteria or default settings. We discuss some design choices for implementing decision functions in Section 8.5.

Synchronizer 5 *Unidirectional, non-incremental, original-target-independent, and to-many synchronizer using artifact translation with choice:*
$S5_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{D}_\mathcal{T} \to \mathcal{T}$

$$S'_\mathcal{S}, D_{\mathcal{D}_\mathcal{T}} \implies T'_\mathcal{T} = D_{\mathcal{D}_\mathcal{T}}(AT^*_{\mathcal{S},\mathcal{T}}(S'_\mathcal{S})) \implies T'_\mathcal{T}$$

Synchronizer 5 is only of interest for scenarios where the target artifact is not supposed to be manually edited, e.g., code generation in model compilation.

Examples for Synchronizer 5.

Example 11. Code and model compilation.

In compilation, the resulting artifacts, regardless if they are machine code, byte code, or code in a high-level programming language, depend on many settings of the compiler such as optimizations or coding style. Although the relation between the source and target artifacts is many-to-many, the selection of options allows the synchronizer (the compiler) to produce a single result.

Example 12. Pretty printing.

Similarly to the previous example, many code style options influence the result of pretty printing an abstract syntax tree representing a program.

Homogeneous asymmetric artifact merge. Unlike the first variant, which completely replaces the original target with the new one, the second non-incremental variant uses a merge operator to preserve some structures from the original target.

The merge operator is homogeneous as it merges two artifacts of the same type. It is also asymmetric as one of the artifacts is a *master artifact* and the other one is a *slave artifact*, that is, the operator merges the master and slave artifacts in such a way that the result of the merge satisfies the same property as the master artifact does. The merge can be implemented in two ways: by copying structures from the master artifact to the slave artifact or vice versa.

In our context, the slave artifact will be the original target and the master artifact will be the target obtained by translating the new source into the target artifact type. The property of the master artifact to be preserved will be its consistency with the new source artifact.

We model artifact properties as binary functions.

Definition 5. Artifact Property. *A property function ϕ for artifacts of type $\mathcal{T}$ is a function with the following signature $\phi : \mathcal{T} \to \{0, 1\}$. We say that the property ϕ holds for an artifact $T_\mathcal{T}$ iff $\phi(T_\mathcal{T}) = 1$. We denote the set of all properties for an artifact type $\mathcal{T}$ as $\Phi_\mathcal{T}$.*

Operator 6 *Homogeneous asymmetric artifact merge with choice: $M^*_\mathcal{T} : \mathcal{T} \times \mathcal{T} \times \Phi_\mathcal{T} \to \mathcal{P}^+(\mathcal{T})$. For a slave artifact $T_\mathcal{T}$, a property ϕ, and a master artifact $S_\mathcal{T}$ such that $\phi(S_\mathcal{T}) = 1$, the operator $M^*_\mathcal{T}(T_\mathcal{T}, S_\mathcal{T}, \phi)$ computes a non-empty subset of $\{T'_\mathcal{T} : \phi(T'_\mathcal{T}) = 1\}$. The elements of the subset preserve structures from both master and slave artifacts according to some criteria.*

The key intention behind this operator, which is only partially captured by the formal part, is that the resulting set contains artifacts obtained by combining structures from both input artifacts such that each of the artifacts in the resulting set satisfies the input property. The merge returns a subset of all the artifacts satisfying the property, meaning that some artifacts satisfying the property are rejected if they do not preserve structures from both artifacts well enough according to some criteria. The operator returns a set of artifacts rather than a single artifact since, in general, there may be more than one satisfactory way to merge the input artifacts.

Synchronizer 6 *Unidirectional, non-incremental, original-target-dependent, and to-many synchronizer using artifact translation with choice and homogeneous asymmetric artifact merge with choice:*

$$S6_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{T} \times \mathcal{D}_{\mathcal{T}} \times \mathcal{D}_{\mathcal{T}} \to \mathcal{T}$$

$$S'_{\mathcal{S}}, T_{\mathcal{T}}, D_{\mathcal{D}_{\mathcal{T}}}, E_{\mathcal{D}_{\mathcal{T}}} \implies \begin{array}{l} S'_{\mathcal{T}} = D_{\mathcal{D}_{\mathcal{T}}}(AT^{*}_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}})) \\ T'_{\mathcal{T}} = E_{\mathcal{D}_{\mathcal{T}}}(M^{*}_{\mathcal{T}}(T_{\mathcal{T}}, S'_{\mathcal{T}}, \phi_{\Phi_{\mathcal{T}}})) \end{array} \implies T'_{\mathcal{T}}$$

$$\textit{where} \quad \phi_{\Phi_{\mathcal{T}}}(T) = \begin{cases} 1 & \textit{if } (S'_{\mathcal{S}}, T) \in R \\ 0 & \textit{otherwise} \end{cases}$$

The synchronizer takes two decision functions. The first function selects a translation of the new source artifact into the target artifact type from the alternatives returned by the artifact translation with choice. The selected translation $S'_{\mathcal{T}}$ is then merged with the original target artifact, where the property passed to the merge is consistency with the new source artifact $S'_{\mathcal{S}}$. The second decision function is used to select one target artifact from the alternatives returned by the merge.

In practice, the decision functions are likely to be realized as default settings allowing the entire synchronizer to be executed automatically. Furthermore, practical implementations, while focusing on preserving manual edits from the original target, often do not restore the full consistency during the merge. In such cases, the developers are expected to complete the merge by manual editing.

An example for Synchronizer 6.

Example 13. JET and JMerge.

An example implementation of artifact merge with choice is *JMerge*, which is a part of Java Emitter Templates (JET) [19]. JET is a template-based code generation framework in Eclipse.

JMerge can be used to merge an old version of Java code (slave artifact) with a new version (master artifact), such that developers can control which parts of the old versions get overridden by the corresponding parts from the new version. JMerge replaces Java classes, methods, and fields of the slave artifact that are annotated with `@generated` with their corresponding new versions from the master artifact. Developers can remove the `@generated` annotation from the elements they modify in order to preserve their modifications during subsequent merges. The behavior of JMerge is parameterized with a set of rules, which is an implementation of the decision function $\mathrm{E}_{\mathcal{D}_{\mathcal{T}}}$. JMerge is not concerned with preserving the consistency of the master artifact with the new source, meaning that the merged result might require manual edits in order to make it consistent. However, JMerge guarantees that all program elements in the slave that are not annotated with the `@generated` annotation remain unchanged in the merged result.

The code generator of Eclipse Modeling Framework (EMF) [20] implements Synchronizer 6 and uses JMerge as an implementation of the merge operator. The code generator is based on JET and takes a new EMF model as an input, which is the new source artifact $S'_{\mathcal{S}}$. Code generation is controlled by a separate *generator model*, which specifies both global generation options and options that

are specific to some source model elements. The latter can be thought of as decorations or mark-up of the source elements, but ones that are stored in a separate artifact. Effectively, the generator model corresponds to the decision function $\mathrm{D}_{\mathcal{D}_\mathcal{T}}$. The code generator emits the Java code implementing the model, i.e., $S'_\mathcal{T}$. JMerge is then used to merge the freshly-generated code $S'_\mathcal{T}$ (master artifact) with the original Java code $T_\mathcal{T}$ (slave artifact) that may contain developer's customizations. The resulting new Java code $T'_\mathcal{T}$ is now synchronized with the new model in the sense that all code elements annotated with the `@generated` annotation were replaced with the code elements generated from the new model.

The JMerge approach is an example of the concept of *protected blocks*. Protected blocks are specially marked code sections that are preserved during code re-generation. In JMerge, protected blocks are marked by virtue of not being annotated with `@generated`.

Heterogeneous artifact comparison. Analogously to the incremental synchronizers from the previous section, incremental to-many synchronizers can be realized using either heterogeneous comparison or update translation. However, both operators need to be modified to produce sets of results.

Operator 7 *Heterogeneous artifact comparison with choice:* $AC^*_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{T} \to \mathcal{P}^+(\Delta\mathcal{T})$. *For artifacts* $S'_\mathcal{S}$ *and* $T_\mathcal{T}$*, the operator* $AC^*_{\mathcal{S},\mathcal{T}}(S'_\mathcal{S}, T_\mathcal{T})$ *computes a non-empty subset of* $\{U_{\Delta\mathcal{T}} : (S'_\mathcal{S}, U_{\Delta\mathcal{T}}(T_\mathcal{T})) \in R\}$. *The elements of the subset preserve structures from* $T_\mathcal{T}$ *according to some criteria.*

We can now state the first incremental synchronizer as follows.

Synchronizer 7 *Unidirectional, incremental, original-target-dependent, and to-many synchronizer using heterogeneous artifact comparison with choice:* $S7_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{T}} \to \mathcal{T}$

$$S'_\mathcal{S}, T_\mathcal{T}, D_{\mathcal{D}_{\Delta\mathcal{T}}} \implies \begin{array}{l} U_{\Delta\mathcal{T}} = D_{\mathcal{D}_{\Delta\mathcal{T}}}(AC^*_{\mathcal{S},\mathcal{T}}(S'_\mathcal{S}, T_\mathcal{T})) \\ T'_\mathcal{T} = U_{\Delta\mathcal{T}}(T_\mathcal{T}) \end{array} \implies T'_\mathcal{T}$$

An example for Synchronizer 7.

Example 14. Lenses in Harmony.

Synchronizer 7 corresponds to the *putback* function in Lenses [17]. In Lenses, the source-to-target relationship is many-to-one and *putback* is used in the target-to-source direction. In other words, *putback* is a unidirectional *to-many* synchronizer. The function takes the new view and the original source and returns the new source. A full lens combines *putback* with *get* (cf. Example 9) to form a bidirectional synchronizer (cf. Example 18).

Update translation with choice. The second incremental variant uses update translation with choice.

Operator 8 *Update translation with choice:* $UT^*_{\mathcal{S},\mathcal{T}} : \Delta\mathcal{S} \times \mathcal{S} \times \mathcal{T} \to \mathcal{P}^+(\Delta\mathcal{T})$. *For two consistent artifacts* $S_\mathcal{S}$ *and* $T_\mathcal{T}$ *and an update* $U_{\Delta\mathcal{S}}$ *of* $S_\mathcal{S}$*, the operator* $UT^*_{\mathcal{S},\mathcal{T}}(U_{\Delta\mathcal{S}}, S_\mathcal{S}, T_\mathcal{T})$ *computes a non-empty subset of* $\{U_{\Delta\mathcal{T}} : (U_{\Delta\mathcal{S}}(S_\mathcal{S}), U_{\Delta\mathcal{T}}(T_\mathcal{T})) \in R\}$. *The elements of the subset preserve structures from* $T_\mathcal{T}$ *according to some criteria.*

Synchronizer 8 *Unidirectional, incremental, original-target-dependent, and to-many synchronizer using update translation with choice:*
$S8_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{T}} \to \mathcal{T}$

$$
\begin{array}{lll}
S_\mathcal{S}, U_{\Delta\mathcal{S}}, T_\mathcal{T}, D_{\mathcal{D}_{\Delta\mathcal{T}}} & & \\
(S_\mathcal{S}, T_\mathcal{T}) \in R \implies & U_{\Delta\mathcal{T}} = D_{\mathcal{D}_{\Delta\mathcal{T}}}(UT^*_{\mathcal{S},\mathcal{T}}(U_{\Delta\mathcal{S}}, S_\mathcal{S}, T_\mathcal{T})) & \\
& T'_\mathcal{T} = U_{\Delta\mathcal{T}}(T_\mathcal{T}) & \implies \quad T'_\mathcal{T}
\end{array}
$$

Examples for Synchronizer 8.

