SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

October 2022, Vol. 65 202103:1–202103:24
https://doi.org/10.1007/s11432-020-3276-5

A theoretic framework of bidirectional transformation between systems and models

Xiao HE[1,2*], Zhenjiang HU[3] & Na MENG[4]

[1] *School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing* 100083*, China;*
[2] *Engineering Research Center of Intelligent Supercomputing, Ministry of Education, Beijing* 100083*, China;*
[3] *School of Information Science and Technology, Peking University, Beijing* 100871*, China;*
[4] *College of Engineering, Virginia Tech, Blacksburg* VA 24061*, USA*

Received 15 October 2020/Revised 11 November 2020/Accepted 10 March 2021/Published online 27 September 2022

Abstract Synchronization between systems and models has been explored in model-driven engineering to enable model-based system management. Despite its promising use, there is a lack of a theoretic foundation for state-based system-model synchronization. This paper proposes a theory for state-based system-model bidirectional transformation (BX), and defines seven combinators for system-model BX to facilitate the development of well-behaved synchronizer programs. A system-model BX is a single program that converts a system with a model consistently. Forwards, it creates a model according to a system as a conventional BX. Backwards, it generates a set of system edits, which can turn the current system into a new state that is consistent with the given model. System-model BX is fully aware of the domain constraints about how to change a system, and plans a reasonable execution order for those edits, rather than applying them blindly. The paper also demonstrates the use of system-model BX by building a generic system-model synchronizer and a concrete file system synchronizer.

Keywords bidirectional transformation, system-model synchronization, change propagation, model-driven engineering, system edit

Citation He X, Hu Z J, Meng N. A theoretic framework of bidirectional transformation between systems and models. Sci China Inf Sci, 2022, 65(10): 202103, https://doi.org/10.1007/s11432-020-3276-5

1 Introduction

Bidirectional programming [1, 2] aims to develop a single program, i.e., a bidirectional transformation (BX), to maintain the consistency between two data structures of different shapes, propagating the changes from one structure to the other. Recently, the principle of bidirectional programming has been adopted as the foundation of data synchronization [3–6] and model synchronization [7–10].

A BX over S and V (i.e., $S \leftrightarrow V$) is a pair (get, put)[1)] of functions, where get : $S \to V$ is called forward transformation that converts a source of type S into a view of type V, and put : $S \times V \to S$ is called backward transformation that takes the original source and the updated view as input and produces an updated source. A BX is well behaved if the following round-trip properties hold:

$$\mathrm{put}(s, \mathrm{get}(s)) = s, \tag{1}$$

$$\mathrm{get}(\mathrm{put}(s, v)) = v, \tag{2}$$

where Eq. (1) is the GetPut law, which states that a backward transformation immediately after a forward one should not cause any change, and Eq. (2) is the PutGet law, which states that the forward conversion of an updated source outputted by a backward conversion of a view produces the same view. In short, the GetPut and PutGet laws prescribe that a well-behaved BX must satisfy the

[*] Corresponding author (email: hexiao@ustb.edu.cn)

1) Without the loss of generality, this paper adopts the definition of asymmetric lens [1, 7].

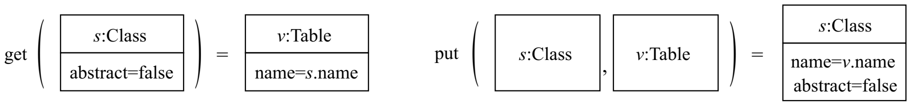


Figure 1 A simple BX between a class and a table.

following two conditions: (a) if a source and a view are synchronized, then neither the forward nor the backward transformation will make any further changes; and (b) the execution of a forward/backward transformation will result in a pair of synchronized source and view data.

Example 1. Figure 1 illustrates a simple BX between a class in UML class model and a table in the relational database management model. The forward transformation states that given a non-abstract class s, a table v must be created whose name is assigned s.name. The backward transformation declares that for a class s and a table v, we must update s by assigning `false` to s.abstract and assigning v.name to s.name. It is not difficult to verify that this BX is well-behaved.

Various theories and tools of bidirectional programming have been proposed. However, existing research efforts assume (either implicitly or explicitly) that the data to be bidirectionally converted are free data, i.e., values that can be changed without restriction. Targeting free data simplifies the research of BX. The free data own the following two characteristics.

(C1) Creating, altering, and deleting free data have no side effects. As a result, there is nearly no constraint on changing free data. In put function of Example 1, we alter both s.abstract and s.name when s is an abstract class whose name is different from the table's name. The two changes are independent and may happen in any order (even simultaneously).

(C2) Free data conform to a well-defined interface to read and edit. In put function of Example 1, we use name $= v$.name to edit the name of s with the value read from v in a highly declarative way.

In-memory data, such as JSON literals, are free data. The values in functional programming languages (e.g., records, lists, and dictionaries) are free data. In Java, a POJO can be viewed as free data. A software model, such as a UML model (see Example 1), can also be viewed as free data.

Nevertheless, there are many scenarios where we need to handle restricted data (i.e., non-free data). For example, in the technology of runtime models [11], we use a runtime model to represent and manage a system, e.g., a running software program. The runtime model monitors the running system to keep itself up-to-date. Moreover, if we change the runtime model, then the changes are also propagated to the running system. Obviously, this round-trip process is a bidirectional transformation.

In this paper, we focus on such bidirectional transformations between systems and models. We use the term system to denote restricted data. A system may refer to (but not limited to) a file system, a running software system, and an IoT system. A system is different from free data in the following two aspects.

(1) Interacting a system may have side effects and must follow some domain constraints (e.g., execution order). For example, supposing that we want to change an existing file named f to f' and create a new file named f in a certain folder, we must rename the existing file before creating the new file; otherwise, a file name collision arises.

(2) The read/edit interfaces of a system are domain knowledge. For instance, in Java SDK, to rename a file, we must call API java.io.File.renameTo(); to create a new file, we must call API java.io.File.createNewFile(). Different systems define diverse sets of APIs and impose various domain constraints. Without knowing these domain knowledge, we cannot manipulate a system.

Due to these differences, the existing BX technologies, especially bidirectional model transformation [8–10], cannot handle BXs between systems and models. Specifically, we cannot declaratively specify the expected state of the system as the output of the backward transformation (as we did in Example 1), according to which existing BX approaches do not know what APIs should be invoked and in what order they should be invoked to alter the original system.

This paper develops a general-purpose theoretic framework of bidirectional transformation between systems and models, named system-model BX. A system-model BX assumes that the source of the BX is a system, rather than free data. It regards the system type as an abstract data type, where the read/edit interfaces are abstract operations. It is also aware of the domain constraints among system edit operations. The forward transformation of a system-model BX behaves like that of a classical BX that converts a system into a model with the help of system read operations. The backward transformation of a system-model BX is very different from a classical BX—instead of producing an updated source directly, it generates a set of system edits by differencing the given model and the original system, then it orders the generated system edits based on domain constraints, and finally, it applies the edits to the original system to change the system into a new state that is consistent with the model. The system-model BX guarantees the correct synchronization between systems and models when necessary domain knowledge is correctly specified.

Paper organization. Section 2 discusses the related work. Section 3 formalizes the modules of systems. Section 4 proposes the concept of system-model BX and defines several combinators. Section 5 derives a generic system-model synchronizer based on system-model BX. The last section concludes the paper and discusses the future work.

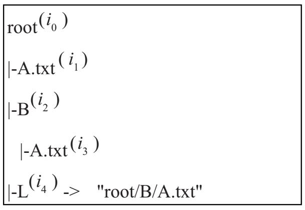
2 Related work

Runtime models. How to bridge the gap between systems and models is a fundamental problem and has been investigated in the community of runtime models. SM@RT [12], API2MoL [13], and EMF-Syncer [14] are some representative solutions. SM@RT [12] is a runtime-model-based approach to synchronizing running systems with models. SM@RT injects code that reads/writes the systems into the implementation of models so that when models are manipulated, the injected code will be executed instantly to access systems. In fact, SM@RT creates an adapter that converts system interfaces into model interfaces. API2MoL is an API-MDE bridge over API objects and models [13]. In brief, API2MoL maps every model change onto a system API so that when a model is changed, the corresponding API can be invoked. A virtue of API2MoL is that it can automatically infer the mapping between APIs and model changes. However, in many situations, it is impossible to establish such a simple API-model mapping. EMF-Syncer [14] is a change-driven synchronization framework that bidirectional converts system edits and model changes consistently. It can incrementally propagate the changes from the model domain to the system domain to achieve efficient synchronization. The major limitation of these solutions is that they are not fully aware of the domain constraints over system edits. To ensure that the system edits can be successfully applied, a user must manually follow the domain constraints while changing the model. If the model is changed by an automated model management operation, then existing solutions cannot ensure that required system edits can be successfully performed.

Model-code synchronization. Several approaches to round-trip engineering between source code and models [15–17] have been proposed. These approaches focused on a special case of system-model synchronization, i.e., code-model synchronization in round-trip engineering. They are not general purpose and cannot be extended to other cases. Some studies (e.g., [18]) used model-code synchronizers but did not discuss how to develop these synchronizers. There are also some studies (e.g., [19, 20]) that converts source code into internal representations (e.g., XML) to check the project-specific constraints. However, they cannot fix the violations in the internal representations and propagate the changes back.

Bidirectional transformation. There have been a large number of studies on bidirectional transformation. Particularly, the research efforts [6, 10, 21–26], as well as our previous studies [3, 4, 8, 9], discussed both algebraic and solver-based BX techniques for free data (e.g., XML files and models). However, this paper focuses on the synchronization between free data and restricted data. Fritsche et al. [27] proposed an approach to efficient model synchronization by automatically generating and applying the shortcut rule that merges multiple edits into an equivalent shortcut one. Hofmann et al. [6] proposed edit lens that is able to convert the edit from one domain to another domain bidirectionally while keeping the two domains consistent. Although edit lens focused on edits, it did not consider the execution order because it is still built on free data. Weidner et al. [28] discussed how to keep consistency between distributed replicas (free data). They argued that concurrent and non-commutative edits must be carefully merged to enable out-of-order execution. Different from [28], we argue that edits to a system should be executed in a proper order that may not be identical to the occurrence order of model changes. We regard our work as an extension, rather than a replacement, to BX theories, because our framework bridges systems and models, two diverse categories of data, which can be combined with existing BX approaches to handle complex synchronization problems.

Self-adaptive systems. Self-adaptive systems have been intensively discussed in the community of software engineering [29–32]. The major challenge of self-adaptive systems is the construction of a MAPE-K loop. Various engineering approaches have been proposed [30], such as model-based approaches (i.e., runtime models), architecture-based approaches, reflection approaches, and agent-based approaches.



root$^{(i_0)}$
|-A.txt$^{(i_1)}$
|-B$^{(i_2)}$
  |-A.txt$^{(i_3)}$
|-L$^{(i_4)}$ -> "root/B/A.txt"

Figure 2 A simple file system, where object identifiers are marked as superscripts.

Existing approaches mainly focused on how to monitor the systems, analyze the system qualities, and plan adaption strategies. We view our work as a complement to the technique of runtime models, which may further contribute to self-adaptive systems: our work focuses on how to maintain the casual relation between a system and a model via bidirectional programming, upon which complex model-based analysis and reasoning procedures can be adopted.

3 Systems and their operations

In this paper, Π denotes a set of systems. Π is also called the type of systems. We follow the design principle of encapsulation and regard Π as a black-box, rather than defining the inner structure of Π.

3.1 Systems, objects, and feature values

We assume that a system is structurally composed by a set of objects and feature values that specify the attributes (i.e., slots) and the relationships (i.e., links) of objects. Objects and feature values are defined in Definitions 1 and 2.

Definition 1 (Object)**.** An object is an instance of a certain class $\textsc{c}$. Every object has an object identifier $i \in \mathcal{I}$, where $\mathcal{I}$ denotes the set of all object identifiers. Given a system π, we say $i \mapsto \textsc{c} \in \pi$ if there is an object of class $\textsc{c}$, whose identifier is i and belongs to π; and we say $i \in \pi$ when $i \mapsto \_ \in \pi$. If we do not care about its class, then we may also say i is an object when there is no conflict.

The system class (or class) is the type of objects. A system class $\textsc{c}$ consists of many structural features (i.e., references and attributes). We assume that a reference is a directed relationship between two classes, and that an attribute is a directed relationship from a class to a primitive data type (e.g., string and int), and $\mathcal{F}$ is the set of all features. A reference may be either a containment reference (i.e., aggregation in OO terminology) or a non-containment reference. Instances of structural features (i.e., feature values) are called links and slots, according to OO terminology, which are defined as follows.

Definition 2 (Feature value)**.** A feature value has a form of $\mathcal{I} \xrightarrow{\mathsf{f}} \tau$, where τ is $\mathcal{I}$ if the feature f is a reference, or τ is primitive data type if the feature f is an attribute. A concrete feature value $i \xrightarrow{\mathsf{f}} v \in \pi$ means that in system π, object i is associated with value v via feature f. If f is a reference, then $i \xrightarrow{\mathsf{f}} v$ is a link and $v \in \mathcal{I}$; if f is an attribute, then $i \xrightarrow{\mathsf{f}} v$ is a slot and v is a primitive value.

A feature may be either ordered or unordered, depending on its domain semantics. For instance, the content of a folder (i.e., files and sub-folders) can be viewed as an unordered collection; the widgets of a running SWT dialog are ordered, since the order may affect the layout of the widgets. If a feature f is ordered, then any feature value is associated with an additional integer p (written as $i \xrightarrow{\mathsf{f}[p]} v$) to represent its position. We may omit the position integer when we do not care about it.