Example 15. Incremental code update in FSMLs.

An example of Synchronizer 8 is incremental code update in FSMLs [10]. During forward propagation of model updates to code, code update transformations are executed for every added, modified, or removed model element. This translation of element updates into corresponding code updates is an example of an update translation function. Different code updates can be applied for a given model update depending on the desired implementation variant. An example of an implementation variant is the creation of an assignment to a field either as a separate statement or as an expression of the field's initializer. The variants can be selected based on source model annotations that are provided by default and can also be modified by the developer. This annotation mechanism represents an implementation of the decision function $\mathrm{D}_{\mathcal{D}_{\Delta\mathcal{T}}}$.

Example 16. Co-evolution of models with metamodels.

Wachsmuth [13] describes an approach to the synchronization of models in response to certain well-defined kinds of updates in their metamodels. The updates are classified into *refactoring*, *construction*, and *destruction*. These metamodel updates are then translated into the corresponding updates of the models. The model updates are an example of updates whose sets of reference artifacts contain more than one artifact (cf. Definition 2).

5 Bidirectional Synchronizers

Propagating change only in one direction is often not practical as certain changes may only be possible in certain artifacts. *Bidirectional synchronization* involves propagating changes in both directions using bidirectional synchronizers. Bidirectional synchronization is also referred to as *round-trip engineering* [21–23].

In this section, we focus on synchronization where changes to one artifact are propagated to the other artifact only in one direction at a time, whereas in

the next section we focus on synchronization in which changes to both artifacts can be reconciled and propagated in both directions at once.

A sample bidirectional synchronization scenario with a source-to-target synchronization followed by a target-to-source synchronization is shown in Figure 7. The results of the first synchronization are placed in boxes with subscript one and the results of the second synchronization are placed in boxes with subscript two. The first synchronization is executed in response to update $U_{\Delta\mathcal{S}}$, and the second synchronization is executed in response to update $Y'_{\Delta\mathcal{T}}$.

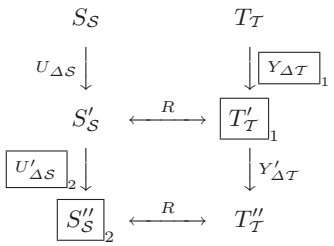

Fig. 7. Bidirectional synchronization scenario with a source-to-target synchronization followed by a target-to-source synchronization

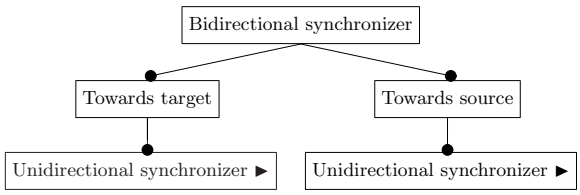

Fig. 8. Bidirectional synchronizer

A bidirectional synchronizer can be thought of as a pair of unidirectional synchronizers, one synchronizer for one direction, as shown in Figure 8. The feature *towards target* represents the unidirectional synchronizer from source to target, and the feature *towards source* represents the unidirectional synchronizer from target to source. Both synchronizers could be constructed separately using a unidirectional language, or they could be derived from a single description in a bidirectional language. We discuss these possibilities in Section 8.6.

Properties. According to Stevens [24], the key property of a pair of unidirectional synchronizers implementing bidirectional synchronization for a given relation is that they are *correct* with respect to the relation. Correctness means

that each synchronizer enforces the relation between the source and target artifacts. Clearly, any pair $(Si_{\mathcal{S},\mathcal{T}}, Sj_{\mathcal{T},\mathcal{S}})$ of the unidirectional synchronizers defined in the previous sections (where i and j may be equal) is correct with respect to R by the definition of the synchronizers.

Another desired property of a synchronization synchronizer is *hippocraticness* [24], meaning that the synchronizer should not modify any of the artifacts if they already are in the relation. The *hippocraticness* property is also referred to as *check-then-enforce*, which suggests that the synchronizer should only enforce the relation if the artifacts are not in the relation.

Note that, in practice, a synchronization step may be partial in the sense that it does not establish full consistency. Artifact developers may choose to synchronize only certain changes at a time and ignore parts of the artifacts that are not yet ready to be synchronized. Therefore, the correctness property only applies to complete synchronization.

Examples of bidirectional synchronizers.

Example 17. Triple Graph Grammars in FUJABA.

Giese and Wagner describe an approach to bidirectional synchronization using Triple Graph Grammars (TGG) [25]. Their approach is implemented in the Fujaba tool suite [26]. TGG rules are expressed using a bi-directional, graphical language. For two models, the user can choose the direction of synchronization. Both models are then matched by TGG rules, which can be viewed as an implementation of the heterogeneous artifact comparison. The updates determined by each rule are applied to the target in a given direction, which amounts to incremental synchronization. The authors assume that the relationship between source and target is a bijection [25, p. 550]. Thus, the approach can be described as $(S2_{\mathcal{S},\mathcal{T}}, S2_{\mathcal{T},\mathcal{S}})$.

Example 18. Lenses in Harmony.

A *lens* [17] is a bidirectional synchronizer for the many-to-one case. It consist of two unidirectional synchronizers: *get* (cf. Example 9) and *putback* (cf. Example 14). In other words, a lens can be described as $(S1_{\mathcal{S},\mathcal{T}}, S7_{\mathcal{T},\mathcal{S}})$. Note that the second synchronizer executes in the target-to-source direction, i.e., the direction towards the end with the cardinality of many, and the artifact at that end can be edited. Consequently, the synchronizer should be one of the unidirectional, to-many, and original-target-dependent synchronizers, which is satisfied by $S7_{\mathcal{T},\mathcal{S}}$.

6 Bidirectional Synchronizers with Reconciliation

In this section we focus on synchronization where both artifacts can be changed simultaneously in-between two consecutive synchronizations and the changes can be reconciled and propagated in both directions during a single synchronization.

Bidirectional synchronization with reconciliation involves up to six artifacts (cf. Figure 9). Four of them are the same as in the case of unidirectional synchronization (cf. Figure 2), except that the original source $S_{\mathcal{S}}$ and the original target $T_{\mathcal{T}}$ are now assumed to be consistent. Furthermore, the new target $T'_{\mathcal{T}}$ is given as a result of a user update $Y_{\Delta\mathcal{T}}$ just as the new source $S'_{\mathcal{S}}$ is given as a result of another user update $U_{\Delta\mathcal{S}}$. The purpose of a bidirectional synchronizer with reconciliation is to compute a *reconciled source artifact* $S''_{\mathcal{S}}$ and a *reconciled target artifact* $T''_{\mathcal{T}}$, such that the two are consistent. In essence, such a synchronizer can also be viewed as a *heterogeneous symmetric merge operation*.

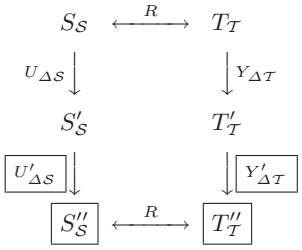

Fig. 9. Artifacts involved in bidirectional synchronization with reconciliation

As in the unidirectional case, some of the four input artifacts may be missing. The two extreme cases are when only the new source or only the new target exist. The synchronization in these cases corresponds to the initial generation of the target artifact or the source artifact, respectively. The case where both original artifacts are missing corresponds to the situation where two artifacts are synchronized for the first time. Note that a "missing" artifact corresponds to a special value that represents a *minimal* artifact, that is, an artifact that contains the minimum structure required by its artifact type. We assume that minimal artifacts of all types are always consistent.

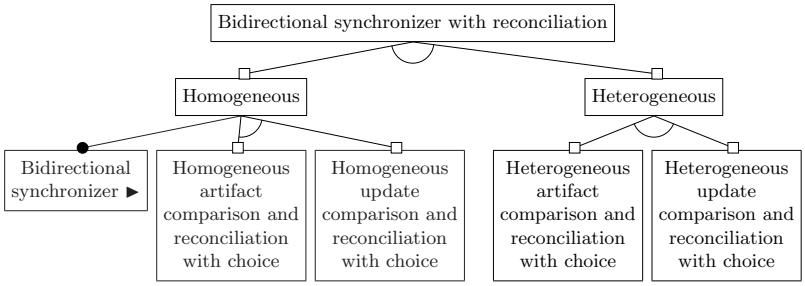

Fig. 10. Bidirectional synchronizer with reconciliation

In general, bidirectional synchronization with reconciliation involves

- translation of updates, artifacts, or both;
- identification of conflicting updates;
- creation of updates that resolve conflicts and reconcile the artifacts; and
- application of the updates.

The identification of conflicting updates and their resolution can be performed in *homogeneous* or *heterogeneous* fashion as indicated in Figure 10.

Homogeneous reconciliation means that updates to both source and target artifacts are compared and then reviewed by the user in terms of one artifact type, which is either the source or the target type. In other words, if the comparison and review (and resolution of potential conflicts) is done on the target side, the new source artifact or the update of the source artifact need to be first translated into the target type. Depending whether the entire artifact or just the update is translated, the comparison and reconciliation is done either by *homogeneous artifact comparison and reconciliation with choice* or its *update* counterpart (cf. Figure 10). Assuming reconciliation on the target side, both operators return an update of the new source artifact (but expressed in the target artifact type) and an update to the new target artifact, such that the two updates reconcile both artifacts. Finally, the first update has to be translated back into the source artifact type and applied to the new source artifact, and the second update is applied to the new target artifact. Note that the translation of artifacts or updates in one direction and the translation of updates in the other direction essentially requires a bidirectional synchronizer, as indicated in Figure 10 by a reference to the feature *bidirectional synchronizer*.

Heterogeneous reconciliation implies a heterogeneous comparison between the artifacts or the updates. A bidirectional synchronizer with heterogeneous reconciliation can be implemented using the operator *heterogeneous artifact comparison and reconciliation with choice* or its *update* counterpart (cf. Figure 10). The operators are similar to their homogeneous counterparts with the difference that they directly compare artifacts of different types and thus do not require a pair of unidirectional synchronizers for both directions.

6.1 Comparison and reconciliation procedures

In general, comparison and reconciliation operators work at the level of individual structural updates that occurred within the overall update of the source artifact $U_{\Delta\mathcal{S}}$ and the overall update of the target artifact $Y_{\Delta\mathcal{T}}$. The updates can be *atomic*, such as element additions, removals, and relocations and attribute value modifications. The updates can also be *composite*, i.e., consisting of other atomic and composite updates.