Example 2. Assume that Π is the set of file systems. There are three concrete classes $\textsc{c}_{\text{folder}}$, $\textsc{c}_{\text{file}}$, and $\textsc{c}_{\text{link}}$ to denote folders, (normal) files, and symbolic links. Besides, an unordered containment reference $\textsc{c}_{\text{folder}} \xrightarrow{\mathsf{f}_{\text{items}}} \textsc{c}_{\text{item}}$ is defined for Π to represent the relation between a folder and its contents, where $\textsc{c}_{\text{item}}$ is the abstract parent class of $\textsc{c}_{\text{folder}}$, $\textsc{c}_{\text{file}}$, and $\textsc{c}_{\text{link}}$. A singleton non-containment reference $\textsc{c}_{\text{link}} \xrightarrow{\text{pointsTo}} \textsc{c}_{\text{item}}$ is defined for Π to represent the link target of a symbolic link. Finally, an attribute $\textsc{c}_{\text{item}} \xrightarrow{\text{name}}$ `String` is also defined for Π to represent the folder/file name.

Figure 2 shows a simple file system π that contains two folders (i.e., root and root/B), two normal files (i.e., root/A.txt and root/B/A.txt), and a symbolic link (i.e., root/L). Object identifiers (e.g., inode IDs in Linux) are marked as superscripts. This file system can be abstractly represented as a set of objects and feature values as follows:

$$\pi = \left\{ \begin{array}{l} i_0 : \mathrm{C}_{\mathrm{folder}}, i_1 : \mathrm{C}_{\mathrm{file}}, i_2 : \mathrm{C}_{\mathrm{folder}}, i_3 : \mathrm{C}_{\mathrm{file}}, i_4 : \mathrm{C}_{\mathrm{link}}, \\ i_0 \xrightarrow{\mathrm{name}} \text{“root”}, i_1 \xrightarrow{\mathrm{name}} \text{“A.txt”}, i_2 \xrightarrow{\mathrm{name}} \text{“B”}, i_3 \xrightarrow{\mathrm{name}} \text{“A.txt”}, i_4 \xrightarrow{\mathrm{name}} \text{“L”}, \\ i_0 \xrightarrow{\mathrm{items}} i_1, i_0 \xrightarrow{\mathrm{items}} i_2, i_2 \xrightarrow{\mathrm{items}} i_3, i_0 \xrightarrow{\mathrm{items}} i_4, i_4 \xrightarrow{\mathrm{pointsTo}} i_2 \end{array} \right\}.$$

3.2 Abstract operations

We treat classes and features as abstract data types. Hence, we define some abstract operations, which should be provided by BX developers, to read and edit objects and systems, as follows.

- isAlive $: \mathcal{I} \to \Pi \to \{\texttt{true}, \texttt{false}\}$ is a domain-specific predicate that checks whether an object is alive in a system. In some systems, an object is only a handler that might refer to an invalid entity (i.e., not alive). For instance, in Java, an instance of java.io.File is just a file handler. It is possible to create a java.io.File object in memory that refers to an invalid file path (e.g., before java.io.File.createNewFile() is invoked). If an object is not alive, then it means that certain edits to this object may fail.
- $\mathrm{con}_{\mathrm{C}} : \mathcal{I} \times [\mathcal{F}[\tau]] \to \Pi \to \Pi$[2] is an abstraction of object constructor for certain class C. The first parameter $\mathcal{I}$ is a new object identifier, and the second parameter $[\mathcal{F}[\tau]]$ is a list of feature initializers. $\mathrm{con}_{\mathrm{C}}\ (i, [\mathsf{f}_1[v_{1,1}, v_{1,2}, \ldots], \mathsf{f}_2[v_{2,1}, v_{2,2}, \ldots], \ldots])\ \pi = \pi'$ changes the system state from π to π' by creating an object i of class C and initializes feature f_j of this object to $[i \xrightarrow{\mathsf{f}_j} v_{j,1}, i \xrightarrow{\mathsf{f}_j} v_{j,2}, \ldots]$ in the new state π', and f_j is also called a (formal) constructor parameter.
- $\mathrm{des}_{\mathrm{C}} : \mathcal{I} \to \Pi \to \Pi$ is the destructor of C. $\mathrm{des}_{\mathrm{C}}\ i\ \pi = \pi'$ deletes object i from system state π to form a new state π'. Note that object destructor is not necessary in some systems. For instance, in Java-based systems, an object does not need a destructor.
- $\mathrm{get}_{\mathsf{f}} : \mathcal{I} \to \Pi \to [\mathcal{I} \xrightarrow{\mathsf{f}} \tau]$ is the getter operation of feature f. $\mathrm{get}_{\mathsf{f}}\ i\ \pi = [i \xrightarrow{\mathsf{f}} v_1, i \xrightarrow{\mathsf{f}} v_2, \ldots]$ returns the feature values of object i in system π and $i \xrightarrow{\mathsf{f}} v \in \pi \Leftrightarrow i \xrightarrow{\mathsf{f}} v \in \mathrm{get}_{\mathsf{f}}\ i\ \pi$. Because we do not assume that the structures of systems are known, we need such getter operations to access the inner data. We always assume that a getter operation returns a collection.

For ordered feature f and a collection of feature values, we define the following edit operations.

- $\mathrm{insert}_{\mathsf{f}} : \mathcal{I} \times \tau \times \mathbb{N} \to \Pi \to \Pi$ is an additive edit for f. $\mathrm{insert}_{\mathsf{f}}\ (i, v, p)\ \pi = \pi'$ changes system state π to π' by adding a new link/slot $i \xrightarrow{\mathsf{f}} v$ at position p.
- $\mathrm{remove}_{\mathsf{f}} : \mathcal{I} \times \mathbb{N} \to \Pi \to \Pi$ is a deletion edit for f. $\mathrm{remove}_{\mathsf{f}}\ (i, p)\ \pi = \pi'$ changes system state π to π' by deleting a new link/slot at position p.
- $\mathrm{modify}_{\mathsf{f}} : \mathcal{I} \times \tau \times \mathbb{N} \times \mathbb{N} \to \Pi \to \Pi$ is a modification edit for f. $\mathrm{modify}_{\mathsf{f}}\ (i, v, p_s, p_t)\ \pi = \pi'$ changes the system state π to π' by replacing a link/slot at position p_s with a new link/slot $i \xrightarrow{\mathsf{f}} v$ and moving it to position p_t. If $i \xrightarrow{\mathsf{f}} v$ is the value at position p_s, then this edit just reorders it.
- $\mathrm{moveIn}_{\mathsf{f}} : \mathcal{I} \times \tau \times \mathbb{N} \times \mathcal{I} \to \Pi \to \Pi$ is an additive edit for f. $\mathrm{moveIn}_{\mathsf{f}}\ (i, v, p, i')\ \pi = \pi'$ changes system state π to π' by adding $i \xrightarrow{\mathsf{f}} v$ at position p, where v was originally a feature value of object i'.
- $\mathrm{moveOut}_{\mathsf{f}} : \mathcal{I} \times \mathbb{N} \times \mathcal{I} \to \Pi \to \Pi$ is a deletion edit for f. $\mathrm{moveOut}_{\mathsf{f}}\ (i, p, i')\ \pi = \pi'$ changes system state π to π' by deleting a link/slot at position p, where the value will be further moved to object i'.

Note that $\mathrm{moveIn}_{\mathsf{f}}$ and $\mathrm{moveOut}_{\mathsf{f}}$ are only defined when f is a containment reference. In OO methodology, if a link $i \xrightarrow{\mathsf{f}} i_c$ is removed and f is a containment reference, then there is an implication that i_c will also be deleted. Nevertheless, $\mathrm{moveOut}_{\mathsf{f}}$ states that the value to be moved out will not be deleted; rather, it will further be moved into another container.

Similar to the ordered case, there are also five edits for the unordered feature, including $\mathrm{insert}_{\mathsf{f}} : \mathcal{I} \times \tau \to \Pi \to \Pi$, $\mathrm{remove}_{\mathsf{f}} : \mathcal{I} \times \tau \to \Pi \to \Pi$, $\mathrm{modify}_{\mathsf{f}} : \mathcal{I} \times \tau \times \tau \to \Pi \to \Pi$, $\mathrm{moveIn}_{\mathsf{f}} : \mathcal{I} \times \tau \times \mathcal{I} \to \Pi \to \Pi$, and $\mathrm{moveOut}_{\mathsf{f}} : \mathcal{I} \times \tau \times \mathcal{I} \to \Pi \to \Pi$.

Note that feature edit operations are only defined for the feature that is not a constructor parameter, because we assume that a constructor parameter is immutable since the object is created.

Example 3. Assume that f is an ordered feature. As shown in Figure 3, initially, in system π_1, there are three values for i and f, i.e., $[i \xrightarrow{\mathsf{f}} v_1, i \xrightarrow{\mathsf{f}} v_2, i \xrightarrow{\mathsf{f}} v_3]$. After executing $\mathrm{insert}_{\mathsf{f}}\ (i, v_4, 3)\ \pi_1$, $i \xrightarrow{\mathsf{f}} v_4$

[2] In this paper, all system edits are defined in the curried form. In this way, the partial application of a system edit, e.g., $\mathrm{con}_{\mathrm{C}}\ (i, [\mathsf{f}_1[v_{1,1}, v_{1,2}, \ldots], \mathsf{f}_2[v_{2,1}, v_{2,2}, \ldots], \ldots])$, is a function of systems. In the rest of this paper, an edit, e.g., $\mathrm{con}_{\mathrm{C}}$, also refers to its partial application and is viewed as a function of systems, if there is no conflict in context.

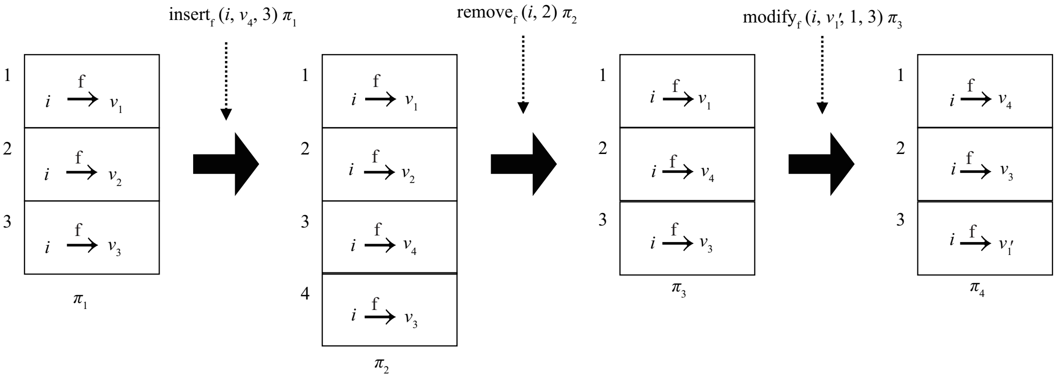


Figure 3 Changing ordered feature values.

is inserted at position 3 in the new system π_2. Then, $\text{remove}_{\mathsf{f}}$ $(i, 2)$ π_2 deletes $i \xrightarrow{\mathsf{f}} v_2$ and results in π_3. Finally, $\text{modify}_{\mathsf{f}}$ $(i, v_1', 1, 3)$ π_3 replaces $i \xrightarrow{\mathsf{f}} v_1$ with $i \xrightarrow{\mathsf{f}} v_1'$, and moves the value to the end of the collection in π_4 (consequently, $i \xrightarrow{\mathsf{f}} v_4$ and $i \xrightarrow{\mathsf{f}} v_3$ are moved forward).

Now we are able to define system edits.

Definition 3 (System edits)**.** Given a system type Π, supposing that the classes $\textsc{c}_1, \textsc{c}_2, \ldots$ and features $\mathsf{f}_1, \mathsf{f}_2, \ldots$ are defined in Π, $\partial\Pi$ is the set of system edits that is defined as follows:

$$\partial\Pi = \bigcup_{\textsc{c}_i} \{\text{con}_{\textsc{c}_i}, \text{des}_{\textsc{c}_i}\} \cup \bigcup_{\mathsf{f}_i} \{\text{insert}_{\mathsf{f}_i}, \text{remove}_{\mathsf{f}_i}, \text{modify}_{\mathsf{f}_i}, \text{moveIn}_{\mathsf{f}_i}, \text{moveOut}_{\mathsf{f}_i}\}.$$

Given any class $\textsc{c}$ (any feature f) in Π, we can also define corresponding set of edits $\partial\textsc{c}$ ($\partial\mathsf{f}$). Since $\partial\textsc{c}$ ($\partial\mathsf{f}$) is a subset of $\partial\Pi$, it is safe to cast $\partial\textsc{c}$ ($\partial\mathsf{f}$) to $\partial\Pi$.

3.3 Contracts of edits

As discussed above, system edits are domain-specific because their implementation and constraints must be provided by BX developers. If the implementation and/or the execution order of system edits are incorrect, then we cannot guarantee the well-behavedness of system-model BX. For example, supposing that $\text{insert}_{\mathsf{f}}$ does not insert a value but removes one, then the transformation will surely fail. Hence, we must be able to verify the correctness of the implementation and the execution order of system edits.

We resort to contract-based programming to address this issue. Specifically, for every system edit e, a pair of pre-/post-conditions must be defined to specify when the edit can be applied (i.e., precondition pre_e) and the expected effect on the system (i.e., postcondition post_e).

As expected, the postcondition post_e is viewed as a predicate on the system, i.e., $\Pi \to$ Boolean. However, in this paper, the precondition pre_e is a binary predicate $\Pi \times 2^{\partial\Pi} \to$ Boolean, where the first parameter denotes the current system state and the second parameter denotes the set of all remaining edits. pre_e returns true if e can be applied to the given system before the set of remaining edits.

Remark 1. By specifying the pre-/post-conditions of an edit, we may adopt state-of-the-art static verification technologies [33] or testing technologies [34–36] to verify the implementation of this edit.

Remark 2. Although the contract of an edit is generally domain specific, every edit declared in Subsection 3.2 does have some generic conditions[3]. When we are creating the domain implementation, we only need to append the domain-specific conditions.

Remark 3. The preconditions can be used to find and check the execution order of edits. Given a sequence $[e_1, e_2, \ldots, e_n]$, we say this sequence in the correct order for the initial system state π_0 only if

$$\forall p \in [0, n] \big(\text{pre}_{e_p}(\pi_{p-1}, \{e_{p+1}, e_{p+2}, \ldots, e_n\}) \wedge \pi_p = e_p\ \pi_{p-1}\big), \tag{3}$$

where $e\ \pi_p$ denotes performing edit e on system π. Eq. (3) means that the former edit does not break the precondition of the later so that $e_1, e_2, \ldots, e_n$ can be successfully executed one-by-one.