We categorize updates into *synchronizing*, *propagating*, *consistent*, *conflicting*, *non-reflectable*, and *inverse*. An update in one artifact is *synchronizing* if it establishes the consistency of the artifact with the related artifact. An update in one artifact is *propagating* if it forces a synchronizing update in the related

artifact. Two updates, one in each artifact, are *consistent* if one is a synchronizing update of the other one. On the other hand, two updates, one in each artifact, are *conflicting* if the propagation of one update would override the other one. An update in one artifact is *non-reflectable* if it does not force any synchronizing update in the other artifact. An *inverse* update (intuitively *undo*) for a given update and a reference artifact maps the result of applying the given update to the reference artifact back to the reference artifact.

A *maximal* synchronizer [27] is one that propagates *all* propagating updates. The following strategy is used to compute $U'_{\Delta\mathcal{S}}$ and $Y'_{\Delta\mathcal{T}}$ for achieving maximum synchronization:

- consistent and non-reflectable updates in $U_{\Delta\mathcal{S}}$ and $Y_{\Delta\mathcal{T}}$ are ignored since both artifacts are already consistent with respect to these updates;
- out of several conflicting updates in $U_{\Delta\mathcal{S}}$ and $Y_{\Delta\mathcal{T}}$, exactly one update can be accepted as a propagating update; and
- for each propagating update in $U_{\Delta\mathcal{S}}$, a synchronizing update needs to be included in $Y'_{\Delta\mathcal{T}}$; similarly, for each propagating update in $Y_{\Delta\mathcal{T}}$, a synchronizing update needs to be included in $U'_{\Delta\mathcal{S}}$.

After the updates are classified into consistent, non-reflectable, conflicting, and propagating by the comparison operator, the user typically reviews the classification, resolves conflicts by rejecting some of the conflicting updates, and then the final updates $U'_{\Delta\mathcal{S}}$ and $Y'_{\Delta\mathcal{T}}$ are computed by determining and composing the necessary synchronizing updates. In practice, simple acceptance or rejection of updates might not be sufficient to resolve all conflicts, in which case the input artifacts may need to be manually edited to resolve and merge conflicting updates.

In general, conflict resolution is not the only possible conflict management strategy. Other possibilities include storing all conflicting updates in each reconciled artifact or allowing artifacts to diverge for conflicting updates [27].

6.2 Bidirectional synchronizers for one-to-one relations

Note that due to the need for reconciliation, none of the synchronizers can be fully non-incremental since at least one artifact needs to be updated by update application. Let us first consider a target-incremental synchronizer. This variant requires *homogeneous artifact comparison and reconciliation with choice* operation. The need for choice arises from the fact that conflicts may be resolved in different ways.

Operator 9 *Homogeneous artifact comparison and reconciliation with choice:* $ACR^*_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \times \mathcal{T} \to \mathcal{P}^+(\Delta\mathcal{T} \times \Delta\mathcal{T})$. *For two artifacts* $S'_{\mathcal{T}}$ *and* $T'_{\mathcal{T}}$*, and the reference artifact* $T_{\mathcal{T}}$*, the operator* $ACR^*_{\mathcal{T}}(S'_{\mathcal{T}}, T'_{\mathcal{T}}, T_{\mathcal{T}})$ *computes a non-empty subset of* $\{(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) : U'_{\Delta\mathcal{T}}(S'_{\mathcal{T}}) = Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}})\}$. *Each pair of updates* $(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}})$ *from that subset is such that the updates resolve conflicting changes and enforce all propagating changes from* $U_{\Delta\mathcal{T}}$ *and* $Y_{\Delta\mathcal{T}}$*, where* $U_{\Delta\mathcal{T}} = AC_{\mathcal{T}}(T_{\mathcal{T}}, S'_{\mathcal{T}})$ *and* $Y_{\Delta\mathcal{T}} = AC_{\mathcal{T}}(T_{\mathcal{T}}, T'_{\mathcal{T}})$.

The operator $\mathrm{ACR}^*_{\mathcal{T}}$ performs a three-way comparison of the artifacts and returns a set of pairs of updates. The reference artifact is included in the three-way comparison as it allows precisely determining the kind and location of updates. In particular, it allows determining whether certain updates occurred consistently in both artifacts, inconsistently in both artifacts, or only in one artifact. Each resulting pair of updates modifies both artifacts $S'_{\mathcal{T}}$ and $T'_{\mathcal{T}}$ such that they become identical and all conflicting updates are resolved and all propagating updates are propagated. The second condition is necessary: without it, the operator could simply return updates that could, for example, revert each artifact back to the reference artifact, or even to the minimal artifact. Each pair of resulting updates represents one possible way of reconciling conflicts. The resulting updates are constructed using the strategy given at the end of the previous section.

Now we are ready to formulate the target-incremental synchronizer. Note that all discussed synchronizers perform reconciliation on the target side.

Synchronizer 9 *Bidirectional, target-incremental, and one-to-one synchronizer using artifact translation and homogeneous artifact comparison and reconciliation with choice:*
$S9_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}} \to \mathcal{S} \times \mathcal{T}$

$$
\begin{array}{lll}
S'_{\mathcal{S}}, T'_{\mathcal{T}}, T_{\mathcal{T}}, & & \\
F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}} & \Longrightarrow & S'_{\mathcal{T}} = AT_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}}) \\
 & & (\_, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}(ACR^*_{\mathcal{T}}(S'_{\mathcal{T}}, T'_{\mathcal{T}}, T_{\mathcal{T}})) \\
 & & T''_{\mathcal{T}} = Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}}) \\
 & & S''_{\mathcal{S}} = AT_{\mathcal{T},\mathcal{S}}(T''_{\mathcal{T}}) \qquad \Longrightarrow \quad S''_{\mathcal{S}}, T''_{\mathcal{T}}
\end{array}
$$

In the target-incremental variant, source artifact is first translated into the target artifact type. Next, the operator $\mathrm{ACR}^*_{\mathcal{T}}$ computes new updates for each artifact. In the target-incremental synchronizers, the update for the artifact $S'_{\mathcal{T}}$ is simply ignored. Next, the reconciled target artifact $T''_{\mathcal{T}}$ is created by applying the update $Y'_{\Delta\mathcal{T}}$ to $T'_{\mathcal{T}}$. Finally, the reconciled source artifact $S''_{\mathcal{S}}$ is obtained by translating $T''_{\mathcal{T}}$ back into the artifact type $\mathcal{S}$.

A fully-incremental variant, in which the new source $S'_{\mathcal{S}}$ is incrementally updated, is also possible.

Synchronizer 10 *Bidirectional, fully-incremental, and one-to-one synchronizer using artifact translation, homogeneous artifact comparison and reconciliation with choice, and update translation:*
$S10_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}} \to \mathcal{S} \times \mathcal{T}$

$$
\begin{array}{lll}
S'_{\mathcal{S}}, T'_{\mathcal{T}}, T_{\mathcal{T}}, & & \\
F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}} & \Longrightarrow & S'_{\mathcal{T}} = AT_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}}) \\
 & & (U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}(ACR^*_{\mathcal{T}}(S'_{\mathcal{T}}, T'_{\mathcal{T}}, T_{\mathcal{T}})) \\
 & & T''_{\mathcal{T}} = Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}}) \\
 & & U'_{\Delta\mathcal{S}} = UT_{\mathcal{T},\mathcal{S}}(U'_{\Delta\mathcal{T}}, S'_{\mathcal{T}}, S'_{\mathcal{S}}) \\
 & & S''_{\mathcal{S}} = U'_{\Delta\mathcal{S}}(S'_{\mathcal{S}}) \qquad \Longrightarrow \quad S''_{\mathcal{S}}, T''_{\mathcal{T}}
\end{array}
$$

A fully-incremental variant can also be realized by translating updates instead of translating the entire artifacts. The fully-incremental case requires a *homogeneous update comparison and reconciliation* operator.

Operator 10 *Homogeneous update comparison and reconciliation with choice:* $UCR^*_{\mathcal{T}} : \Delta\mathcal{T} \times \Delta\mathcal{T} \times \mathcal{T} \to \mathcal{P}^+(\Delta\mathcal{T} \times \Delta\mathcal{T})$*. For two updates* $U_{\Delta\mathcal{T}}$ *and* $Y_{\Delta\mathcal{T}}$ *of a reference artifact* $T_{\mathcal{T}}$*, the operator* $UCR^*_{\mathcal{T}}(U_{\Delta\mathcal{T}}, Y_{\Delta\mathcal{T}}, T_{\mathcal{T}})$ *computes a non-empty subset of* $\{(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) : U'_{\Delta\mathcal{T}}(U_{\Delta\mathcal{T}}(T_{\mathcal{T}})) = Y'_{\Delta\mathcal{T}}(Y_{\Delta\mathcal{T}}(T_{\mathcal{T}}))\}$*. Each pair of updates* $(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}})$ *from that subset is such that the updates resolve all conflicting changes and enforce all propagating changes from* $U_{\Delta\mathcal{S}}$ *and* $Y_{\Delta\mathcal{T}}$*.*

Synchronizer 11 *Bidirectional, fully-incremental, and one-to-one synchronizer using update translation and homogeneous update comparison and reconciliation with choice:*
$S11_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \Delta\mathcal{T} \times \mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}} \to \mathcal{S} \times \mathcal{T}$

$$
\begin{array}{l}
S_{\mathcal{S}}, S'_{\mathcal{S}}, U_{\Delta\mathcal{S}}, \\
T_{\mathcal{T}}, T'_{\mathcal{T}}, Y_{\Delta\mathcal{T}}, \\
F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}} \\
U_{\Delta\mathcal{S}}(S_{\mathcal{S}}) = S'_{\mathcal{S}} \\
Y_{\Delta\mathcal{T}}(T_{\mathcal{T}}) = T'_{\mathcal{T}} \\
(S_{\mathcal{S}}, T_{\mathcal{T}}) \in R \implies
\begin{array}{l}
U_{\Delta\mathcal{T}} = UT_{\mathcal{S},\mathcal{T}}(U_{\Delta\mathcal{S}}, S_{\mathcal{S}}, T_{\mathcal{T}}) \\
(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}(UCR^*_{\mathcal{T}}(U_{\Delta\mathcal{T}}, Y_{\Delta\mathcal{T}}, T_{\mathcal{T}})) \\
T''_{\mathcal{T}} = Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}}) \\
U'_{\Delta\mathcal{S}} = UT_{\mathcal{T},\mathcal{S}}(U'_{\Delta\mathcal{T}}, T'_{\mathcal{T}}, S'_{\mathcal{S}}) \\
S''_{\mathcal{S}} = U'_{\Delta\mathcal{S}}(S'_{\mathcal{S}}) \qquad \implies \quad S''_{\mathcal{S}}, T''_{\mathcal{T}}
\end{array}
\end{array}
$$

Analogously to the non-incremental variant, the $\text{UCR}^*_{\mathcal{T}}$ operator performs the three-way comparison of the updates with respect to the reference artifact $T_{\mathcal{T}}$. Again, the result is a pair of reconciled updates. The update $U'_{\Delta\mathcal{T}}$ needs to be translated into the artifact type $\mathcal{S}$. Finally, the reconciled updates are applied.