[3] Project page. http://www.softlang.cn/index.php/Research/Bidirectional_Transformation/System-Model_Synchronization.

Take the file system as an example. Assume that e_1 is to create (insert) a new file named f and e_2 is to rename (modify) an existing file f to g. Obviously, e_1 must satisfy a precondition that there is no other file named f; and e_2 must satisfy a precondition that there is a file named f, and must also satisfy a postcondition that the file named f is renamed to g. Given a folder that contains a file named f (i.e., a system π_0), e_2 must be executed before e_1 because $\mathrm{pre}_{e_2}(\pi_0) \wedge \pi_1 = e_2\ \pi_0 \wedge \mathrm{pre}_{e_1}(\pi_1)$ but $\neg \mathrm{pre}_{e_1}(\pi_0, \_)$.

Remark 4. By viewing pre_e as a binary predicate, we get a chance to straightforwardly encode the ordering constraints among edits into the precondition because we can specify before which edits, the given edit cannot be performed. For example, supposing that $e = \mathrm{moveIn}_{\mathsf{f}}\ (i, v, i')$, we can append the condition $\nexists e'(e' \in es \wedge e' = \mathrm{moveOut}_{\mathsf{f'}}\ (i', v, i))$ to $\mathrm{pre}_e(\pi, es)$ to specify that a moveIn cannot be performed before a moveOut if they intend to move the same value.

In theory, Eq. (3) can be checked with the help of state-of-the-art model checking and SMT solving technologies, e.g., Alloy [37], by encoding the pre-/post-conditions and the initial state into a set of logic expressions and asking Alloy to verify those expressions. It is also possible to ask Alloy to compute the execution order of a set of edits by finding a valid sequence that satisfies Eq. (3). It is necessary to emphasize the following facts.

- Given a set of edits, there may be many valid edit sequences that satisfy Eq. (3). It is because if two edits are totally commutable, then they can be executed in any order.
- Given a set of edits, it is also possible that there is no valid edit sequence that satisfies Eq. (3). In such a case, there must be some conflicts among those edits.
- We consider solving Eq. (3) with a solver to be theoretically feasible. However, it might be inexpressive and/or inefficient in practice to encode and solve Eq. (3) in a first-order-logic-based solver. The design of a practical solution is temporarily out of our concern.

3.4 Module of system

As discussed above, given a set of system edits, we can compute a reasonable execution order, i.e., an edit sequence satisfying Eq. (3). This can be formalized as a function[4] $\rhd : 2^{\partial\Pi} \to \Pi \to [\partial\Pi]$. Given a sequence of edits in a reasonable order, we can apply this edit sequence to a certain system. We also define an edit application function $\odot : [\partial\Pi] \to \Pi \to \Pi$ that takes a sequence of edits and an initial system as input, and applies the edits one by one. The function $\odot$ is recursively defined as follows.

- $[\,] \odot \pi = \pi$;
- $[e, e_1, \ldots] \odot \pi = [e_1, \ldots] \odot (e\ \pi)$.

Finally, we put everything together into the module of systems, which is defined as follows.

Definition 4 (System module). A system module is a tuple $(\Pi, \mathsf{uuid}, \partial\Pi, \rhd, \odot)$, where Π is a system type, $\partial\Pi$ is the set of system edits, $\rhd$ is the edit ordering function, and $\odot$ is the edit application function. uuid can be viewed as a function $\mathsf{uuid}^{(\mathcal{I})} : \mathbb{U} \to \mathcal{I}$, where $\mathbb{U}$ is any value. $\mathsf{uuid}^{(i')}\ u$ returns a new object identifier for given input u (so we can retrieve the same identifier by sending the same u), and the optional i' denotes the old identifier that is intended to be replaced by the new identifier (see Subsection 5.1). In the rest of this paper, we still use Π to denote system module. For simplicity, given a set es of edits and a system π, $\mathrm{es} \rhd \odot \pi \equiv (\mathrm{es} \rhd \pi) \odot \pi$.

We can also define object module and feature value module in the same way. Similar to the discussion in Definition 3, the object module and feature value module can also be cast to the system module.

Definition 5 (Disjoint systems). Given two systems $\pi_1 \in \Pi_1, \pi_2 \in \Pi_2$ (Π_1 may be equal to Π_2), if there is an edit e for π_i $(i = 1, 2)$, and $[e] \odot (\pi_1 \cup \pi_2)$ is equal to $([e] \odot \pi_i) \cup \pi_{3-i}$), then π_1, π_2 are disjoint about e, i.e., e affects π_i only when e can be applied to π_i. Supposing $\mathrm{es} = [e, e_1, \ldots]$ is an edit sequence and e is defined for π_i, then π_1, π_2 are disjoint about es if π_1, π_2 are disjoint about e, and $[e] \odot \pi_i$ and π_{3-i} are disjoint about $[e_1, \ldots]$. If π_1 and π_2 are disjoint about any valid edit sequence, then they are disjoint. If any $\pi_1 \in \Pi_1$ and $\pi_2 \in \Pi_2$ are disjoint, then Π_1 and Π_2 are disjoint.

Remark 5. Two isolated systems are obviously disjoint because their changes will not affect each other. Nevertheless, Definition 5 also implies that a single system may conceptually be viewed as a composition of two disjoint sub-systems if the system edits are deliberately defined. For example, a file system π can be viewed as two sub-systems π_D and π_F. π_D keeps the directory hierarchy of this file system, and π_F maintains all file information. Although π_D and π_F are mutually related, they are disjoint about the

[4] In fact, $\rhd$ is not a (deterministic) function, because there may be multiple orders for the same set of edits.

edit e that deletes an empty folder: when e can be applied, the folder to be deleted must be empty, so e affects π_D only. However, if e can delete non-empty folders, then π_D and π_F are not disjoint.

The concept of disjoint systems is very important in this paper. It allows us to isolate the effect of a system edit to a sub-system from other disjoint parts.

Given two system modules Π_1 and Π_2, we can combine them into a new system module Π_{1+2} by merging all the definitions. Specifically, $\partial\Pi_{1+2} = \partial\Pi_1 \cup \partial\Pi_2$. Nevertheless, $\rhd_{1+2}$ may require additional knowledge to arrange a set of edits that mix the edits from both $\partial\Pi_1$ and $\partial\Pi_2$ into an edit sequence (unless $\rhd_1$ and $\rhd_2$ already embody those knowledge). Given a system $\pi_{1+2} \in \Pi_{1+2}$ and an edit sequence $\mathrm{es}_{1+2} \subseteq \partial\Pi_{1+2}$, supposing that π_1 and es_1 are projections of π_{1+2} and es_{1+2} onto Π_1 and $\partial\Pi_1$, respectively, if es_1 is a valid sequence for π_{1+2}, then es_1 must also be a valid sequence for π_1, and $\mathrm{es}_1 \odot \pi_1$ must be the projection of $\mathrm{es}_{1+2} \odot \pi_{1+2}$ onto Π_1.

4 System-model bidirectional transformation

4.1 Definition

Before we define system-model BX, we must formally define models and model types.

Definition 6 (Model and model type)**.** A model type (say Y) denotes a set of models. In model-driven engineering, a model type is usually encoded as a metamodel. We adopt an EMF[5]-like type system, and assume that a model type consists of a set of EClasses and EStructuralFeatures. An EStructuralFeature is either an EReference or an EAttribute. A model y consists of a set of model elements (i.e., EObjects—the instances of EClasses) and relationships (i.e., Settings—instances of EStructuralFeatures). The set of all model elements is denoted as $\mathcal{I}_E$. Two models y_1 and y_2 are disjoint if $y_1 \cap y_2 = \emptyset$. Two model types Y_1 and Y_2 are disjoint if any $y_1 \in Y_1$ and $y_2 \in Y_2$ are disjoint.

Because a model conforms to well defined interfaces, e.g., EObject and Setting in EMF, it can be read/written as free data. Hence, for any EClass Y, $i_E \mapsto Y$ declares an model element i_E of type Y; For a feature f_E, $i_E \xrightarrow{\mathsf{f}_E} v$ declares a relationship, and $i_E.\mathsf{f}_E$ returns the values associated with i_E via f_E. We also term $i_E \xrightarrow{\mathsf{f}_E} v$ a feature value of i_E. For an ordered feature, we assume that a value is associated with its position, say $i_E \xrightarrow{\mathsf{f}_E[p]} v$.

A classical BX over types S and V consists of two functions get : $S \rightarrow V$ and put : $S \rightarrow V \rightarrow S$. However, the synchronization between a system and a model requires additional information that preserves the correspondences between objects and model elements (as known as casual relations [11] and complement [6]). We use Ψ to denote the correspondences, and we will discuss this later. Now we define system-model BX as follows.

Definition 7 (System-model BX)**.** A system-model BX l between system module Π and model type Y, denoted as $l : \partial\Pi \overset{\Psi}{\longleftrightarrow} Y$, consists of a correspondence set Ψ, two functions $\Rrightarrow\ : \Pi \times \Psi \rightarrow Y \times \Psi$ (i.e., the forward transformation) and $\Lleftarrow\ : \Pi \times \Psi \times Y \rightarrow 2^{\partial\Pi} \times \Psi$ (i.e., the backword transformation), and a ternary consistency relation $K \subseteq \Pi \times \Psi \times Y$ that defines synchronized states, such that

- $(\epsilon, \epsilon, \epsilon) \in K$, where ϵ means empty data that belongs to any data;
- if $\Rrightarrow (\pi, \psi) = (y, \psi')$, then $(\pi, \psi', y) \in K$;
- if $(\pi, \psi, y) \in K$, then $\Lleftarrow (\pi, \psi, y) = (\emptyset, \psi)$;
- if $(\pi, \psi, y) \notin K$, $\Lleftarrow (\pi, \psi, y) = (d\pi, \psi')$, and $d\pi \rhd \pi$ is defined, then $(d\pi \rhd \odot \pi, \psi', y) \in K$.

Remark 6. Given $d\pi$ that is generated by $l.\!\Lleftarrow\!(\pi, \_, y)$, the fact that $d\pi \rhd \pi$ is undefined implies that the model y is invalid because π cannot be changed to a consistent state with existing system edits. In such a case, we must alter y to make it synchronizable.

Remark 7. System-model BX is very different from classical BX because in backward direction, it is impossible for system-model BX to obtain an intermediate system. While in classical BX, the intermediate result is essential for BX composition and runtime checks, such as branch condition checks. Hence, existing BX approaches cannot handle the synchronization between systems and models.

Similar to classical BX, a system-model BX must satisfy two round-trip properties as follows:

$$\Rrightarrow\!(\pi, \psi) = (y, \psi') \Rightarrow\, \Lleftarrow\!(\pi, \psi', y) = (\emptyset, \psi'), \tag{4}$$

[5] EMF project. https://www.eclipse.org/modeling/emf/.

$$\Lleftarrow(\pi, \psi, y) = (d\pi, \psi') \wedge d\pi \triangleright \pi \text{ is defined} \Rightarrow \Rrightarrow(d\pi \triangleright \odot \pi, \psi') = (y, \psi'). \tag{5}$$

Despite the different forms, Eqs. (4) and (5) share the same rationale with Eqs. (1) and (2): (a) if a system and a model are already synchronized, then neither the forward transformation ($\Rrightarrow$) nor the backward one ($\Lleftarrow$) will cause any further changes, and (b) both $\Rrightarrow$ and $\Lleftarrow$ will reach a synchronized state. We call Eqs. (4) and (5) implementation-perfect round-trip properties because we require that all system edits and function $\triangleright$ are correctly implemented. Our approach guarantees implementation-perfect round-trip properties, but provides no warrant for the case when there is any error in the implementation of system edits and $\triangleright$, which should be tested before the construction of system-model BXs.

Definition 8 (Correspondence)**.** The correspondence data in our approach tells how objects in a system and model elements in a model are mutually mapped. It may contain any information. For simplicity, we assume that the correspondence data $\psi \in \Psi$ can be used as a partial bijective function, i.e., $\psi \subset \mathcal{I} \times \mathcal{I}_E$.

We can update ψ by replacing a certain object-element mapping as follows: $\psi[i \to i_E] \equiv \{(i, i_E)\} \cup \{(x, y) | (x, y) \in \psi \wedge x \neq i \wedge y \neq i_E\}$. For two $\psi_1 \in \Psi_1$ and $\psi_2 \in \Psi_2$, $\psi_1 \uplus \psi_2$ is called consistent union, where $\psi_1 \uplus \psi_2 = \psi_1 \cup \psi_2$ if $\psi_1 \cup \psi_2$ is still a partial bijective mapping; otherwise, $\psi_1 \uplus \psi_2 = \perp$.

Example 4. The system type Π_{VM} denotes the set of virtual machines (VMs), and each VM has an attribute ip (a constructor parameter). Π_{VM} is equipped with two edits (i.e., $\mathrm{cons}_{\mathrm{VM}}\ i\ \mathsf{ip}[v]$ and $\mathrm{des}_{\mathrm{VM}}\ i$) that creates and deletes a VM, respectively. The model type Y_{VM} denotes the set of VM elements, each of which has an attribute $\mathsf{ip_E}$ to denote the IP value. We consider a VM and a VM element are consistent if they hold the same IP address. Hence, we can define a consistency relation $K_{\mathrm{VM}} = \{(\pi, \{(i, i_E)\}, y) | \pi = \{i, i \xrightarrow{\mathsf{ip}} v\} \wedge y = \{i_E, i_E \xrightarrow{\mathsf{ip_E}} v\}\}$. To ensure the consistency relation, the forward transformation can be defined as $\Rrightarrow(\{i, i \xrightarrow{\mathsf{ip}} v\}, \psi) = (\{i_E, i_E \xrightarrow{\mathsf{ip_E}} v\}, \{(i, i_E)\})$, which creates a new VM element that is consistent with the given VM; the backward transformation can be defined as

$$\Lleftarrow(\{i, i \xrightarrow{\mathsf{ip}} v\}, \psi, \{i_E, i_E \xrightarrow{\mathsf{ip_E}} u\}) = \begin{cases} (\emptyset, \{(i, i_E)\}), & \text{if } v = u, \\ (\{\mathrm{des}_{\mathrm{VM}}\ i, \mathrm{cons}_{\mathrm{VM}}\ \mathsf{uuid}^{(i)}(i_E)\ \mathsf{ip}[u]\}, \{(\mathsf{uuid}^{(i)}(i_E), i_E)\}), & \text{otherwise.} \end{cases}$$

It is not difficult to verify that $K_{\mathrm{VM}}, \Rrightarrow$, and $\Lleftarrow$ satisfy Definition 7. Moreover, they also satisfy Eqs. (4) and (5). Hence, $\Rrightarrow$ and $\Lleftarrow$ form a well-behaved system-model BX between Π_{VM} and Y_{VM}.

Remark 8. Definition 7 and Eqs. (4) and (5) describe what is a well-behaved system-model BX, but do not tell how to construct one efficiently. If there are some predefined primitive system-model BXs, then we can combine them into a complex BX by using the combinators defined in Subsection 4.2. However, how to construct primitive system-model BX is out of our concern. We assume that putback-based bidirectional programming, such as [3–5, 8], may facilitate this task.

Remark 9. The indented usage of system-model BX is as follows. Given a system of type Π and a model type Y, the forward transformation $\Rrightarrow$ generates a model-based abstraction of the system. Afterward, a user or an automated model management operation, e.g., model transformation [38] and model fixing [39], changes the generated model. Finally, the backward transformation $\Lleftarrow$ generates a set of system edits by differencing the system and the model. The generated system edits have the ability to change the system into a new state that is consistent with the model, if all the edits are successfully applied.

4.2 Combinators

In this subsection, we present some combinators of system-model BXs to facilitate the development well-behaved system-model BXs.

Parallel union ($\otimes$). If Π_1 and Π_2 are disjoint, then $\Pi_1 \otimes \Pi_2$ forms a new module, where $\otimes$ denotes disjoint union. Intuitively, for any $\pi_{1+2} \in \Pi_1 \otimes \Pi_2$, we can always split π_{1+2} into $\pi_1 \in \Pi_1$ and $\pi_2 \in \Pi_2$, such that $\pi_1 \cap \pi_2 = \pi_{1+2}$ and $\pi_1 \cap \pi_2 = \emptyset$. For model types, $\otimes$ is also defined analogously.

For two disjoint modules Π_1 and Π_2 and two disjoint model types Y_1 and Y_2, the combinator parallel union combines two system-model BXs $l_1 : \partial\Pi_1 \xleftrightarrow{\Psi_1} Y_1$ and $l_2 : \partial\Pi_2 \xleftrightarrow{\Psi_2} Y_2$ into $l_1 \otimes l_2 : \partial(\Pi_1 \otimes \Pi_2) \xleftrightarrow{\Psi_1 \uplus \Psi_2} Y_1 \otimes Y_2$, where

(1) $l_1 \otimes l_2.K = \{(\pi_1 \cup \pi_2, \psi_1 \uplus \psi_2, y_1 \cup y_2) | (\pi_1, \psi_1, y_1) \in l_1.K \wedge (\pi_2, \psi_2, y_2) \in l_2.K\}$;

(2) $l_1 \otimes l_2. \Rrightarrow (\pi_{1+2}, \psi_{1+2}) = (y_1 \cup y_2, \psi'_1 \uplus \psi'_2)$, such that $\pi_1 \in \Pi_1 \wedge \pi_2 \in \Pi_2 \wedge \pi_1 \cup \pi_2 = \pi_{1+2}$, $\psi_1 \in \Psi_1 \wedge \psi_2 \in \Psi_2 \wedge \psi_1 \uplus \psi_2 = \psi_{1+2} \neq \perp$, and $l_j.\Rrightarrow(\pi_j, \psi_j) = (y_j, \psi'_j)$ $(j = 1, 2)$;

(3) $l_1 \otimes l_2.\Lleftarrow(\pi_{1+2}, \psi_{1+2}, y_{1+2}) = (d\pi_1 \cup d\pi_2, \psi'_1 \uplus \psi'_2)$, such that $\pi_1 \in \Pi_1 \wedge \pi_2 \in \Pi_2 \wedge \pi_1 \cup \pi_2 = \pi_{1+2}$, $\psi_1 \in \Psi_1 \wedge \psi_2 \in \Psi_2 \wedge \psi_1 \uplus \psi_2 = \psi_{1+2} \neq \perp$, $y_1 \in Y_1 \wedge y_2 \in Y_2 \wedge y_1 \cup y_2 = y_{1+2}$, $l_j.\Lleftarrow(\pi_j, \psi_j, y_j) = (d\pi_j, \psi'_j)$ $(j = 1, 2)$, and $(d\pi_1 \cup d\pi_2) \rhd \pi_{1+2}$ is defined.

Intuitively, if a system and a model can be partitioned into disjoint parts that can be synchronized by l_1 and l_2, respectively, then $l_1 \otimes l_2$ can synchronize the complete system and model. Note that we do not require $\psi_1 \cap \psi_2 = \emptyset$, as long as they are changed consistently.

Example 5. Assume that l_A and l_B are two system-model BXs that bidirectionally convert the local file systems of two computers `A` and `B` with two models, respectively. Because the two file systems are completely unrelated (disjoint), the edits generated by l_A (l_B) never change the file system on `B` (`A`). Thus, $l_A \otimes l_B$ synchronizes the two file systems with the corresponding two models in parallel.

Theorem 1. If l_1 and l_2 are system-model BX, then $l_1 \otimes l_2$ is also a system-model BX.

Proof. Because Π_1 and Π_2 are disjoint, and Y_1 and Y_2 are disjoint, l_1 and l_2 do not interrupt each other. The definition of parallel union preserves the well-behavedness of l_1 and l_2. Especially, when applying the set $d\pi_1 \cup d\pi_2$ of edits, when $d\pi_1 \cup d\pi_2 \rhd \pi_{1+2}$ is defined, $d\pi_1 \cup d\pi_2 \rhd\!\odot \pi_{1+2}$ is conceptually equivalent to $(d\pi_1 \rhd\!\odot \pi_1) \cup (d\pi_2 \rhd\!\odot \pi_2)$.

Sequential union (;). Now consider the case that two system-model BXs are chained with the help of correspondence data. When Π_1, Π_2, and Y_1, Y_2 are disjoint, given two system-model BXs $l_1 : \partial\Pi_1 \xleftrightarrow{\Psi_1} Y_1$ and $l_2 : \partial\Pi_2 \xleftrightarrow{\Psi_2} Y_2$, $l_1; l_2 : \partial(\Pi_1 \otimes \Pi_2) \xleftrightarrow{\Psi_2} Y_1 \otimes Y_2$ is defined as follows:

(1) $l_1; l_2.K = \{(\pi_1 \cup \pi_2, \psi_2, y_1 \cup y_2) | \psi_1 \subseteq \psi_2 \wedge (\pi_1, \psi_1, y_1) \in l_1.K \wedge (\pi_2, \psi_2, y_2) \in l_2.K\}$;

(2) $l_1; l_2.\Rrightarrow(\pi_{1+2}, \psi_2) = (y_1 \cup y_2, \psi'_2)$, such that $\pi_{1+2} = \pi_1 \cup \pi_2 \wedge \pi_1 \in \Pi_1 \wedge \pi_2 \in \Pi_2$, $\psi_1 \subseteq \psi_2 \wedge \psi_1 \in \Pi_1$, $l_1.\Rrightarrow(\pi_1, \psi_1) = (y_1, \psi'_1)$, $l_2.\Rrightarrow(\pi_2, (\psi_2 - \psi_1) \uplus \psi'_1) = (y_2, \psi'_2)$, and $\psi'_1 \subseteq \psi'_2$;

(3) $l_1; l_2.\Lleftarrow(\pi_{1+2}, \psi_2, y_{1+2}) = (d\pi_1 \cup d\pi_2, \psi'_2)$, such that $\pi_{1+2} = \pi_1 \otimes \pi_2 \wedge \pi_1 \in \Pi_1 \wedge \pi_2 \in \Pi_2$, $y_{1+2} = y_1 \otimes y_2 \wedge y_1 \in Y_1 \wedge y_2 \in Y_2$, $\psi_1 \subseteq \psi_2 \wedge \psi_1 \in \Pi_1$, $l_1.\Lleftarrow(\pi_1, \psi_1, y_1) = (d\pi_1, \psi'_1)$, $l_2.\Lleftarrow(\pi_2, (\psi_2 - \psi_1) \uplus \psi'_1, y_2) = (d\pi_2, \psi'_2)$, $\psi'_1 \subseteq \psi'_2$, and $d\pi_1 \cup d\pi_2 \rhd \pi_{1+2}$ is defined.

Intuitively, sequential union is used when l_1 and l_2 convert disjoint systems (and models), and l_1 produces auxiliary and read-only information for l_2, which is stored in the correspondence data of l_1.

Example 6. Assume that l_{VM} converts between a set of virtual machines and a set of VM elements bijectively, where a VM element is the model representation of a virtual machine. A virtual machine is identified by its IP address, and a VM element is identified by a VM ID. A VM element does not record an IP address, because it is dynamically allocated when a virtual machine is created. Assume that l_{NW} bidirectionally converts a virtual network and a connection model that specifies the connections among VM elements. To configure the virtual network, l_{NW} must know the IP addresses of all virtual machines, which cannot be provided by the network model (since VM elements do not record IP addresses). In such a case, we perform l_{VM} first, and save the mapping between IP addresses and VM IDs in the correspondence data. Afterward, we perform l_{NW} that reads the mapping saved in the correspondence data of l_{VM} and converts the network connections. This process is abstracted by $l_{\mathrm{VM}}; l_{\mathrm{NW}}$.

Theorem 2. If l_1 and l_2 are system-model BX, then $l_1; l_2$ is also a system-model BX.

Proof. In both directions, the definition of $l_1; l_2$ ensures that l_2 never overwrites the correspondence data generated by l_1. Due to the fact that Π_1, Π_2, and Y_1, Y_2 are disjoint, similar to parallel union, it is not difficult to verify that $l_1; l_2$ preserves the well-behavedness of l_1 and l_2.

Sum ($\oplus$). For two sets X_1 and X_2, $X_1 \oplus X_2 \equiv \{x | x \in X_1 \vee x \in X_2\}$. When Π_1, Π_2, and Y_1, Y_2 are disjoint, given two system-model BXs $l_1 : \partial\Pi_1 \xleftrightarrow{\Psi_1} Y_1$ and $l_2 : \partial\Pi_2 \xleftrightarrow{\Psi_2} Y_2$, $l_1 \oplus l_2 : \partial(\Pi_1 \oplus \Pi_2) \xleftrightarrow{\Psi_1 \oplus \Psi_2} Y_1 \oplus Y_2$ is defined as follows:

(1) $l_1 \oplus l_2.K = \{(\pi, \psi, y) | (\pi, \psi, y) \in l_1.K \vee (\pi, \psi, y) \in l_2.K\}$;

(2) If $\pi \in \Pi_j$ and $\psi \in \Psi_j$, then $l_1 \oplus l_2.\Rrightarrow(\pi, \psi) = l_j.\Rrightarrow(\pi, \psi)$, $j = 1, 2$;

(3) If $\pi \in \Pi_j$ and $\psi \notin \Psi_j$, then $l_1 \oplus l_2.\Rrightarrow(\pi, \psi) = l_j.\Rrightarrow(\pi, \epsilon_{\Psi_j})$, $j = 1, 2$;

(4) If $\pi \in \Pi_j$, $\psi \in \Psi_j$, and $y \in Y_j$, then $l_1 \oplus l_2.\Lleftarrow(\pi, \psi, y) = l_j.\Lleftarrow(\pi, \psi, y)$, $j = 1, 2$;

(5) If $\pi \in \Pi_j$, $\psi \notin \Psi_j$, and $y \in Y_j$, then $l_1 \oplus l_2.\Lleftarrow(\pi, \psi, y) = l_j.\Lleftarrow(\pi, \epsilon_{\Psi_j}, y)$, $j = 1, 2$;

(6) If $\pi \notin \Pi_j$, $\psi \in \Psi_j$, and $y \in Y_j$, then $l_1 \oplus l_2.\Lleftarrow(\pi, \psi, y) = (dj \cup \{\text{edits that destroy } \pi\}, \psi')$, such that $(dj, \psi') = l_j.\Lleftarrow(\epsilon_{\Pi_j}, \psi, y)$ $(j = 1, 2)$, where the edits that destroy π must result in an empty system;

(7) If $\pi \notin \Pi_j$, $\psi \notin \Psi_j$, and $y \in Y_j$, then $l_1 \oplus l_2.\Lleftarrow(\pi, \psi, y) = (dj \cup \{\text{edits that destroy } \pi\}, \psi')$, such that $(dj, \psi') = l_j.\Lleftarrow(\epsilon_{\Pi_j}, \epsilon_{\Psi_j}, y)$ $(j = 1, 2)$, where the edits that destroy π must result in an empty system.

Intuitively, in forward direction, $l_1 \oplus l_2$ calls l_j if the given source π belongs to Π_j $(j = 1, 2)$. In backward direction, $l_1 \oplus l_2$ calls l_j if the given view y belongs to Y_j $(j = 1, 2)$.

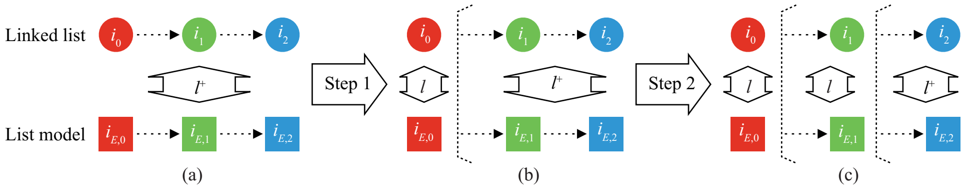


Figure 4 Illustration of recursion. (a) A linked list and a list model are to be synchronized with l^+; (b) in step 1, the heads are synchronized with l and the tails are handled by l^+; (c) in step 2, the tails of step 1 are recursively handled as is done in step 1.