6.3 Bidirectional synchronizers for many-to-one relations

For many-to-one relations, we only consider homogeneous reconciliation on the target side since source artifacts or updates can be unambiguously translated in the target direction. We show two synchronizers in this category. The first synchronizer uses a non-incremental unidirectional synchronizer in the source-to-target direction, while the other uses an incremental one. For the target-to-source direction, we need to use one of the unidirectional to-many synchronizers that are original-target-dependent, where the "original target" corresponds to the new source in our context. The reason is that we want to preserve non-reflectable edits from the new source. Both synchronizers use update translation with choice in the target-to-source direction.

Synchronizer 12 *Bidirectional, fully-incremental, and many-to-one synchronizer using artifact translation, homogeneous artifact comparison and reconciliation with choice, and update translation with choice:*
$S12_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{S}} \times \mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}} \to \mathcal{S} \times \mathcal{T}$

$$
\begin{array}{l}
S_{\mathcal{S}}, S'_{\mathcal{S}}, T_{\mathcal{T}}, T'_{\mathcal{T}}, \\
D_{\mathcal{D}_{\Delta\mathcal{S}}}, \\
F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}} \\
(S_{\mathcal{S}}, T_{\mathcal{T}}) \in R \implies S'_{\mathcal{T}} = AT_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}}) \\
\qquad\qquad (U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}(ACR^*_{\mathcal{T}}(S'_{\mathcal{T}}, T'_{\mathcal{T}}, T_{\mathcal{T}})) \\
\qquad\qquad T''_{\mathcal{T}} = Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}}) \\
\qquad\qquad U'_{\Delta\mathcal{S}} = D_{\mathcal{D}_{\Delta\mathcal{S}}}(UT^*_{\mathcal{T},\mathcal{S}}(U'_{\Delta\mathcal{T}}, S'_{\mathcal{T}}, S'_{\mathcal{S}})) \\
\qquad\qquad S''_{\mathcal{S}} = U'_{\Delta\mathcal{S}}(S'_{\mathcal{S}}) \qquad\qquad \implies S''_{\mathcal{S}}, T''_{\mathcal{T}}
\end{array}
$$

Synchronizer 13 *Bidirectional, fully-incremental, and many-to-one synchronizer using update translation, homogeneous update comparison and reconciliation with choice, and update translation with choice:*
$S13_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \Delta\mathcal{T} \times \mathcal{D}_{\Delta\mathcal{S}} \times \mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}} \to \mathcal{S} \times \mathcal{T}$

$$
\begin{array}{l}
S_{\mathcal{S}}, S'_{\mathcal{S}}, U_{\Delta\mathcal{S}}, \\
T_{\mathcal{T}}, T'_{\mathcal{T}}, Y_{\Delta\mathcal{T}}, \\
D_{\mathcal{D}_{\Delta\mathcal{S}}}, \\
F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}} \\
U_{\Delta\mathcal{S}}(S_{\mathcal{S}}) = S'_{\mathcal{S}} \\
Y_{\Delta\mathcal{T}}(T_{\mathcal{T}}) = T'_{\mathcal{T}} \\
(S_{\mathcal{S}}, T_{\mathcal{T}}) \in R \implies U_{\Delta\mathcal{T}} = UT_{\mathcal{S},\mathcal{T}}(U_{\Delta\mathcal{S}}, S_{\mathcal{S}}, T_{\mathcal{T}}) \\
\qquad\qquad (U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}(UCR^*_{\mathcal{T}}(U_{\Delta\mathcal{T}}, Y_{\Delta\mathcal{T}}, T_{\mathcal{T}})) \\
\qquad\qquad T''_{\mathcal{T}} = Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}}) \\
\qquad\qquad U'_{\Delta\mathcal{S}} = D_{\mathcal{D}_{\Delta\mathcal{S}}}(UT^*_{\mathcal{T},\mathcal{S}}(U'_{\Delta\mathcal{T}}, T'_{\mathcal{T}}, S'_{\mathcal{S}})) \\
\qquad\qquad S''_{\mathcal{S}} = U'_{\Delta\mathcal{S}}(S'_{\mathcal{S}}) \qquad\qquad \implies S''_{\mathcal{S}}, T''_{\mathcal{T}}
\end{array}
$$

An example for Synchronizer 12.

Example 19. Synchronization in FSMLs.

The FSML infrastructure [10] supports synchronization according to Synchronizer 12. Source artifact is Java code, XML code, or a combination of both. Target artifact is a model in an FSML designed for a particular framework, e.g., Apache Struts (cf. Example 3). The relation between source and target is many-to-one. The infrastructure performs homogeneous artifact comparison and reconciliation on the model (target) side since every code update has a unique representation on the model side. The reverse is not true: a model update can be translated in different ways into code updates.

The first step of the synchronizer is to retrieve $S'_{\mathcal{T}}$, i.e., the *model of the new code*, from the new code $S'_{\mathcal{S}}$ using $\mathrm{AT}_{\mathcal{S},\mathcal{T}}$, which is implemented by a set of code queries (cf. Example 8).

The second step is a three-way compare between the model of the new code $S'_{\mathcal{T}}$ and the new model $T'_{\mathcal{T}}$ while using the original model $T_{\mathcal{T}}$ as a reference

artifact. The original model corresponds to the initial situation when the model and the code were consistent after the previous synchronization, and the new model and the new code are the results of independent updates of the respective original artifacts (cf. Figure 9).

The artifact comparison and reconciliation $\text{ACR}^*_{\mathcal{T}}$ operates on framework-specific models. A model is an object structure conforming to a class model, i.e., the metamodel. The object structure consists of objects (i.e., *model elements*), attributes with primitive values, and containment and reference links between objects. The containment links form a containment hierarchy, which is a tree. The comparison process starts with establishing the correspondence among the model elements in all three models, namely $S'_{\mathcal{T}}$, $T'_{\mathcal{T}}$, and $T_{\mathcal{T}}$. The correspondence is established using structural matching, which takes into account the location of the elements in the containment hierarchy and their identification keys that are specified in the metamodel. Approaches to establishing correspondence are further discussed in Section 8.2. The result of the matching is a set of 3-tuples, where each tuple contains the corresponding elements from the three input models. Each position in a 3-tuple is dedicated to one of the three input models and contains the corresponding element from the model or a special symbol representing the absence of the corresponding element from that model.

Table 2. Results of three-way compare of the corresponding elements t, s, and r in the new artifacts $T'_{\mathcal{T}}$ and $S'_{\mathcal{T}}$, and the reference artifact $T_{\mathcal{T}}$, respectively. The absence of a corresponding element is represented by -. Table adapted from [10].

$T'_{\mathcal{T}}$	$S'_{\mathcal{T}}$	$T_{\mathcal{T}}$	**condition**	**detected updates to element**	**update classification**
s	t	r	$t = s = r$	unchanged	no updates
s	t	r	$t = s \wedge t \neq r$	modified consistently in $T'_{\mathcal{T}}$ & $S'_{\mathcal{T}}$	consistent updates
s	t	-	$t = s$	added consistently to $T'_{\mathcal{T}}$ & $S'_{\mathcal{T}}$	consistent updates
s	t	r	$t \neq s \wedge t = r$	modified in $S'_{\mathcal{T}}$	propagating update in $S'_{\mathcal{T}}$
s	t	r	$t \neq s \wedge s = r$	modified in $T'_{\mathcal{T}}$	propagating update in $T'_{\mathcal{T}}$
s	t	r	$t \neq s \neq r \neq t$	modified inconsistently in $T'_{\mathcal{T}}$ & $S'_{\mathcal{T}}$	conflicting updates
s	t	-	$t \neq s$	added inconsistently to $T'_{\mathcal{T}}$ & $S'_{\mathcal{T}}$	conflicting updates
s	-	r	$t = r$	removed from $S'_{\mathcal{T}}$	propagating update in $S'_{\mathcal{T}}$
s	-	r	$t \neq r$	removed from $S'_{\mathcal{T}}$, modified in $T'_{\mathcal{T}}$	conflicting updates
s	-	-	-	added to $T'_{\mathcal{T}}$	propagating update in $T'_{\mathcal{T}}$
-	t	r	$s = r$	removed from $T'_{\mathcal{T}}$	propagating update in $T'_{\mathcal{T}}$
-	t	r	$s \neq r$	removed from $T'_{\mathcal{T}}$, modified in $S'_{\mathcal{T}}$	conflicting updates
-	t	-	-	added to $S'_{\mathcal{T}}$	propagating update in $S'_{\mathcal{T}}$
-	-	r	-	removed from $T'_{\mathcal{T}}$ & $S'_{\mathcal{T}}$	consistent updates

The comparison process continues by processing each 3-tuple to establish the updates that occurred in the new source and the new target according to Table 2. The first and the second column classifies each 3-tuple according to whether all three elements or only some were present and whether the corresponding elements were equal or not. Two elements are equal iff their corresponding attribute values are equal, their corresponding reference links point to the same element,

and the corresponding contained elements are equal. The third column describes the detected updates as element additions, modifications, and removals, and the fourth column classifies the updates as propagating, consistent, or conflicting (cf. Section 6.1).

The classification results are then presented to the user, who can review each of the updates and decide to accept or reject it. More precisely, a propagating update can be accepted or rejected and a pair of conflicting updates can be enforced in the forward or the reverse direction or rejected all together. Note that the decisions can be taken at different levels in the containment hierarchy. In a extreme, the user might only review the updates at the level of the corresponding model elements representing the model roots. The user might also desire to drill down the hierarchy and review the updates at a finer granularity.

The conflict resolution decisions made by the user correspond to the decision function $\mathrm{F}_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}$. It is desirable that the decisions taken by the user should result in a well-formed model $T''_{\mathcal{T}}$ before the code is updated. However, in practice, developers may choose to synchronize one element at a time. Also, only accepting and/or rejecting updates may not be enough to arrive at the desired model, meaning that developers might need to perform some additional edits during reconciliation.

The last stage of $\mathrm{ACR}^{*}_{\mathcal{T}}$ is to compute the resulting updates $U''_{\Delta\mathcal{T}}$ and $Y''_{\Delta\mathcal{T}}$. The resulting update $U''_{\Delta\mathcal{T}}$ for $S'_{\mathcal{T}}$ is computed by collecting the synchronizing update for every accepted propagating and conflicting update to $T'_{\mathcal{T}}$ and the inverse updates to the rejected propagating updates. An inverse update reverts an element back to its state from $T_{\mathcal{T}}$. There is no need to include an inverse update for the rejected update from a conflicting pair since the accepted update will override the corresponding element. The update $Y''_{\Delta\mathcal{T}}$ is computed in a similar way.

Finally, the update of the model representing the new code, $U''_{\Delta\mathcal{T}}$, is translated into the update of the new code, $U''_{\Delta\mathcal{S}}$. The translation is achieved using update translation $\mathrm{UT}^{*}_{\mathcal{T},\mathcal{S}}$ as described in Example 15. At last, both the new code and the new model are incrementally updated by applying $U''_{\Delta\mathcal{S}}$ and $Y''_{\Delta\mathcal{T}}$, respectively, and the synchronizer returns the two reconciled artifacts $S''_{\mathcal{S}}$ and $T''_{\mathcal{T}}$.