Theorem 3. If l_1 and l_2 are system-model BX, then $l_1 \oplus l_2$ is also a system-model BX.

Proof. Note that the core behavior of $l_1 \oplus l_2$ is actually achieved by l_1 and l_2, which are well-behaved system-model BXs. Then, it is trivial to prove $l_1 \oplus l_2$ is well behaved by straightforwardly checking whether $l_1 \oplus l_2$ satisfies Eqs. (4) and (5).

Recursion ($^+$). Given a system-model BX $l : \partial\Pi \overset{\Psi}{\longleftrightarrow} Y$, assume that there are two partition functions $\mathrm{part}^l_{\Rrightarrow} : \Pi \times \Psi \rightarrow (\Pi \times \Pi) \times (\Psi \times \Psi)$ and $\mathrm{part}^l_{\Lleftarrow} : \Pi \times \Psi \times Y \rightarrow (\Pi \times \Pi) \times (\Psi \times \Psi) \times (Y \times Y)$ that satisfy the following conditions:

- $\mathrm{part}^l_{\Rrightarrow}$ partitions π and ψ, i.e., $\mathrm{part}^l_{\Rrightarrow}(\pi,\psi) = ((\pi_a,\pi_b),(\psi_a,\psi_b))$, such that $\pi_a \cap \pi_b = \emptyset \wedge \pi_a \cup \pi_b = \pi$, $\psi_a \cap \psi_b = \emptyset \wedge \psi_a \cup \psi_b = \psi$, and $\mathrm{part}^l_{\Rrightarrow}(\pi_a,\psi) = ((\pi_a,\epsilon),(\psi_a,\psi_b))$; moreover, $\mathrm{part}^l_{\Rrightarrow}$ partitions π irrespective of ψ, i.e., $\forall\psi'(\mathrm{part}^l_{\Rrightarrow}(\pi,\psi) = ((\pi_a,\pi_b),(\psi_a,\psi_b)) \wedge \mathrm{part}^l_{\Rrightarrow}(\pi,\psi') = ((\pi_a,\pi_b),(\psi'_a,\psi'_b)))$;
- $\mathrm{part}^l_{\Lleftarrow}$ partitions π, ψ, and y, i.e., $\mathrm{part}^l_{\Lleftarrow}(\pi,\psi,y) = ((\pi_a,\pi_b),(\psi_a,\psi_b),(y_a,y_b))$, such that $\pi_a \cap \pi_b = \emptyset \wedge \pi_a \cup \pi_b = \pi$, $\psi_a \cap \psi_b = \emptyset \wedge \psi_a \cup \psi_b = \psi$, $y_a \cap y_b = \emptyset \wedge y_a \cup y_b = y$, and $\mathrm{part}^l_{\Lleftarrow}(\pi,\psi,y_a) = ((\pi_a,\pi_b),(\psi_a,\psi_b),(y_a,\epsilon))$; moreover, $\mathrm{part}^l_{\Lleftarrow}$ partitions y irrespective of π and ψ, i.e., $\forall\pi',\psi'\big(\mathrm{part}^l_{\Lleftarrow}(\pi,\psi,y) = ((\pi_a,\pi_b),(\psi_a,\psi_b),(y_a,y_b)) \wedge \mathrm{part}^l_{\Lleftarrow}(\pi',\psi',y) = ((\pi_a,\pi_b),(\psi'_a,\psi'_b),(y'_a,y'_b))\big)$;
- If $(\pi,\psi,y) \in l.K$, then $\mathrm{part}^l_{\Lleftarrow}(\pi,\psi,y) = ((\pi,\epsilon),(\psi,\epsilon),(y,\epsilon))$ and $\mathrm{part}^l_{\Rrightarrow}(\pi,\psi) = ((\pi,\epsilon),(\psi,\epsilon))$, i.e., π, ψ, and y cannot be divided by $\mathrm{part}^l_{\Rrightarrow}$ and $\mathrm{part}^l_{\Lleftarrow}$ when (π,ψ,y) is a consistent tuple;
- If $(\pi,\psi,y) \in l.K$, $\mathrm{part}_{\Lleftarrow}(\pi,\psi,y) = ((\pi,\epsilon),(\psi,\epsilon),(y,\epsilon))$, then for any π',ψ', when $\pi \subseteq \pi'$ and $\psi \subseteq \psi'$, $\mathrm{part}^l_{\Lleftarrow}(\pi',\psi',y) = ((\pi,\pi'_b),(\psi,\psi'_b),(y,\epsilon))$, i.e., $\mathrm{part}^l_{\Lleftarrow}$ only extracts necessary data from π and ψ.

We can define recursion of system-model BX $l^+ : \partial\Pi \overset{\Psi}{\longleftrightarrow} Y$ as follows:

(1) $l^+.K = \{(\pi,\psi,y) | (\pi_a,\psi_a,y_a) \in l.K \wedge ((\pi_b = \epsilon \wedge y_b = \epsilon) \vee (\pi_b,\psi,y_b) \in l^+.K)\}$, when $\mathrm{part}^l_{\Lleftarrow}(\pi,\psi,y) = ((\pi_a,\pi_b),(\psi_a,\psi_b),(y_a,y_b))\}$;

(2) $l^+.\Rrightarrow(\pi,\psi) = (y,\psi'_a \cup \psi_b)$, when $((\pi,\epsilon),(\psi_a,\psi_b)) = \mathrm{part}^l_{\Rrightarrow}(\pi,\psi)$, $(y,\psi'_a) = l.\Rrightarrow(\pi,\psi_a)$;

(3) $l^+.\Rrightarrow(\pi,\psi) = (y_a \cup y_b,\psi')$, when $((\pi_a,\pi_b),(\psi_a,\psi_b)) = \mathrm{part}^l_{\Rrightarrow}(\pi,\psi)$, $(y_a,\psi'_a) = l.\Rrightarrow(\pi_a,\psi_a)$, $(y_b,\psi') = l^+.\Rrightarrow(\pi_b,\psi_b \uplus \psi'_a)$, $\psi'_a \subseteq \psi'$, and $\mathrm{part}^l_{\Lleftarrow}(\pi,\psi',y_a \cup y_b) = ((\pi_a,\pi_b),(\psi'_a,\psi'_b),(y_a,y_b))$;

(4) $l^+.\Lleftarrow(\pi,\psi,y) = (d\pi,\psi'_a \uplus \psi_b)$, when $\mathrm{part}^l_{\Lleftarrow}(\pi,\psi,y) = ((\pi,\epsilon),(\psi_a,\psi_b),(y,\epsilon))$, $(d\pi,\psi'_a) = l.\Lleftarrow(\pi,\psi_a,y)$;

(5) $l^+.\Lleftarrow(\pi,\psi,y) = (d\pi_a \cup d\pi_b,\psi')$, when $\mathrm{part}^l_{\Lleftarrow}(\pi,\psi,y) = ((\pi_a,\pi_b),(\psi_a,\psi_b),(y_a,y_b))$, $(d\pi_a,\psi'_a) = l.\Lleftarrow(\pi_a,\psi_a,y_a)$, $(d\pi_b,\psi') = l^+.\Lleftarrow(\pi_b,\psi_b \uplus \psi'_a,y_b)$, $\psi'_a \subseteq \psi'$, $d\pi_a \cup d\pi_b \rhd \pi$ is defined, and π_a and π_b are disjoint about $d\pi_a \cup d\pi_b \rhd \pi$.

Intuitively, recursion performs the synchronization recursively. Each iteration handles part of the system/model that is disjoint from the remainder. Sequential union is a special case of recursion.

Example 7. Assume that we want to synchronize in-memory linked lists Π with list models Y. Π_1 denotes minimal lists that contain exactly one circle and its incoming arrow (optional), and Y_1 denotes minimal list models that contain exactly one box and its incoming arrow (optional). As shown in Figure 4(a), a linked list is represented as a chain of colored circles and a list model is represented as a chain of colored boxes. If there is l that can convert Π_1 and Y_1 bidirectionally, then l^+ can synchronize the entire linked list with the list model. For instance, as shown in Figure 4(b), the linked list and the list model are split into list heads and list tails. We apply l to handle the head nodes and convert the tails with l^+. The correspondence data of l, which contains the mapping between the heads, e.g., $(i_0, i_{E,0})$, will be passed to the later call to l^+. Finally, as shown in Figure 4(c), the list tails are recursively handled.

Theorem 4. If l is a system-model BX, then l^+ is also a system-model BX.

Proof. The proof is similar to that of sequential union. We only discuss the second branch of backward transformation as follows. Because $\mathrm{part}^l_{\Lleftarrow}(\pi,\psi,y) = ((\pi_a,\pi_b),(\psi_a,\psi_b),(y_a,y_b))$, we have $\mathrm{part}^l_{\Lleftarrow}(d\pi_a \cup d\pi_b \rhd \odot\pi,\psi',y) = ((\pi'_a,\pi'_b),(\psi''_a,\psi''_b),(y_a,y_b))$, and then $\mathrm{part}^l_{\Lleftarrow}(d\pi_a \cup d\pi_b \rhd \odot\pi,\psi',y_a) = ((\pi'_a,\pi'_b),(\psi''_a,\psi''_b),(y_a,\epsilon))$. Since $(d\pi_a \rhd \odot\pi_a,\psi'_a,y_a) \in l.K$, we must have $\mathrm{part}^l_{\Lleftarrow}(d\pi_a \cup d\pi_b \rhd \odot\pi,\psi',y) = ((d\pi_a \rhd \odot\pi_a, d\pi_b \rhd$

$\odot\pi_b), (\psi'_a,\psi'_b), (y_a,y_b))$. So the round-trip properties can be ensured.

Edit chain ($\gg$). Assume that we want to bidirectionally convert a certain type of systems Π_1 and a model type Y, and we already have a system-model BX $l : \partial\Pi_2 \overset{\Psi}{\longleftrightarrow} Y$. If there is an edit lens [6] $l_S : \partial\Pi_1 \overset{C}{\longleftrightarrow} \partial\Pi_2$6) between Π_1 and Π_2 that converts edits for Π_1 and Π_2 bidirectionally, then it is possible to obtain an edit chain $l \gg l_S : \partial\Pi_1 \xleftrightarrow{\Pi_2\times C\times\Psi} Y$7) that is defined as follows:

(1) $l \gg l_S.K = \{(\pi_1, (\pi_2, c, \psi), y) | \pi_2 \in \Pi_2 \wedge (\pi_1, c, \pi_2) \in l_S.K \wedge (\pi_2, \psi, y) \in l.K\}$;

(2) If $(\pi_1, c, \pi_2) \in l_S.K$, $l \gg l_S.\Rrightarrow(\pi'_1, (\pi_2, c, \psi)) = (y, (\pi'_2, c', \psi'))$, where es_1 is the edit sequence that can turn π_1 into $\pi'_1$8), $(\mathrm{es}_2, c') = l_S.\Rrightarrow(\mathrm{es}_1, c)$, $\pi'_2 = \mathrm{es}_2 \rhd\odot\pi_2$, and $(y, \psi') = l.\Rrightarrow(\pi'_2, \psi)$;

(3) If $(\pi_1, c, \pi_2) \in l_S.K$, $l \gg l_S.\Lleftarrow(\pi_1, (\pi_2, c, \psi), y) = (d\pi_1, (\pi'_2, c', \psi'))$, where $(d\pi_2, \psi') = l.\Lleftarrow(\pi_2, \psi, y)$, $\pi'_2 = d\pi_2 \rhd\odot\pi_2$, and $(d\pi_1, c') = l_S.\Lleftarrow(d\pi_2, c)$.

Example 8. Edit chain can be used when it is difficult to define a system-model BX between Π_1 and Y straightforwardly. Assume that we want to synchronize a local model with a remote system (i.e., Π_1) that is inaccessible directly. We can define an interface system (i.e., Π_2) that exposes some APIs for remote clients, and deploy an edit lens between Π_1 and Π_2 on the remote computer. Afterward, we define a system-model BX between Π_2 and Y locally. Edit chain ensures that the combination of the remote edit lens and the local system-model BX achieves the synchronization between Π_1 and Y.

Theorem 5. When l and l_S are well behaved, $l \gg l_S$ is also well behaved.

Proof. It is trivial to verify that the definition of $l \gg l_S$ preserves the well-behavedness of l and l_S.

Set mapping ($\{\cdot\}$). In some cases, a system $\pi \in \Pi$ can be viewed as a set of disjoint sub-systems $\pi_1, \ldots, \pi_n \in \Pi'$ (and the same to a model). We say $\Pi = \{\Pi'\}$ and term it a set module. In concept, a set module should contain three edits, i.e., insert, remove, and modify, which are used to add new values into a set, to delete existing values, and to modify existing values. However, we do not define these edits here, and only use them as placeholders. They should be concretized in different contexts (see Section 5).

When both a system and a model can be viewed as sets, we can achieve the bidirectional conversion between them by synchronizing the sub-systems with the sub-models. Given a system-model BX $l : \partial\Pi \overset{\Psi}{\longleftrightarrow} Y$, we can lift l to a set mapping $\{l\}$ when there is a pairing function $\mathrm{pair}^{\{l\}} : \{\Pi\} \times \Psi \times \{Y\} \to 2^{\Pi\times\Psi\times Y}$ that satisfies the following conditions.

- For any $\pi \in \pi_{\mathrm{set}}$, $\mathrm{pair}^{\{l\}}$ uses π exactly once, i.e., $\{\pi | (\pi, \psi, y) \in \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}}) \wedge \pi \neq \epsilon\} = \pi_{\mathrm{set}}$ and $|\{(\pi, \psi, y) | (\pi, \psi, y) \in \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}}) \wedge \pi \neq \epsilon\}| = |\pi_{\mathrm{set}}|$;
- For any $y \in y_{\mathrm{set}}$, $\mathrm{pair}^{\{l\}}$ uses y exactly once, i.e., $\{y | (\pi, \psi, y) \in \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}}) \wedge y \neq \epsilon\} = y_{\mathrm{set}}$, $|\{(\pi, \psi, y) | (\pi, \psi, y) \in \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}}) \wedge y \neq \epsilon\}| = |y_{\mathrm{set}}|$;
- $\mathrm{pair}^{\{l\}}$ does not return meaningless tuples, i.e., for any $(\pi, \psi, y) \in \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}})$, $\pi \neq \epsilon \vee y \neq \epsilon$;
- For any $(\_, \psi, \_) \in \mathrm{pair}^{\{l\}}(\_, \psi_{\mathrm{set}}, \_)$, $\psi \subseteq \psi_{\mathrm{set}}$;
- If $(\pi, \psi, y) \in l.K$ and $\pi \in \pi_{\mathrm{set}} \wedge \psi \subseteq \psi_{\mathrm{set}} \wedge y \in y_{\mathrm{set}}$, then $(\pi, \psi, y) \in \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}})$.