6.4 Bidirectional synchronizers for many-to-many relations

Reconciliation for many-to-many relations can be performed in the homogeneous or heterogeneous fashion. A bidirectional synchronizer with homogeneous reconciliation for a many-to-many relation needs to use unidirectional original-target-dependent to-many synchronizers in both directions.

First we show a bidirectional synchronizer with homogeneous reconciliation that uses update translation with choice in both directions.

Synchronizer 14 *Bidirectional, fully-incremental, and many-to-many synchronizer using update translation with choice and homogeneous update comparison*

and reconciliation with choice:
$S14_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \Delta\mathcal{T} \times \mathcal{D}_{\Delta\mathcal{S}} \times \mathcal{D}_{\Delta\mathcal{T}} \times \mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}} \to \mathcal{S} \times \mathcal{T}$

$S_{\mathcal{S}}, S'_{\mathcal{S}}, U_{\Delta\mathcal{S}},$
$T_{\mathcal{T}}, T'_{\mathcal{T}}, Y_{\Delta\mathcal{T}},$
$D_{\mathcal{D}_{\Delta\mathcal{S}}}, E_{\mathcal{D}_{\Delta\mathcal{T}}},$
$F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}$
$U_{\Delta\mathcal{S}}(S_{\mathcal{S}}) = S'_{\mathcal{S}}$
$Y_{\Delta\mathcal{T}}(T_{\mathcal{T}}) = T'_{\mathcal{T}}$
$(S_{\mathcal{S}}, T_{\mathcal{T}}) \in R \implies U_{\Delta\mathcal{T}} = E_{\mathcal{D}_{\Delta\mathcal{T}}}(UT^*_{\mathcal{S},\mathcal{T}}(U_{\Delta\mathcal{S}}, S_{\mathcal{S}}, T_{\mathcal{T}}))$
$(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}(UCR^*_{\mathcal{T}}(U_{\Delta\mathcal{T}}, Y_{\Delta\mathcal{T}}, T_{\mathcal{T}}))$
$T''_{\mathcal{T}} = Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}})$
$U'_{\Delta\mathcal{S}} = D_{\mathcal{D}_{\Delta\mathcal{S}}}(UT^*_{\mathcal{T},\mathcal{S}}(U'_{\Delta\mathcal{T}}, S_{\mathcal{S}}, T_{\mathcal{T}}))$
$S''_{\mathcal{S}} = U'_{\Delta\mathcal{S}}(S'_{\mathcal{S}}) \implies S''_{\mathcal{S}}, T''_{\mathcal{T}}$

The heterogeneous variant of the many-to-many synchronizer requires a heterogeneous comparison and reconciliation operator.

Operator 11 *Heterogeneous artifact comparison and reconciliation with choice:* $ACR^*_{\mathcal{S},\mathcal{T}} : \mathcal{S}\times\mathcal{T}\times\mathcal{S}\times\mathcal{T} \to \mathcal{P}^+(\Delta\mathcal{S}\times\Delta\mathcal{T})$*. For two artifacts* $S'_{\mathcal{S}}$ *and* $T'_{\mathcal{T}}$*, and two consistent reference artifacts* $S_{\mathcal{S}}$ *and* $T_{\mathcal{T}}$*, the operator* $ACR^*_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}}, T'_{\mathcal{T}}, S_{\mathcal{S}}, T_{\mathcal{T}})$ *computes a non-empty subset of* $\{(U'_{\Delta\mathcal{S}}, Y'_{\Delta\mathcal{T}}) : (U'_{\Delta\mathcal{S}}(S'_{\mathcal{S}}), Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}})) \in R\}$*. Each pair of updates* $(U'_{\Delta\mathcal{T}}, Y'_{\Delta\mathcal{T}})$ *from that subset is such that the updates resolve conflicting updates and enforce all propagating updates from* $U_{\Delta\mathcal{S}}$ *and* $Y_{\Delta\mathcal{T}}$*, where* $U_{\Delta\mathcal{S}} = AC_{\mathcal{S}}(S_{\mathcal{S}}, S'_{\mathcal{S}})$ *and* $Y_{\Delta\mathcal{T}} = AC_{\mathcal{T}}(T_{\mathcal{T}}, T'_{\mathcal{T}})$*.*

Synchronizer 15 *Bidirectional, fully-incremental, and many-to-many synchronizer using heterogeneous artifact comparison and reconciliation with choice:*
$S15_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{S} \times \mathcal{T} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{S}\times\Delta\mathcal{T}} \to \mathcal{S} \times \mathcal{T}$

$S_{\mathcal{S}}, S'_{\mathcal{S}}, T_{\mathcal{T}}, T'_{\mathcal{T}},$
$F_{\mathcal{D}_{\Delta\mathcal{S}\times\Delta\mathcal{T}}}$
$(S_{\mathcal{S}}, T_{\mathcal{T}}) \in R \implies (U'_{\Delta\mathcal{S}}, Y'_{\Delta\mathcal{T}}) = F_{\mathcal{D}_{\Delta\mathcal{S}\times\Delta\mathcal{T}}}(ACR^*_{\mathcal{S},\mathcal{T}}(S'_{\mathcal{S}}, T'_{\mathcal{T}}, S_{\mathcal{S}}, T_{\mathcal{T}}))$
$S''_{\mathcal{S}} = U'_{\Delta\mathcal{S}}(S'_{\mathcal{S}})$
$T''_{\mathcal{T}} = Y'_{\Delta\mathcal{T}}(T'_{\mathcal{T}}) \implies S''_{\mathcal{S}}, T''_{\mathcal{T}}$

We introduce the last variant, Synchronizer 16, by first discussing its sample implementation.

An example for Synchronizer 16.

Example 20. ATL Virtual Machine extension for synchronization.

An example of a bidirectional many-to-many synchronizer is an extension to the Atlas Transformation Language (ATL) [28] virtual machine [29]. While the synchronizer works in the reconciliation setting as illustrated in Figure 9 and allows independent updates to the original source and the original target, it only supports partial reconciliation. More specifically, while the synchronizer

propagates all propagating updates, it does not support conflict resolution. Furthermore, the synchronizer does not tolerate additions made to the original target model. In any of the above situations, the synchronizer reports an error and terminates.

The mapping between source and target is given as an artifact translator expressed in ATL, which is a unidirectional transformation language. While an ATL translator is a partial function, the extension supports many-to-many relations by merging the translation results with existing artifacts using asymmetric homogeneous merge (cf. Operator 6). In this way, the non-reflectable updates from the new source and the new target can be preserved.

The following synchronizer describes the synchronization procedure.

Synchronizer 16 *Bidirectional, source-incremental, and many-to-many synchronizer using artifact translation, homogeneous artifact comparison, update translation with choice, and homogeneous asymmetric artifact merge with choice:* $S16_{\mathcal{S},\mathcal{T}} : \mathcal{S} \times \mathcal{S} \times \mathcal{T} \times \mathcal{D}_{\Delta\mathcal{S}} \times \mathcal{D}_{\Delta\mathcal{S}} \times \mathcal{D}_{\Delta\mathcal{T}} \to \mathcal{S} \times \mathcal{T}$

$$
\begin{array}{lll}
S_{\mathcal{S}}, S'_{\mathcal{S}}, T'_{\mathcal{T}}, & & \\
D_{\mathcal{D}_{\mathcal{S}}}, E_{\mathcal{D}_{\mathcal{S}}}, F_{\mathcal{D}_{\mathcal{T}}} & \Longrightarrow & T_{\mathcal{T}} = AT_{\mathcal{S},\mathcal{T}}(S_{\mathcal{S}}) \\
 & & Y_{\Delta\mathcal{T}} = AC_{\mathcal{T}}(T_{\mathcal{T}}, T'_{\mathcal{T}}) \\
 & & Y_{\Delta\mathcal{S}} = D_{\mathcal{D}_{\mathcal{S}}}(UT^{*}_{\mathcal{T},\mathcal{S}}(Y_{\Delta\mathcal{T}}, T_{\mathcal{T}}, S_{\mathcal{S}})) \\
 & & S^{Y}_{\mathcal{S}} = Y_{\Delta\mathcal{S}}(S_{\mathcal{S}}) \\
 & & S''_{\mathcal{S}} = E_{\mathcal{D}_{\mathcal{S}}}(M^{*}_{\mathcal{S}}(S'_{\mathcal{S}}, S^{Y}_{\mathcal{S}}, \phi^{1}_{\Phi_{\mathcal{S}}})) \\
 & & S''_{\mathcal{T}} = AT_{\mathcal{S},\mathcal{T}}(S''_{\mathcal{S}}) \\
 & & T''_{\mathcal{T}} = F_{\mathcal{D}_{\mathcal{T}}}(M^{*}_{\mathcal{T}}(T'_{\mathcal{T}}, S''_{\mathcal{T}}, \phi^{2}_{\Phi_{\mathcal{T}}})) \qquad \Longrightarrow \quad S''_{\mathcal{S}}, T''_{\mathcal{T}}
\end{array}
$$

$$
\textit{where} \quad \phi^{1}_{\Phi_{\mathcal{S}}}(S) = \begin{cases} 1 & \textit{if } (T'_{\mathcal{T}}, S) \in R \\ 0 & \textit{otherwise} \end{cases}
$$

$$
\phi^{2}_{\Phi_{\mathcal{T}}}(T) = \begin{cases} 1 & \textit{if } (S''_{\mathcal{S}}, T) \in R \\ 0 & \textit{otherwise} \end{cases}
$$

First, the original target $T_{\mathcal{T}}$ is obtained by executing an artifact translation written in ATL. Next, the update of the original target $Y_{\Delta\mathcal{T}}$ is translated into the corresponding update of the original source $S_{\mathcal{S}}$ using the virtual machine extension. The information that is necessary for the update translation in the reverse direction was recorded by the ATL virtual machine extension during the execution of the artifact translation in the forward direction. Next, the new source $S'_{\mathcal{S}}$ is merged with $S^{Y}_{\mathcal{S}}$, which is the updated original source incorporating the source translation of $Y_{\Delta\mathcal{T}}$. The merged artifact $S''_{\mathcal{S}}$ is the reconciled source artifact. Finally, the reconciled source $S''_{\mathcal{S}}$ is translated into the artifact $S''_{\mathcal{T}}$, which is then merged with the new target $T'_{\mathcal{T}}$ to produce the reconciled target artifact $T''_{\mathcal{T}}$.