With such a pairing function, $\{l\} : \partial\{\Pi\} \overset{\Psi}{\longleftrightarrow} \{Y\}$ is defined as follows:

(1) $\{l\}.K = \{(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}}) | \forall(\pi, \psi, y)((\pi, \psi, y) \in \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}}) \Rightarrow (\pi, \psi, y) \in l.K)\}$;

(2) If $\{(\pi_1, \psi_1, \epsilon), (\pi_2, \psi_2, \epsilon), \ldots, (\pi_n, \psi_n, \epsilon)\} = \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, \epsilon)$, $(y_i, \psi'_i) = l.\Rrightarrow(\pi_i, \psi_i)$ $(i = 1, \ldots, n)$, $i \neq j \Rightarrow y_i \cap y_j = \emptyset$, $\biguplus_{i=1}^n \psi'_i \downarrow$, then $\{l\}.\Rrightarrow(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, \epsilon) = (\{y_1, y_2, \ldots, y_n\}, \biguplus_{i=1}^n \psi'_i)$;

(3) If $\{(\pi_1, \psi_1, y_1), (\pi_2, \psi_2, y_2), \ldots, (\pi_n, \psi_n, y_n)\} = \mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}})$, $(d\pi_i, \psi'_i) = l.\Lleftarrow(\pi_i, \psi_i, y_i)$ $(i = 1, \ldots, n)$, then $d\pi'_i$ is defined as follows:

(a) If $\pi_i = \epsilon$, then $d\pi'_i = d\pi_i \cup \{$insert the system $d\pi_i \rhd\odot\epsilon$ in the set$\}$;

(b) If $y_i = \epsilon$, then $d\pi'_i = d\pi_i \cup \{$remove the system π_i from the set$\}$;

(c) If $d\pi_i \neq \emptyset$, then $d\pi'_i = d\pi_i \cup \{$modify the set to replace π_i with $d\pi_i \rhd\odot\pi_i\}$;

(d) Otherwise, $d\pi'_i = d\pi_i$, and in general, we expect $d\pi_i = \emptyset$.

If $i \neq j \Rightarrow \pi_i$ and π_j are disjoint about both $d\pi'_i$ and $d\pi'_j$, $\biguplus_{i=1}^n \psi'_i \downarrow$, and $\bigcup_{i=1}^n d\pi'_i \rhd \pi_{\mathrm{set}}$ is defined, then $\{l\}.\Lleftarrow(\pi_{\mathrm{set}}, \psi_{\mathrm{set}}, y_{\mathrm{set}}) = (\bigcup_{i=1}^n d\pi'_i, \biguplus_{i=1}^n \psi'_i)$.

Intuitively, set mapping ensures a bijective mapping between the system set and the model set by using l to synchronize paired sub-systems and sub-models.

6) For edit lens $l_S : \partial\Pi_1 \overset{C}{\longleftrightarrow} \partial\Pi_2$, C is called complement, and $l_S.K$ is also the consistency relation such that $l_S.K \subseteq \Pi_1 \times C \times \Pi_2$. Please refer to [6] for more information.

7) Here, we slightly extend the definition of correspondence to allow it carry extra data. If we want to turn $\Pi_2 \times C \times \Psi$ into a partial bijective mapping between Π_1 and Y, then we need an injective function from Π_2 to Π_1.

8) According to [6], es_1 can be obtained by elaborately differencing π'_1 and π_1.

Theorem 6. $\{l\} : \partial\{\Pi\} \overset{\Psi}{\longleftrightarrow} \{Y\}$ is a well-behaved system-model BX.

Proof. In forward direction, because every $(\pi_i, \psi'_i, y_i) \in l.K$ and $\mathrm{pair}^{\{l\}}(\pi_{\mathrm{set}}, \biguplus_{i=1}^{n} \psi'_i, \{y_1, y_2, \ldots, y_n\}) = \{(\pi_i, \psi'_i, y_i)|i = 1, \ldots, n\}$, we must have $(\pi_{\mathrm{set}}, \biguplus_{i=1}^{n} \psi'_i, \{y_1, y_2, \ldots, y_n\}) \in \{l\}.K$. In backward direction, $\{l\}$ first pairs π_{set} with y_{set} and obtains (π_i, ψ_i, y_i) $(i = 1, \ldots, n)$. Afterward, $\{l\}$ calls $l.\Lleftarrow$ for every (π_i, ψ_i, y_i) to compute $d\pi_i$. $\{l\}$ may also append some set edits to $d\pi_i$ and obtains $d\pi'_i$. For instance, when $\pi_i = \epsilon$, $\{l\}$ generates an extra edit that intends to add $d\pi_i \rhd\odot\epsilon$ to the system set. In this way, $d\pi'_i$ not only changes π_i but also ensures that the changed sub-system is added/removed/modified in the system set. When $\bigcup_{i=1}^{n} d\pi'_i$ is applied to π_{set} and the resulting system is π'_{set}, we must have (1) $y_i = \epsilon$ (and $d\pi'_i$ removes π_i from the set) or $(d\pi_i \rhd \odot\pi_i, \psi'_i, y_i) \in l.K$, and (2) $\mathrm{pair}^{\{l\}}(\pi'_{\mathrm{set}}, \biguplus_{i=1}^{n} \psi'_i, y_{\mathrm{set}}) = \{(d\pi_i \rhd\odot\pi_i, \psi'_i, y_i)|i = 1, \ldots, n \wedge y_i \neq \epsilon\}$. As a result, $(\pi'_{\mathrm{set}}, \biguplus_{i=1}^{n} \psi'_i, y_{\mathrm{set}}) \in \{l\}.K$.

Remark 10. If a system π_{set} (a model y_{set}) is not a real set, to apply set mapping, there must be a way that can divide π_{set} (y_{set}) into a set and merge the sub-systems (sub-models) into π_{set} (y_{set}) again.

List mapping ($[\cdot]$). Consider the case that a system π_{lst} (a model y_{lst}) is actually a list $[\pi_1, \pi_2, \ldots, \pi_n]$ of sub-systems (a list of $[y_1, y_2, \ldots, y_m]$ of sub-models), where $\pi_{\mathrm{lst}} \in [\Pi]$ and $\pi_i \in \Pi$ ($y_{\mathrm{lst}} \in [Y]$ and $y_i \in Y$). Then, $[\Pi]$ $([Y])$ is a list system module (a list model type). In concept, a list module should contain three edits, i.e., insert, remove, and modify, which are used to add new values into a list, to delete existing values, and to modify existing values. Similar to set mapping, we still use those edits as placeholders.

Given a system-model BX $l : \partial\Pi \overset{\Psi}{\longleftrightarrow} Y$, we need a pairing function $\mathrm{pair}^{[l]} : [\Pi] \times \{\Psi\} \times [Y] \to [\Pi \times \Psi \times Y]$ satisfying the following conditions.

- For any $\pi \in \pi_{\mathrm{lst}}$, $\mathrm{pair}^{[l]}$ uses π exactly once, i.e., $\{\pi|(\pi, \psi, y) \in \mathrm{pair}^{[l]}(\pi_{\mathrm{lst}}, \psi_{\mathrm{set}}, y_{\mathrm{lst}}) \wedge \pi \neq \epsilon\} = \pi_{\mathrm{lst}}$ and $|\{(\pi, \psi, y)|(\pi, \psi, y) \in \mathrm{pair}^{[l]}(\pi_{\mathrm{lst}}, \psi_{\mathrm{set}}, y_{\mathrm{lst}}) \wedge \pi \neq \epsilon\}| = |\pi_{\mathrm{lst}}|$, where we cast π_{lst} to a set;
- For any $y \in y_{\mathrm{lst}}$, $\mathrm{pair}^{[l]}$ uses y exactly once, i.e., $\{y|(\pi, \psi, y) \in \mathrm{pair}^{[l]}(\pi_{\mathrm{lst}}, \psi_{\mathrm{set}}, y_{\mathrm{lst}}) \wedge y \neq \epsilon\} = y_{\mathrm{lst}}$, $|\{(\pi, \psi, y)|(\pi, \psi, y) \in \mathrm{pair}^{[l]}(\pi_{\mathrm{lst}}, \psi_{\mathrm{set}}, y_{\mathrm{lst}}) \wedge y \neq \epsilon\}| = |y_{\mathrm{lst}}|$, where we cast y_{lst} to a set;
- $\mathrm{pair}^{[l]}$ does not return meaningless tuples, i.e., for any $(\pi, \psi, y) \in \mathrm{pair}^{[l]}(\pi_{\mathrm{lst}}, \psi_{\mathrm{set}}, y_{\mathrm{lst}})$, $\pi \neq \epsilon \vee y \neq \epsilon$;
- For any $(\_, \psi, \_) \in \mathrm{pair}^{[l]}(\_, \psi_{\mathrm{set}}, \_)$, $\psi \subseteq \psi_{\mathrm{set}}$;
- Given i, if $\forall k(k < i \to (\pi_k, \_, y_k) \in \mathrm{pair}^{[l]}(\pi_{\mathrm{lst}}, \psi_{\mathrm{set}}, y_{\mathrm{lst}}))$ and there exists $\psi \subseteq \psi_{\mathrm{set}}$ such that $(\pi_i, \psi, y_i) \in l.K$, then $(\pi_i, \_, y_i) \in \mathrm{pair}^{[l]}(\pi_{\mathrm{lst}}, \psi_{\mathrm{set}}, y_{\mathrm{lst}})$; in short, if the objects and the elements occurring before i are mutually paired, and if π_i and y_i are consistent, then π_i and y_i must be also paired.

Now we define list mapping $[l] : \partial[\Pi] \overset{\{\Psi\}}{\longleftrightarrow} [Y]$ as follows.

(1) $[l].K = \{([\pi_1, \pi_2, \ldots, \pi_n], \psi_{\mathrm{set}}, [y_1, y_2, \ldots, y_n])|\mathrm{pair}^{[l]}([\pi_1, \pi_2, \ldots, \pi_n], \psi_{\mathrm{set}}, [y_1, y_2, \ldots, y_n]) = [(\pi_1, \psi_1, y_1), (\pi_2, \psi_2, y_2), \ldots, (\pi_n, \psi_n, y_n)] \wedge \forall i((\pi_i, \psi_i, y_i) \in l.K)\}$;

(2) If $\mathrm{pair}^{[l]}([\pi_1, \pi_2, \ldots, \pi_n], \psi_{\mathrm{set}}, \epsilon) = [(\pi_1, \psi_1, \epsilon), (\pi_2, \psi_2, \epsilon), \ldots, (\pi_n, \psi_n, \epsilon)]$, $(y_i, \psi'_i) = l.\Rrightarrow(\pi_i, \psi_i)$ $(i = 1, \ldots, n)$, $\biguplus_{i=1}^{n} \psi'_i \downarrow$, then $[l].\Rrightarrow([\pi_1, \pi_2, \ldots, \pi_n], \psi_{\mathrm{set}}) = ([y_1, y_2, \ldots, y_n], \biguplus_{i=1}^{n} \psi'_i)$;

(3) If $\mathrm{pair}^{[l]}([\pi_1, \pi_2, \ldots, \pi_n], \psi_{\mathrm{set}}, [y_1, y_2, \ldots, y_m]) = [(\pi_{i_1}, \psi_{j_1}, y_{k_1}), (\pi_{i_2}, \psi_{j_2}, y_{k_2}), \ldots, (\pi_{i_p}, \psi_{j_p}, y_{k_p})]$, supposing $(d\pi_{i_q}, \psi'_{j_q}) = l.\Lleftarrow(\pi_{i_q}, \psi_{j_q}, y_{k_q})$ $(q = 1, \ldots, p)$, then we compute $d\pi'_{j_q}$ as follows.

(a) If $\pi_{i_q} = \epsilon$, then $d\pi'_{i_q} = d\pi_{i_q} \cup \{$insert the system $d\pi_{i_q} \rhd\odot\epsilon$ at k_q in the list$\}$;

(b) If $y_{k_q} = \epsilon$, then $d\pi'_{i_q} = d\pi_{i_q} \cup \{$remove the system π_{i_q} at i_q from the list$\}$;

(c) Otherwise, $d\pi'_{i_q} = d\pi_{i_q} \cup \{$modify π_{i_q} at i_q in the list with $d\pi_{i_q} \rhd\odot\pi_{i_q}$ and move it to $k_q\}$.

If $u \neq v \Rightarrow \pi_{i_u}$ and π_{i_v} are disjoint about both $d\pi'_{i_u}$ and $d\pi'_{i_v}$, $\biguplus_{u=1}^{p} \psi'_{j_u} \downarrow$, and $\bigcup_{u=1}^{p} d\pi'_{i_u} \rhd \pi_{\mathrm{lst}}$ is defined, then $\{l\}.\Lleftarrow(\pi_{\mathrm{lst}}, \psi_{\mathrm{set}}, y_{\mathrm{lst}}) = (\bigcup_{u=1}^{p} d\pi'_{i_u}, \biguplus_{u=1}^{p} \psi'_{j_u})$.

Intuitively, list mapping ensures (1) a bijective mapping between the system list and the model list with l and (2) the ith value (i.e., π_i) in the system list must be paired with the ith value (i.e., y_i) in the model list when a consistent state is reached.