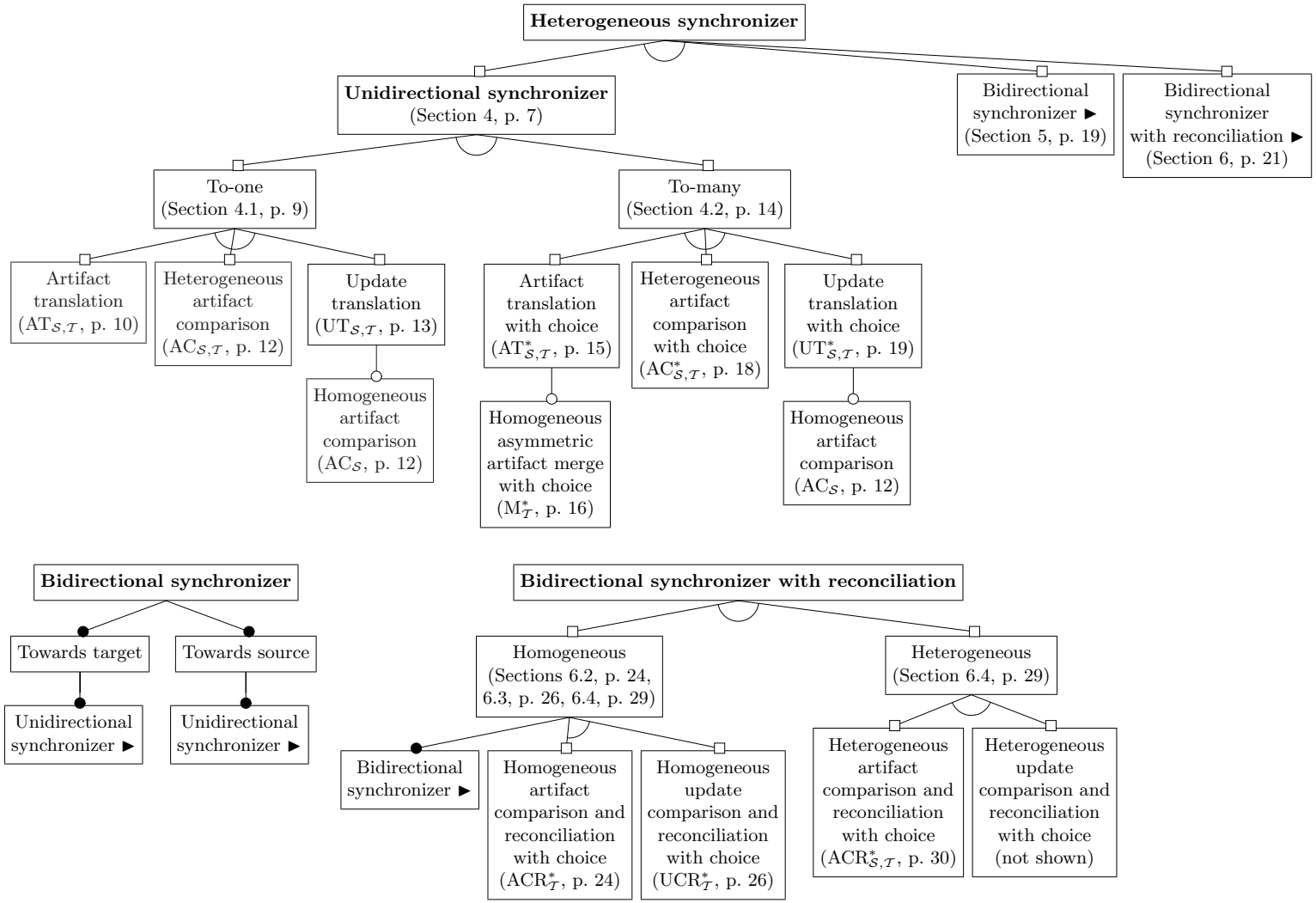

Fig. 11. Design space of heterogeneous synchronizers

7 Summary of Synchronizers and Tradeoffs

In this section, we summarize the presented synchronizers and discuss the tradeoffs among them. Figure 11 presents a composite feature model of the design space of heterogeneous synchronizers. The feature model serves two purposes: 1) it consolidates the fragments of the feature model spread over the course of the tutorial, and 2) it provides section and page numbers of the feature descriptions. The leaf features that are not references, i.e., the leafs without ►, correspond to artifact operators.

Table 7 shows a feature comparison of the presented synchronizers and their inputs. Synchronizers 1–8 are unidirectional, of which Synchronizers 1–4 are to-one and Synchronizers 5–8 are to-many. Synchronizers 9–16 are bidirectional. Among them, Synchronizers 9–11 are one-to-one, Synchronizers 12–13 are many-to-one, and Synchronizers 14–16 are many-to-many.

Tradeoffs for unidirectional to-one synchronizers. The incremental variants offer higher performance than the non-incremental one because only individual updates are considered instead of the whole artifacts. Consequently, they enable more frequent synchronization for large artifacts. However, implementing heterogeneous artifact comparison or update translation operators is usually more complex than implementing artifact translation. The reason is that additional design decisions for implementing updates (cf. Section 8.1) and matching (cf. Section 8.2) need to be considered.

Furthermore, while the incremental variant based on update translation is likely to be more efficient than the one based on heterogeneous artifact comparison, the additional requirement that the original versions of the artifacts need to be consistent may be too restrictive in some situations. For example, it could be sufficient for the original versions to be nearly consistent.

Tradeoffs for unidirectional to-many synchronizers. The first synchronizer, i.e., the one based on artifact translation with choice and without homogeneous merge, is only useful if the target is not going to be manually edited in between target regenerations, as in the case of compiling a program into object code. If the target is intended to be edited, any of the remaining to-many variants needs to be used.

The synchronizer using the merge operator is simple to implement for cases where the structures of the target artifact that are non-reflectable in the source are well separated from the structures that are reflectable in the source. Such separation simplifies the implementation of the merge. If both kinds of structures are strongly intertwined, one of the incremental to-many synchronizers may be a better choice since they take the original target into account already in the translation operator.

The tradeoffs between the two incremental synchronizers, i.e., the one using heterogeneous artifact comparison with choice and the other one using update

Table 3. Summary of features synchronizers on their inputs

		$\rightarrow 1$				$\rightarrow *$				$1 \leftrightarrow 1$			$* \leftrightarrow 1$		$* \leftrightarrow *$		
	Synchronizer	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
increment.	non-incremental	●	·	·	·	●	●	·	·	·	·	·	·	·	·	·	·
	(target-) incremental	·	●	●	●	·	·	●	●	●	·	·	·	·	·	·	·
	source-incremental	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	●
	fully-incremental	·	·	·	·	·	·	·	·	·	●	●	●	●	●	●	·
	original-target-dependent	·	·	·	·	·	●	●	●	●	●	●	●	●	●	●	●
inputs	$S_{\mathcal{S}}$ (original source artifact)	·	·	●	●	·	·	·	●	·	·	●	●	●	●	●	●
	$S'_{\mathcal{S}}$ (new source artifact)	●	●	·	●	●	●	●	·	●	●	●	●	●	●	●	●
	$U_{\Delta\mathcal{S}}$ (update of the orig. source art.)	·	·	●	·	·	·	·	●	·	·	●	·	●	●	·	·
	$\mathrm{D}_{\mathcal{D}_{\mathcal{S}}}$ (decision function on artifact)	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	●
	$\mathrm{D}_{\mathcal{D}_{\Delta\mathcal{S}}}$ (decision function on update)	·	·	·	·	·	·	·	·	·	·	·	●	●	●	·	·
	$\mathrm{E}_{\mathcal{D}_{\mathcal{S}}}$ (decision function on artifact)	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	●
	$T_{\mathcal{T}}$ (original target artifact)	·	●	●	●	·	●	●	●	●	●	●	●	●	●	●	·
	$T'_{\mathcal{T}}$ (new target artifact)	·	·	·	·	·	·	·	·	●	●	●	●	●	●	●	●
	$Y_{\Delta\mathcal{T}}$ (update of the orig. target art.)	·	·	·	·	·	·	·	·	·	·	●	·	●	●	·	·
	$\mathrm{D}_{\mathcal{D}_{\mathcal{T}}}$ (decision function on artifact)	·	·	·	·	●	●	●	·	·	·	·	·	·	·	·	·
	$\mathrm{D}_{\mathcal{D}_{\Delta\mathcal{T}}}$ (decision function on update)	·	·	·	·	·	·	·	●	·	·	·	·	·	·	·	·
	$\mathrm{E}_{\mathcal{D}_{\mathcal{T}}}$ (decision function on artifact)	·	·	·	·	·	●	·	·	·	·	·	·	·	●	·	·
	$\mathrm{F}_{\mathcal{D}_{\Delta\mathcal{T}\times\Delta\mathcal{T}}}$ (decision fun. on updates)	·	·	·	·	·	·	·	·	●	●	●	●	●	●	·	·
	$\mathrm{F}_{\mathcal{D}_{\Delta\mathcal{S}\times\Delta\mathcal{T}}}$ (decision fun. on updates)	·	·	·	·	·	·	·	·	·	·	·	·	·	·	●	·
	$\mathrm{F}_{\mathcal{D}_{\mathcal{T}}}$ (decision function on artifact)	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	●
precon.	$(S_{\mathcal{S}}, T_{\mathcal{T}}) \in R$	·	·	●	●	·	·	·	●	·	·	●	●	●	●	●	·
	$U_{\Delta\mathcal{S}}(S_{\mathcal{S}}) = S'_{\mathcal{S}}$	·	·	·	·	·	·	·	·	·	·	·	·	●	●	·	·
	$Y_{\Delta\mathcal{T}}(T_{\mathcal{T}}) = T'_{\mathcal{T}}$	·	·	·	·	·	·	·	·	·	·	·	·	●	●	·	·
operations	artifact translation	●	·	·	·	·	·	·	·	●	●	·	●	·	·	·	●
	heterogeneous artifact comparison	·	●	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	update translation	·	·	●	●	·	·	·	·	·	●	●	·	●	·	·	·
	homogeneous artifact comparison	·	·	·	●	·	·	·	·	·	·	·	·	·	·	·	●
	artifact translation*	·	·	·	·	●	●	·	·	·	·	·	·	·	·	·	·
	update translation*	·	·	·	·	·	·	·	●	·	·	·	●	●	●	·	●
	homog. asymmetric artifact merge*	·	·	·	·	·	●	·	·	·	·	·	·	·	·	·	●
	heterogeneous artifact comparison*	·	·	·	·	·	·	●	·	·	·	·	·	·	·	·	·
	homog. artifact comp. & recon.*	·	·	·	·	·	·	·	·	●	●	·	●	·	·	·	·
	homog. update comp. & recon.*	·	·	·	·	·	·	·	·	·	·	●	·	●	●	·	·
	heterog. artifact comp. & recon.*	·	·	·	·	·	·	·	·	·	·	·	·	·	·	●	·

translation with choice, are similar to the tradeoffs between their to-one counterparts.

Tradeoffs for bidirectional synchronizers. The choice of the unidirectional synchronizer mainly depends on the cardinalities of the relation's ends, i.e., one or many. Each of the synchronizers can be non-incremental or incremental depending on the performance requirements. If the cardinality of the end towards which the synchronizer should be executed is many, any of the unidirectional

to-many synchronizers that are original-target-dependent should be used. The synchronizer should be original-target-dependent because the target of the synchronizer can be edited.

Tradeoffs for bidirectional synchronizers with reconciliation. Homogeneous reconciliation is most appropriate if the relation between the source and target has at least one end with the cardinality of one because artifacts or updates can be unambiguously translated in the direction of that end. In contrast, heterogeneous reconciliation appears to be more appropriate for many-to-many relations.

8 Additional design decisions

In this Section, we present additional design decisions related to the implementation of the synchronizers: creation and representation of updates, structure identification and matching, modes of synchronization, implementation of decision functions, and construction and correctness of synchronizers.

8.1 Creation and representation of updates

An update describes the change of artifact's internal structure. The application of an update corresponds to the execution of a sequence of artifact update operations such as element additions, removals, and relocations and attribute value changes. One way of obtaining a sequence of artifact update operations is by recording editing operations performed by the artifact's developer. In this case, the artifact at the beginning of the recording is a reference artifact of the recorded update. We refer to updates obtained by recording developer's edits as *history-based updates*. The sequence of developer's edits can be transformed into a *canonical form*, which produces the same result as the original sequence, but lacks redundant edits, such as, modifying the same attribute multiple times. Another way of creating an update is by comparing two artifacts using *homogeneous artifact comparison*. We refer to updates obtained by comparing two artifacts as *state-based updates*. History-based updates contain more information than state-based updates, but may be more difficult to implement in practice. For example, implementations have to ensure that all artifact updates are performed through an appropriate change-tracking interface.

8.2 Structure identification and matching

Comparison operators such as homogeneous and heterogeneous artifact comparison require a way to establish the correspondence between the elements of the artifacts being compared. Furthermore, the implementation of an update function must also contain information about the elements it affects and a way to identify them in the reference artifact. We refer to the process of establishing the correspondence between elements as *matching*.

The two fundamentally different approaches to matching are *non-structural matching* and *structural matching*. Non-structural matching assumes that elements receive globally unique, structure-independent identifiers at the time of their creation. By "globally unique" we mean that the identifiers are unique at least in the scope of the matched artifacts. Structure-independent means that the identifiers are independent of the artifact structure, meaning that they remain constant when the structure evolves. For example, the identifier could be generated as a combination of the IP address of the machine where the identifier is generated, a timestamp, and a random number. This approach greatly simplifies matching among different versions of an artifact as the correspondence of elements can be established immediately based on the equality of the identifiers. The main drawback of this approach is that it tends to be brittle with respect to artifact evolution that involves a deletion and subsequent recreation of an element. For example, consider the removal of a method from a class and its later re-introduction. The new method would have a new identifier, which would mark it as a new element even though it is probably just a new version of the original method. Furthermore, identifiers tend to pollute and bloat the artifacts, especially if they need to be stored in a human-readable textual form.

Structural matching avoids both problems by establishing correspondence through the structural information that is already in the artifacts, e.g., element nesting, element's position in ordered lists, and attribute values such as element's local name. In our method evolution example, the old and the new version of the method could be matched by using the fully qualified name of the containing class and the method's signature as an identifier. The matching can still use precomputed identifiers, but these identifiers would be structure dependent as they encode structural information. The main drawback of structural matching is that sometimes the structural information needed for recovering a particular relationship may be missing or difficult to identify. For example, while the fully qualified name and signature of a method is sufficient to unambiguously match a single call to that method within the body of another method, identifying multiple calls within a single body is challenging. Using the lexical order of the calls is a possible solution, but one that is brittle with respect to evolution when the body is restructured, for example, when additional calls are inserted in the middle of the body. A practical solution may need to use more local context information of each call in order to establish the correspondence between the two versions. The problem of recognizing element relocations in nested structures is an active research topic, e.g., [30].

In practice, both non-structural and structural matching can be used in combination. For example, the model management infrastructure of IBM Rational Software Modeler [31], which is IBM's UML modeling tool, supports both non-structural and structural matching.

Finally, matching could be realized at a *semantic level* rather than a structural (i.e., syntactic) one. For example, Nejati et al. [32] present an approach for matching and merging Statecharts specifications that respects the behavioral semantics of the specifications.

8.3 Instantaneous vs. on-demand synchronization

Another design decision is the time of update propagation. We distinguish between *instantaneous* and *on-demand* synchronization. Instantaneous synchronization translates and applies updates to the target artifact immediately after the updates occurred in the source artifact. On-demand synchronization translates and applies updates at the time most convenient for the developer. Instantaneous synchronization is likely to require an incremental synchronizer since translating the entire source artifact after each update would be highly inefficient.

8.4 Disconnected vs. live synchronization

Update propagation can be implemented as a *disconnected* or a *live* transformation. Live transformation is a transformation that does not terminate [18,25] and whose intermediate execution data, referred to as *execution context*, is preserved. The context of a live transformation maintains the links between structures in the source artifact and the resulting structures in the target artifact. The preservation of the execution context allows for efficient propagation of updates made to the source artifact (cf. Example 10). In contrast, a disconnected transformation terminates and its execution trace is lost, in which case a structure matching mechanism is needed (cf. Section 8.2).

8.5 Strategies for selecting synchronization result from multiple choices

Synchronization in the "to-many" direction requires a way to select a single target from the set of possible targets that are consistent with the source. We distinguish among the following selection strategies:

- *Pre-determined choice*: The choice is fixed by the synchronizer developer and hardcoded in the synchronizer.
- *Interactive selection*: The available choices, typically ranked according to some criteria, are presented to the user interactively. While the number of choices may be infinite, a finite number is presented at a time and the user can ask for more.
- *User-specified defaults*: The user may use global options to specify preference. Alternatively, the choices may be related to individual source elements, in which case the source elements are annotated. Examples of annotation mechanisms are Java annotations and UML profiles.
- *Adaptive defaults*: The default settings could be obtained by mining from the original target or from a corpus of existing sample targets. An example of this strategy is the automatic application of code formatting that was extracted from a corpus of sample programs using data mining techniques [33].
- *Target preservation*: The available choices may be restricted by the desire to preserve structures in the original target. We accounted for this possibility in the original-target-dependent synchronizers.

8.6 Construction of bidirectional synchronizers

Bidirectional synchronizers can be constructed using either a bidirectional or a unidirectional transformation language. Synchronizers constructed using a bidirectional language can be directly executed in both directions from a single specification.

Examples of bidirectional transformation languages include QVT Relations [34], *triple graph grammars* [25, 35] (TGGs), and Lenses for trees [17]. In QVT and TGGs, synchronizers are expressed by a set of rules, which can be executed in both forward and reverse directions. Implementations of the QVT Relations language include ModelMorf by TATA Research Development and Design Centre and Medini QVT by IKV++. Tool support for creating TGG-based synchronizers exists as a plug-in for the FUJABA tool suite [26]. In the Lenses approach complex bidirectional synchronizers are implemented by composing bidirectional primitives using *combinators*. Similarly to lenses, Xiong et al. propose an approach to building bidirectional synchronizers using combinators that translate modification operations performed on one artifact to synchronizing operations on the other artifact [36]. In this approach, a synchronizer is defined by creating a *synchronizer graph*, which consists of primitive synchronizers, input artifacts, and intermediate (temporary) artifacts. The approach additionally supports different synchronization behaviors by parameterizing primitive synchronizers with mode options.

Using a unidirectional transformation language requires either writing two unidirectional synchronizers, one in each direction, or writing a unidirectional synchronizer in one direction and automatically computing its inverse. Depending on the type of relation among the artifacts, the two unidirectional transformations can be constructed in many ways. For some bijections, an inverse transformation can be automatically computed from the transformation in one direction. Pierce provides a list of examples of interesting cases of computing such inverse transformations [37]. Xiong et al. [29] developed an approach that can execute a synchronizer written in ATL, a unidirectional language, in the reverse direction (cf. Example 20). The information that is necessary for the reverse transformation is recorded by an extension to the ATL virtual machine during the execution of the synchronizer in the forward direction.

8.7 Correctness of synchronizers

In practice, establishing full consistency automatically may not always be possible. First, developers may desire to synchronize partially finished artifacts, i.e., the synchronizer may need to be able to handle artifacts of which only parts are well-formed. Second, the complex semantics of some artifacts and relations can sometimes be only approximated by programs implementing translation operators. For example, a synchronizer that operates on program code may need to rely on static approximations of control and data flow.

Code queries for FSMLs exemplify both situations [16]. The precise FSML semantics relate model elements with structural and behavioral patterns in Java

code. However, the code queries implementing the reverse engineering for the behavioral patterns are incomplete and unsound approximations of the behavioral patterns. Furthermore, the code query evaluation engine relies on an incremental Java compiler, which allows for querying code that does not completely compile.

9 Related Work

In this section we discuss related works in three areas: data synchronization in optimistic replication, inconsistency management in software development, and model management and model transformation.

9.1 Data synchronization in optimistic replication

The need for synchronization arises in the area of optimistic replication, which allows replica contents to diverge in the short term in order to allow concurrent work practices and to tolerate failures in low quality communication links. Optimistic replication has applications to file systems, personal digital assistants, internet services, mobile databases, and software revision control. Saito and Shapiro [38] provide an excellent survey of optimistic replication algorithms, which are essentially synchronization algorithms. They distinguish the following phases of synchronization: update submission at multiple sites, update propagation, update scheduling, conflict detection and resolution, and commitment to final reconciliation result. The scheduling of update operations is of particular interest in the context of multiple master sites with background propagation, which leads to the challenge that not all update operations are received at all sites in the same order. Furthermore, Saito and Shapiro distinguish several key characteristics of optimistic replication:

- *Single vs. multi-master synchronization*: Synchronization scenarios can involve different numbers of master sites. Master sites are those that can modify replicas. In contrast, slave sites store read-only replicas. The scope of this tutorial is limited to master-slave (i.e., unidirectional) and master-master (i.e., bidirectional) synchronization.
- *State-transfer vs. operation transfer*: We discussed this distinction in Section 8.1.
- *Conflict detections and resolution granularity*: Conflicts may be easier to resolve if smaller sub-objects are considered.
- *Syntactic vs. semantic update operations*: Replicas can be compared syntactically or semantically. This distinction is concerned with the extent to which the synchronizer system is aware of the application semantics of the replicas and the update operations. Semantic approaches avoid some conflicts that would arise in syntactic approaches, but are more challenging to implement.
- *Conflict management*: This characteristic is concerned with the way the system defines and handles conflicts. Conflict detection policies can be syntactic or semantic. Conflict resolution may involve selecting one update among a

set of conflicting ones while the others are discarded, storing all conflicting updates in each synchronized replica, or allowing replicas to diverge for conflicting updates [27].
- *Update propagation strategy*: This dimension includes the degree of synchrony, e.g., pull vs. push strategies, and the communication topology, e.g., star vs. ad-hoc propagation.
- *Consistency guarantees*: Some synchronizers may guarantee consistency of the accessed replicas while other may give weaker guarantees, such as guaranteeing that the state of replicas will eventually converge to being consistent.

An additional dimension given by Foster et al. [27] is

- *Homogeneity vs. heterogeneity*: This dimension refers to the distinction whether the data to be synchronized adheres to a single schema or to different schemas expressed in the same schema language (e.g., relational algebra). The focus of this tutorial is on heterogeneous synchronization.

Saito and Shapiro [38] and Foster et al. [27] give many example of existing synchronization systems; however, Harmony [27] seems to be the only generic synchronizer handling *heterogeneous* replicas. Harmony is concerned with the special case of mappings which are functions. The same case is also studied in databases as the *view update problem*, e.g., see Bancilhon and Spyratos [39] and Gottlob et al. [40].