Example 9. Assume that l converts π_i with y_i. Given a system list $[\pi_1, \pi_2, \pi_3]$, $[l].\Rrightarrow$ converts every π_i in the list into y_i $(i = 1, 2, 3)$, and then creates a resulting list $[y_1, y_2, y_3]$ based on the value order of the system list. In backward direction, given a system list $[\pi_1, \pi_2, \pi_3]$ and a mode list $[y_3, y_4, y_2]$, $[l].\Lleftarrow$ first aligns the two lists, and knows that y_2 and y_3 are paired with π_2 and π_3, and π_1 and y_4 are unpaired. Afterward, for π_1, $[l].\Lleftarrow$ generates a list edit remove π_1 at 1 because π_1 cannot be paired with any value in the model list; for π_2 and y_2, $[l].\Lleftarrow$ generates a list edit modify the list by moving π_2 from 2 to 3 because in the model list, y_2 appears at position 3; for π_3 and y_3, $[l].\Lleftarrow$ generates a list edit modify by moving π_3 from 3 to 1 because in the model list, y_3 is the first value; for y_4, $[l].\Lleftarrow$ generates a list edit insert π_4 at 2, where $l.\Lleftarrow$ must generate the edits that create π_4.

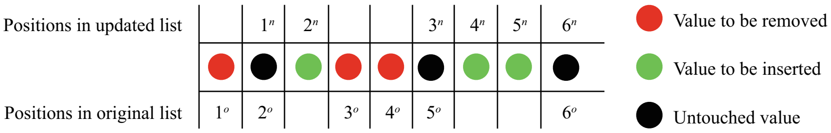


Figure 5 Position conversion.

Before we consider the well-behavedness of list mapping, an important issue must be discussed first. Recall that when we generate list edits (i.e., insert, remove, and modify), we specify the positions at which the values in the list are to be removed, inserted, and changed and moved. However, all those positions are only valid for either the original lists or the updated lists.

For example, consider the original system list $[\pi_1, \pi_2, \pi_3]$ and two edits remove the value at 1^o and insert π_4 at 2^n, where we use i^o and i^n to denote a position i in the original list and the updated list, respectively. The expected result is $[\pi_2, \pi_4, \pi_3]$, i.e., the first value in the original list is removed and π_4 must present at position 2 in the final updated result. If we do the removal first (and we get $[\pi_2, \pi_3]$) and the insertion afterward (and we get $[\pi_2, \pi_4, \pi_3]$), then we obtain the expected result. However, if we do the two edits in the reverse order, then we will get an incorrect result $[\pi_4, \pi_2, \pi_3]$—value π_4 does not appear at position 2. The major cause is that those positions in the edit operations are state-based and they should be dynamically refreshed when a list edit is applied (or not applied). Consider our example again. If the removal of the first value is not done, then we should not insert π_4 at 2 but at 3, so that when the first value is removed in the future, π_4 will become the second value in the final result.

To gain more insights, please consider Figure 5, which shows a list of colored circles, where the red circles are to be removed from the original list, green circles are to be inserted into the updated list, and black circles are preserved in both original and updated list. Below the list, original positions are presented (green circles do not have original positions), and above the list, updated positions are presented (red circles do not have updated positions). For an updated position, e.g., 5^n, obviously, if all the red circles before 5^n are removed from the list and all green circles before 5^n are inserted, then the correct position of 5^n is exactly 5. If there is one red circle that is not removed, then we must increase the position by 1; if there is one green circle that is not inserted, then we must decrease the position by 1. For an original position, e.g., 4^o, if any red circle before 4^o is removed, then we must decrease it by 1; if any green circle before 4^o is inserted, then we must increase it by 1.

We assume that green circles must be inserted after black circles and before red circles as illustrated in Figure 5. Hence, the key of the calculation is to know the original position of the nearest black circle that is in front of a green circle, and we use a function nearestBlk to return the answer. For example, $\text{nearestBlk}(2^n) = 2^o$ and $\text{nearestBlk}(4^n) = \text{nearestBlk}(5^n) = 5^o$. This function helps us to compare the position of a red circle and that a green circle. For instance, because $\text{nearestBlk}(2^n) = 2^o$, we know that the red circle at 1^o is before 2^n, and the red circles at 3^n and 4^n are after 2^n.

Recall the meaning of the updated position i^n. It implies that in the updated list, there are $i-1$ values presented before i^n. The $i-1$ values may be green circles or black circles. That means if we subtract the number of green circles before i^n from $i-1$, then we obtain the number of black circles. In the calculation, insert at j^n is viewed as a green circle; remove at k^o is viewed as a red circle; and modify at k^o and move it to j^n are viewed as a red and a green circles. Function nearestBlk is defined as follows:

$$\begin{aligned}&\text{nearestBlk}(i^n) \equiv \min\ t^o, \text{ such that } \big|\{k^o | k \leqslant t \wedge (\nexists\text{remove at } k^o) \wedge (\nexists\text{modify at } k^o)\}\big| = \text{blkBefore}(i^n),\\ &\text{where blkBefore}(i^n) \equiv i - 1 - \big|\{\text{insert at } j^n | j < i\} \cup \{\text{modify and move the value to } j^n | j^n < i^n\}\big|.\end{aligned}$$

Now we are able to compute the correct position, denoted as function cur(), of any original position k^o and any updated position j^n given a set of list edits at runtime as follows:

$$\begin{aligned}&\text{cur}(j^n) = j, \qquad\qquad \text{/* required position in updated list */}\\ &\qquad\qquad \text{/* subtract the number of green circles that have not been inserted into the list */}\\ &\quad -\big|\{e | i < j \wedge (e = \text{insert at } i^n \vee e = \text{modify and move to } i^n) \wedge e \text{ is not performed}\}\big|,\\ &\qquad\qquad \text{/* add the number of red circles that have not been removed from the list */}\\ &\quad +\big|\{e | i < \text{NB} \wedge (e = \text{remove at } i^o \vee (e = \text{modify at } i^o \text{ and move to } p^n \wedge p \neq j)) \wedge e \text{ is not performed}\}\big|,\end{aligned}$$

where NB = nearestBlk(j^n). And, for k^o, we also have the following calculation:

$$\begin{aligned}
&\mathrm{cur}(k^o) = k, \qquad\qquad \text{/* required position in original list */}\\
&\qquad\qquad \text{/* subtract the number of red circles that have been removed from the list */}\\
&-\big|\{e|i < k \wedge (e = \text{remove at } i^n \vee e = \text{modify at } i^n) \wedge e \text{ is performed}\}\big|,\\
&\qquad\qquad \text{/* add the number of green circles that have been inserted into the list */}\\
&+\big|\{e|\mathrm{nearestBlk}(i^n) < k \wedge (e = \text{insert at } i^n \vee e = \text{modify and move to } i^n) \wedge e \text{ is performed}\}\big|.
\end{aligned}$$

In the rest of this paper, we assume that when a list edit is being performed, it automatically does the position conversion. As a result, the list edits can be performed in any reasonable order. And, when all of them are performed, the values to be deleted from the original list should not appear in the updated list, and the values to be inserted (and moved) must occur in the updated list at the specified positions.

Theorem 7. The list mapping $[l]$ is a well-behaved system-model BX.

Proof. In fact, list mapping is similar to set mapping, so it is analogous to show that list mapping is well-behaved if we ignore the position constraint, i.e., the i-th value in the system list must be paired with the i-th value in the model list. Regarding this position constraint, it is not difficult to find that when generating $d\pi'_{i_q}$, we add list edits to ensure that all the values occur in the correct positions in the updated list. The position conversion strategy discussed above ensures that we can always put a value at the right position at runtime, in whatever order the list edits are executed.

5 A generic system-model synchronizer

This section first demonstrates how to use the concept of system-model BX and its combinators proposed in this paper to develop a generic system-model synchronizer[9] in Subsections 5.1–5.3, and then concretizes a file system synchronizer from the generic one in Subsection 5.4.

The basic idea is as follows.

- For each class C in the system type, map C onto an EClass E in the model type by defining a system-model BX $\partial\mathrm{C} \overset{\Psi}{\longleftrightarrow} E$ (called a class synchronizer). Without the loss of generality, we require that the mapping between system classes and EClasses is bijective[10].
- For each structural feature f in the system type, map f onto an EStructuralFeature f_E by defining $\partial\mathsf{f} \overset{\Psi}{\longleftrightarrow} \mathsf{f}_E$ (called a feature synchronizer). Specifically, if f is a reference, then f_E must be an EReference; if f is an attribute, then f_E must be an EAttribute. The mapping between structural features and EStructuralFeatures must be bijective.
- In both directions, we always synchronize objects with model elements with the help of class synchronizers, and then handle links/slots and relationships.
- We never execute the system edits during the backward transformation because of the side effects and domain constraints. Instead, we collect all the system edits (see Section 4), plan a proper order (see Subsection 3.3), and finally apply them to the system when the entire backward transformation is completed. The round-trip properties are guaranteed by system-model BX.

5.1 Class synchronizer

A class synchronizer $l_{\mathrm{C}} : \partial\mathrm{C} \overset{\Psi}{\longleftrightarrow} E$ synchronizes objects in C and model elements E. The expected behavior of l_{C} is as follows. In forward direction, l_{C} generates a model element from the given object. In backward direction, l_{C} generates some object edits (i.e., $\mathrm{con}_{\mathrm{C}}$ and $\mathrm{des}_{\mathrm{C}}$) by differencing the given object and model element. Especially, if no object is provided, then we must generate a $\mathrm{con}_{\mathrm{C}}$ to create an object; if no model element is provided, then we must generate a $\mathrm{des}_{\mathrm{C}}$ to delete this object because it is redundant; if the given object and model element are inconsistent (e.g., their constructor parameters are inconsistent), then a new object will be created and the old object will be destroyed;

However, synchronizing an object i with an element i_E is more complicated than our expectation. Assume that i has a constructor parameter $i \xrightarrow{\mathsf{f}} v$ but i_E does not. Because we assume that no edits

[9] It is possible to build other generic synchronizers.

[10] In practice, C can be mapped onto multiple EClasses as long as all objects in C can be partitioned into multiple groups, each of which corresponds to only one EClass. In such a case, C can be viewed as a sum of multiple subtypes.

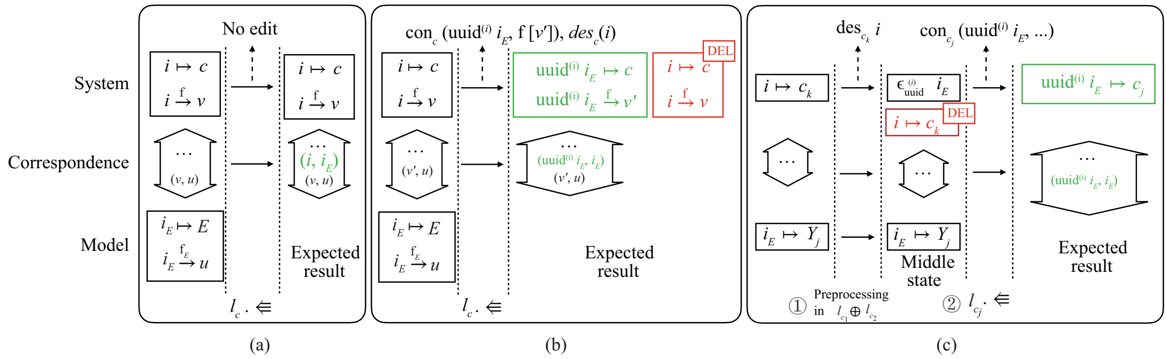

System
Correspondence
Model
No edit
$i \mapsto c$
$i \xrightarrow{f} v$
(v, u)
(i, i_E)
$i_E \mapsto E$
$i_E \xrightarrow{f_E} u$
Expected result
$l_c . \Lleftarrow$
(a)
$\mathrm{con}_c\,(\mathrm{uuid}^{(i)}\, i_E,\, f\,[v']),\, des_c(i)$
$\mathrm{uuid}^{(i)}\, i_E \mapsto c$
$\mathrm{uuid}^{(i)}\, i_E \xrightarrow{f} v'$
DEL
(v', u)
$(\mathrm{uuid}^{(i)}\, i_E, i_E)$
Expected result
$l_c . \Lleftarrow$
(b)
$\mathrm{des}_{c_k}\, i$
$\mathrm{con}_{c_j}\,(\mathrm{uuid}^{(i)}\, i_E, \ldots)$
$i \mapsto c_k$
$\epsilon^{(i)}_{\mathrm{uuid}}\; i_E$
$i_E \mapsto Y_j$
$\mathrm{uuid}^{(i)}\, i_E \mapsto c_j$
Middle state
Expected result
① Preprocessing in $l_{c_1} \oplus l_{c_2}$
② $l_{c_j} . \Lleftarrow$
(c)

Figure 6 Synchronization of objects. (a) Class synchronizer: branch 3(a); (b) class synchronizer: branch 3(b); (c) sum of class synchronizers: overloaded case.

are defined for constructor parameters (see Subsection 3.2), the only way of making the system and the model consistent is to create a new object i' that is consistent with i_E to take the place of i.

We assume that $\pi \in \textsc{c}$ consists of an object $i \mapsto \textsc{c}$ and all its constructor parameters. For simplicity, we still regard $i \mapsto \textsc{c}$ or i as an instance of $\textsc{c}$, while assuming that all its constructor parameters are automatically put into this instance. The case of E is analogous.

Definition 9 (Generic class synchronizer)**.** Given any class $\textsc{c}$ and EClass E, assume that their constructor parameters are $\mathsf{f}_1, \mathsf{f}_2, \ldots, \mathsf{f}_n$ and $\mathsf{f}_{E,1}, \mathsf{f}_{E,2}, \ldots, \mathsf{f}_{E,n}$ (f_j corresponds to $\mathsf{f}_{E,j}$), and there a function F_j for each f_j, such that $F_j(v) \equiv \psi(v)$ if f_j is a reference, or F_j is a bijective function[11] (by default, $F_j(v) = v$). Then, $l_{\textsc{c}} : \partial\textsc{c} \overset{\Psi}{\longleftrightarrow} E$ is defined as follows.