9.2 Data integration and schema mapping

Another related area is *data integration*, which is concerned with integrating data from multiple sources, such as different databases. A particular challenge in this context is *schema integration*, i.e., the integration of the vocabularies defined by the schemas, which is addressed by *schema matching*. Bernstein and Rahm [41] provide an excellent survey of approaches to automated schema matching.

9.3 Inconsistency management in software development

Software artifact synchronization is a topic in *inconsistency management* in software engineering [2, 5, 6, 8, 42]. Spanoudakis and Zisman [8] provide a survey of this area. They identify a broad set of activities related to inconsistency management: detection of overlaps (i.e., identification of relationships), detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, tracking (not all inconsistencies need to be resolved), and specification and application of an inconsistency management policy. Grundy and Hosking [7] explore architectures and user-interface techniques for inconsistency management in the context of multiple-view development environments.

9.4 Model management and model transformation

Software artifact synchronization is also closely related to *model management* and *model transformation*. In *model-driven software development* (MDSD) [12], models are specifications that are inputs to automated processes such as code generation, specification checking, and test generation. Furthermore, models in MDSD are typically represented as object graphs conforming to a class model usually referred to as a *metamodel*.

Model management is concerned with providing operators on models such as comparison, splitting, and merging. Bernstein et al. argued for the need of such generic model operators and the existence of mappings among models as first-class objects [43]. Later, Bernstein applied the model management operators to three problems: schema integration, schema evolution, and round-trip engineering [44]. Brunet et al. [4] wrote a manifesto for model management, in which they argue for an algebraic framework of model operators as a basis for comparing different approaches to model merging. Indeed, the use of operators in our design space was partly inspired by this manifesto. The *diff* operator corresponds to the homogeneous comparison operator presented in this tutorial. Furthermore, the manifesto refers to updates as *transformations* and to the application of updates as *patching*. The manifesto defines additional operators, e.g., *split* and *slice*. The operators in this tutorial treat the relation R as an implicit parameter. In contrast, the operators in the manifesto are defined explicitly over artifacts and *relations*. While the manifesto focuses on homogeneous merge, our design space is concerned with heterogeneous synchronization. In fact, bidirectional synchronizers with reconciliation can be understood as heterogeneous merge operations. One of the uses of model management is detecting and resolving inconsistencies in models, e.g., see work by Egyed [45] and Mens [46]. Sriplakich et al. [47] discuss a middleware approach to exchanging model updates among different tools. Finally, Diskin [48, 49] proposes using category theory as a mathematical formalism for expressing the operators for both homogeneous and heterogeneous generic model management.

Another related area is *model transformation*, which is concerned with providing an infrastructure for the implementation and execution of operations on models. Mens et al. [50] provide a taxonomy of model transformation and apply it to model transformation approaches based on graph transformations [51]. The taxonomy discusses several tool-oriented criteria such as level of automation, preservation, dealing with incomplete and inconsistent models, and automatic suggestion of transformations based on context. Czarnecki and Helsen [52] survey 26 approaches to model transformation. The survey and the design space presented in this tutorial both use a feature-based approach and have some features in common, such as target incrementality, source incrementality, and preservation of user edits in the target. In contrast to this tutorial, the survey mainly focuses on the different paradigms of transformation specification, such as relational, operational, template-based, and structure-driven approaches, and it does not consider reconciliation. Some ideas for an algebraic semantics for model transformations are presented by Diskin and Dingel [53].

The topic of bidirectional model transformation has recently attracted increased attention in the modeling community. Stevens [24] analyzes properties of the relational part of OMG's Query View Transformation (QVT) and argues that more basic research on bidirectional transformation is needed before practical tools will be fully realizable. Giese and Wagner [25] identify a set of concepts around bidirectional incremental transformations. In particular, they distinguish between *bijective* and *surjective* bidirectional transformations. The latter correspond to the situation where several sources correspond to a single target. Furthermore they refer to a transformation as *fully incremental* if the effort of synchronizing a source model change is proportional to the size of the source change. Finally, Ehrig et al. [35] study the conditions under which model transformations based on triple-graph grammars are reversible.

10 Conclusion

In this tutorial we explored the design space of heterogeneous synchronization, i.e., the synchronization of artifacts of different types. We presented a number of artifact operators that can be used in the implementation of synchronizers and presented 16 example synchronizers. The example synchronizers illustrate different approaches to synchronization and can be characterized along a number of dimensions, such as directionality, incrementality, original-target-dependency, and support for the reconciliation of concurrent updates. For some of the synchronizers, we provided examples of existing systems that implement a given approach to synchronization. Furthermore, we discussed a number of additional design decisions such as representation of updates, establishing correspondence among model elements, and strategies for selecting a single synchronization result from a set of alternatives. Finally, we discussed important works in related fields including data synchronization, inconsistency management in software engineering, model management, and model transformation.

Acknowledgments. The authors would like to thank Zinovy Diskin, Lech Tuzinkiewicz, and the anonymous reviewers for their valuable comments on earlier drafts of this tutorial.

References

1. Frederick P. Brooks, J.: No silver bullet: essence and accidents of software engineering. Computer **20**(4) (1987) 10–19
2. Nuseibeh, B., Kramer, J., Finkelstein, A.: Expressing the relationships between multiple views in requirements specification. In: ICSE. (1993) 187–196
3. Maier, M.W., Emery, D., Hilliard, R.: Software architecture: Introducing ieee standard 1471. Computer **34**(4) (2001) 107–109
4. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: GaMMa. (2006) 5–12
5. Balzer, R.: Tolerating inconsistency. In: ICSE. (1991) 158–165

6. Easterbrook, S., Nuseibeh, B.: Using viewpoints for inconsistency management. BCS/IEE Software Engineering Journal **11**(1) (1996) 31–43
7. Grundy, J., Hosking, J., Mugridge, W.B.: Inconsistency management for multiple-view software development environments. IEEE Trans. Softw. Eng. **24**(11) (1998) 960–981
8. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing Co. (2001) 329–380
9. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: IFIP. Volume 4037 of LNCS. (2006) 171–185 `http://www.lina.sciences.univ-nantes.fr/Publications/2006/JB06a`.
10. Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages; examples and algorithms. Technical Report 2007-18, ECE, Univeristy of Waterloo (2007)
11. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)
12. Stahl, T., Völter, M.: Model-Driven Software Development : Technology, Engineering, Management. John Wiley & Sons (2006)
13. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP. Volume 4609 of LNCS. (2007)
14. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
15. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: A progress report. In: OOPSLA International Workshop on Software Factories. (2005) On-line proceedings.
16. Antkiewicz, M., Tonelli Bartolomei, T., Czarnecki, K.: Automatic extraction of framework-specific models from framework-based application code. In: ASE. (2007) 214–223
17. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: POPL. (2005) 233–246
18. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: MoDELS. Volume 4199 of LNCS. (2006) 321–335
19. Eclipse Foundation: Java Emitter Templates Component. (2007) `http://www.eclipse.org/modeling/m2t/?project=jet`.
20. Eclipse Foundation: Eclipse Modeling Framework Project. (2007) `http://www.eclipse.org/modeling/emf/?project=emf`.
21. Nickel, U.A., Niere, J., Wadsack, J.P., Zündorf, A.: Roundtrip engineering with FUJABA. In: WSR, Fachberichte Informatik, Universität Koblenz-Landau (2000)
22. Aßmann, U.: Automatic roundtrip engineering. Electr. Notes Theor. Comput. Sci. **82**(5) (2003)
23. Sendall, S., Küster, J.M.: Taming model round-trip engineering. (2004)
24. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: MoDELS. Volume 4735 of LNCS. (2007) 1–15
25. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: MoDELS. Volume 4199 of LNCS. (2006) 543–557
26. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn (2007)

27. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Exploiting schemas in data synchronization. J. Comput. Syst. Sci. **73**(4) (2007) 669–689
28. ATLAS Group: ATLAS Transformation Language. (2007) `http://www.eclipse.org/m2m/atl/`.
29. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE. (2007) 164–173
30. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D.: Differencing and merging of architectural views. In: ASE. (2006) 47–58
31. IBM: Rational Software Modeler. (2007) `http://www-306.ibm.com/software/awdtools/modeler/swmodeler/`.
32. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of statecharts specifications. In: ICSE. (2007) 54–64
33. Reiss, S.P.: Automatic code stylizing. In: ASE. (2007) 74–83
34. Object Management Group: MOF QVT Final Adopted Specification. OMG Adopted Specification ptc/05-11-01 (2005) Available from `http://www.omg.org/docs/ptc/05-11-01.pdf`.
35. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: FASE. Volume 4422 of LNCS. (2007) 72–86
36. Xiong, Y., Hu, Z., Takeichi, M., Zhao, H., Mei, H.: On-site synchronization of software artifacts. Technical Report METR 2008-21, Department of Mathematical Informatics, University of Tokyo (2008) `http://www.ipl.t.u-tokyo.ac.jp/~xiong/papers/METR08.pdf`.
37. Pierce, B.C.: The weird world of bi-directional programming (2006) ETAPS invited talk, slides available from `http://www.cis.upenn.edu/~bcpierce/papers/lenses-etapsslides.pdf`.
38. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. **37**(1) (2005) 42–81
39. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6**(4) (1981) 557–575
40. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM Trans. Database Syst. **13**(4) (1988) 486–524
41. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. The VLDB Journal **10**(4) (2001) 334–350
42. Nuseibeh, B., Kramer, J., Finkelstein, A.: Viewpoints: meaningful relationships are difficult! In: ICSE. (2003) 676–681
43. Bernstein, P.A., Halevy, A.Y., Pottinger, R.A.: A vision for management of complex models. SIGMOD Rec. **29**(4) (2000) 55–63
44. Bernstein, P.: Applying model management to classical metadata problems. In: CIDR. (2003)
45. Egyed, A.: Fixing inconsistencies in UML design models. In: ICSE. (2007) 292–301
46. Mens, T., Straeten, R.V.D., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: MoDELS. Volume 4199 of LNCS. (2006) 200–214
47. Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting transparent model update in distributed case tool integration. In: SAC. (2006) 1759–1766
48. Diskin, Z., Kadish, B.: Generic model management. In Rivero, Doorn, Ferraggine, eds.: Encyclopedia of Database Technologies and Applications. Idea Group (2005) 258–265

49. Diskin, Z.: Mathemtics of generic specifications for model management. In Rivero, Doorn, Ferraggine, eds.: Encyclopedia of Database Technologies and Applications. Idea Group (2005) 351–366
50. Mens, T., Van Gorp, P.: A taxonomy of model transformation. In: Proc. Int'l Workshop on Graph and Model Transformation. (2005)
51. Mens, T., Van Gorp, P., Varro, D., Karsai, G.: Applying a model transformation taxonomy to graph transformation technology. In: Proc. Int'l Workshop on Graph and Model Transformation. (2005)
52. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3) (2006) 621–645
53. Diskin, Z., Diengel, J.: A metamodel independent framework for model transformation: Towards generic model management patterns in reverse engineering. In Favre, J.M., Gasevic, D., Laemmel, R., Winter, A., eds.: 3rd Int.Workshop on Metamodels, Schemas, Grammas and Ontologies for reverse engineering, ATEM-2006. (2006)