(1) $l_{\textsc{c}}.K = \{(i \mapsto \textsc{c}, \psi, i_E) | \psi(i) = i_E \wedge \forall j, v (i \xrightarrow{\mathsf{f}_j} v \Leftrightarrow F_j(v) \in i_E.\mathsf{f}_{E,j}) \wedge \psi \text{ is minimal}\}$; the definition of $l_{\textsc{c}}.K$ implies that $F_j(v) \neq \bot$;

(2) $l_{\textsc{c}}.\!\Rrightarrow\!(i \mapsto \textsc{c}, \psi) = (y, \psi')$, where i_E is a new element, $y \equiv \{i_E \mapsto E\} \cup \bigcup_{j=1}^{n} \bigcup_{k=1}^{m_j} \{i_E \xrightarrow{\mathsf{f}_{E,j}} F_j(v_{j,k})\}$, and $\psi' = \{(i, i_E)\} \cup \{(v, \psi(v)) | \exists i \xrightarrow{\mathsf{f}_j} v \wedge \psi(v) \neq \bot\}$; in short, $l.\!\Rrightarrow$ creates a new element i_E and sets up its constructor parameters to be consistent with $i \mapsto \textsc{c}$;

(3) $l_{\textsc{c}}.\!\Lleftarrow\!(i \mapsto \textsc{c}, \psi, i_E) =$

(a) $(\emptyset, \psi')$, if $(i \mapsto \textsc{c}, \psi', i_E) \in l_{\textsc{c}}.K \wedge \psi' \subseteq \psi[i \to i_E]$ (see Figure 6(a));

(b) $(\{\mathrm{des}_{\textsc{c}}\ i, \mathrm{creation}(i')\}, \psi')$, if $\nexists \psi''(\psi'' \subseteq \psi[i \to i_E] \wedge (i \mapsto \textsc{c}, \psi'', i_E) \in l_{\textsc{c}}.K)$, where $i' = \mathsf{uuid}^{(i)}\ i_E$ and $\psi' = \{(i', i_E)\} \cup \{(\psi^{-1}(u), u) | u \in i_E.\mathsf{f}_{E,j} \wedge \psi^{-1}(u) \neq \bot\}$, i.e., $l_{\textsc{c}}.\!\Lleftarrow$ creates a new object i' due to the change of constructor parameters (see Figure 6(b));

(4) $l_{\textsc{c}}.\!\Lleftarrow\!(\epsilon_{\mathsf{uuid}^{(i)}\ i_E}, \psi, i_E)$[12]$= (\{\mathrm{creation}(\mathsf{uuid}^{(i)}\ i_E)\}, \psi')$, where $\psi' = \{(\mathsf{uuid}^{(i)}\ i_E, i_E)\} \cup \{(\psi^{-1}(u), u) | u \in i_E.\mathsf{f}_{E,j} \wedge \psi^{-1}(u) \neq \bot\}$;

(5) $l_{\textsc{c}}.\!\Lleftarrow\!(\epsilon, \psi, i_E) = l_{\textsc{c}}.\!\Lleftarrow\!(\epsilon_{\mathsf{uuid}\ i_E}, \psi, i_E)$;

(6) $l_{\textsc{c}}.\!\Lleftarrow\!(i \mapsto \textsc{c}, \psi, \epsilon_E) = (\{\mathrm{des}_{\textsc{c}}\ i\}, \epsilon)$;

Particularly, $\mathrm{creation}(x)$ is $\mathrm{con}_{\textsc{c}}(x, \mathsf{f}_1[F_1^{-1}(u_{1,1}), \ldots, F_1^{-1}(u_{1,m_1})], \ldots, \mathsf{f}_n[F_n^{-1}(u_{n,1}), \ldots, F_n^{-1}(u_{n,m_n})])$, where we assume $u_{j,k} \in i_E.\mathsf{f}_{E,j}$.

Remark 11. It is not difficult to verify that this generic class synchronizer is a well behaved system-model BX that satisfies Definition 7 and Eqs. (4) and (5).

To concretize a class synchronizer, we must (1) identify $\textsc{c}$ and E as well as their constructor parameters $\mathsf{f}_1, \ldots, \mathsf{f}_n$ and $\mathsf{f}_{E,1}, \ldots, \mathsf{f}_{E,n}$, (2) provide bijective functions $F_1, F_2, \ldots, F_n$, and (3) define edits $\mathrm{con}_{\textsc{c}}$ and $\mathrm{des}_{\textsc{c}}$.

Sum of class synchronizers. In practice, there are many classes and EClasses. We can use the generic class synchronizer to develop many concrete synchronizers. Then, we can combine all concrete class synchronizers with sum $\oplus$. For two class synchronizers $l_{\textsc{c}_1} : \partial\textsc{c}_1 \overset{\Psi}{\longleftrightarrow} E_1$ and $l_{\textsc{c}_2} : \partial\textsc{c}_2 \overset{\Psi}{\longleftrightarrow} E_2$, $l_{\textsc{c}_1} \oplus l_{\textsc{c}_2}$ have to be specialized by refactoring the sixth and seventh branches of the sum as follows:

- If $\pi = i \mapsto \textsc{c}_k \wedge \textsc{c}_k \neq \textsc{c}_j$, $y = i_E \mapsto E_j$, and $(dj, \psi') = l_{\textsc{c}_j}.\!\Lleftarrow\!(\epsilon_{\mathsf{uuid}^{(i)}\ i_E}, \psi \backslash (i, \_), y)$, then $l_{\textsc{c}_1} \oplus l_{\textsc{c}_2}.\!\Lleftarrow$ $(\pi, \psi, y) = (dj \cup \{\mathrm{des}_{\textsc{c}_k}\ i\}, \psi')$ $(j = 1, 2)$; in such a case, the given model element $i_E \mapsto E_j$ is paired

[11] In fact, it can be a classical BX. However, for simplicity, we assume that it is a bijective function.

[12] $\epsilon_{\mathsf{uuid}^{(i)}\ i_E}$ (and $\epsilon_{\mathsf{uuid}\ i_E}$) denotes a special empty value that carries additional information of an object identifier.

with an object $i \mapsto \mathrm{C}_k$, but the type of i is inconsistent with E_j; hence, as illustrated in Figure 6(c), we generate $\mathrm{des}_{\mathrm{C}_j}\ i$ (step ①) and ask l_{C_j} to generate an invocation of $\mathrm{con}_{\mathrm{C}_j}$ (step ②).

Remark 12. Because C_1, C_2 and E_1, E_2 are disjoint, based on Theorem 3, $l_{\mathrm{C}_1} \oplus l_{\mathrm{C}_2}$ is also well behaved. In the rest of this section, $l_{\mathrm{C}} : \partial\mathrm{C} \overset{\Psi}{\longleftrightarrow} E$ may also represent a sum of many class synchronizers.

Synchronization of a set of objects. Now, let us consider how to synchronize all the objects in a system with the model elements in a model. Intuitively, if $l_{\mathrm{C}} : \partial\mathrm{C} \overset{\Psi}{\longleftrightarrow} E$ is a class synchronizer, then we can apply set mapping $\{l_{\mathrm{C}}\}$ to the set of objects and the set of model elements.

To apply $\{l_{\mathrm{C}}\}$, we first define a pairing function $\mathrm{pair}^{\{l_{\mathrm{C}}\}} : \{\mathrm{C}\} \times \Psi \times \{E\} \to 2^{\mathrm{C}\times\Psi\times E}$ as follows: given $\pi_{\mathrm{set}} \in \{\mathrm{C}\}$, $\psi \in \Psi$, $y_{\mathrm{set}} \in \{E\}$,

- $\mathrm{new\_pairs} = \{(i, i_E) | i \in \pi_{\mathrm{set}} \wedge (i, \_) \notin \psi \wedge i_E \in y_{\mathrm{set}} \wedge (\_, i_E) \notin \psi \wedge \mathrm{isAligned}(i, i_E)\}$, where $\mathrm{isAligned}(i, i_E)$ is a domain-specific predicate that checks whether an object i and a model element i_E can be paired; intuitively, new_pairs contains the pairs of objects and elements that satisfy isAligned, such that ψ does not include the correspondence information about those objects and model elements;
- $P_{\mathrm{prev}} = \{(i, i_E) | i \in \pi_{\mathrm{set}} \wedge i_E \in y_{\mathrm{set}} \wedge \psi(i) = i_E\}$, i.e., the previously paired objects and elements;
- $P_{\mathrm{new}} = \{i | (i, i_E) \in \mathrm{new\_pairs} \wedge \nexists i'_E (i_E \neq i'_E \wedge (i, i'_E) \in \mathrm{new\_pairs}) \wedge \nexists i' (i \neq i' \wedge (i', i_E) \in \mathrm{new\_pairs})\}$, i.e., newly paired objects and elements;
- $P_{\mathrm{unpairedO}} = \{(i, \epsilon) | i \in \pi_{\mathrm{set}} \wedge \nexists (i, \_) \in P_{\mathrm{prev}} \cup P_{\mathrm{new}}\}$, i.e., the objects that cannot be paired;
- $P_{\mathrm{unpairedE}} = \{(\epsilon, i_E) | i_E \in m \wedge \nexists (\_, i_E) \in P_{\mathrm{prev}} \cup P_{\mathrm{new}}\}$, i.e., the elements that cannot be paired.
- $\mathrm{pair}^{\{l_{\mathrm{C}}\}}(\pi_{\mathrm{set}}, \psi, y_{\mathrm{set}}) = \{(i, \psi_{(i, i_E)}, i_E) | (i, i_E) \in P_{\mathrm{prev}} \cup P_{\mathrm{new}} \cup P_{\mathrm{unpairedO}} \cup P_{\mathrm{unpairedV}}\}$, where $\psi_{(i, i_E)} = \{(i', i'_E) | (i', i'_E) \in \psi \wedge ((i' = i \wedge i'_E = i_E) \vee i \xrightarrow{\mathsf{f}} i' \in \pi_{\mathrm{set}} \vee i'_E \in i_E.\mathsf{f}_E)\}$, and f and f_E are constructor parameters.

Remark 13. It is not difficult to verify that $\mathrm{pair}^{\{l_{\mathrm{C}}\}}$ meets the requirements needed by set mapping. Moreover, any two objects in an object set are mutually disjoint, because an object edit that is generated by l_{C} for a certain object never affects others when the edit is applicable. Accordingly, based on Theorem 6, $\{l_{\mathrm{C}}\}$ must be a well-behaved system-model BX.

Regarding the object set edits, insert and remove are concretized as $\mathrm{con}_{\mathrm{C}}$ and $\mathrm{des}_{\mathrm{C}}$, respectively, while modify is concretized as a pair of $\mathrm{con}_{\mathrm{C}}$ and $\mathrm{des}_{\mathrm{C}}$. However, in most cases, the class synchronizer $\{l_{\mathrm{C}}\}$ already produces the needed edits for the object set.

Although $\{l_{\mathrm{C}}\}$ is a well-behaved system-model BX, it still may fail in the synchronization. Remember that in a system module, an object is always created with a constructor that may require other objects as its (actual) constructor parameters (see Subsection 3.2). If we synchronize an object and its constructor parameters simultaneously, then $\{l_{\mathrm{C}}\}$ may not return a result. For instance, if $\pi = \{i, i', i \xrightarrow{\mathsf{f}} i', i' \xrightarrow{\mathsf{f}'} v\}$, $y = \{i_E, i'_E, i_E \xrightarrow{\mathsf{f}_E} i'_E\}$, and $\psi = \{(i, i_E), (i', i'_E)\}$, then $\{l_{\mathrm{C}}\}. \Lleftarrow (\pi, \psi, y)$ is equal to the merge of $l_{\mathrm{C}}. \Lleftarrow (\{i, i \xrightarrow{\mathsf{f}} i'\}, \{(i, i_E), (i', i'_E)\}, \{i_E, i_E \xrightarrow{\mathsf{f}_E} i'_E\}) = (\emptyset, \{(i, i_E), (i', i'_E)\})$ and $l_{\mathrm{C}}. \Lleftarrow (\{i', i' \xrightarrow{\mathsf{f}'} v\}, \{(i', i'_E)\}, \{i'_E\}) = (\{\mathrm{des}_{\mathrm{C}}\ i', \mathrm{cons}_{\mathrm{C}}\ i''\}, \{(i'', i'_E)\})$. However, the merge will fail according to definition of set mapping, because the resulting correspondence data does not converge.

To reflect such dependencies among objects, we define a partial order $\prec$ as follows: for any two objects i and i', $i \prec i'$ if and only if $i \xrightarrow{\mathsf{f}} i'$ belongs to the current system and f is a constructor parameter. Given a set S of objects, we can pick the largest independent subset, in which for any two objects i_1 and i_2, $i_1 \nprec i_2 \wedge i_2 \nprec i_1$. Similarly, we can also define a partial order $\prec$ over model elements as follows: for any two elements i_E and i'_E, $i_E \prec i'_E$ if and only if $i'_E \in i_E.\mathsf{f}_E$, where f_E is synchronized with f that is a constructor parameter. Given a set M of model elements, we can also pick the largest independent subset, in which for any two elements $i_{E,1}$ and $i_{E,2}$, $i_{E,1} \nprec i_{E,2} \wedge i_{E,2} \nprec i_{E,1}$.

Based on the partial order $\prec$ and $\{l_{\mathrm{C}}\}$, a set of objects is synchronized with a set of model elements by recursion, i.e., $\{l_{\mathrm{C}}\}^{+}$. First, we extract the largest independent subsets out from the given set of objects and the set of model elements, and synchronize the independent subsets. Then, we process the remainder recursively. Note that $\{l_{\mathrm{C}}\}$, in this context, is defined for independent (sub-)sets. During each recursion, we apply set mapping $\{l_{\mathrm{C}}\}$ to synchronize independent objects and model elements, resulting in more independent objects and model elements in the remainder of the given sets. Specifically, in forward direction, $\{l_{\mathrm{C}}\}$ generates an independent set of model elements from an independent set of objects; in backward direction, $\{l_{\mathrm{C}}\}$ turns a set of objects into an independent set of objects that are bijectively mapped onto the given independent set of model elements.

To apply recursion, we define the two partition functions $\mathrm{part}^{\{l_{\mathrm{C}}\}}_{\Rrightarrow}$ and $\mathrm{part}^{\{l_{\mathrm{C}}\}}_{\Lleftarrow}$ as follows.

(1) $\mathrm{part}^{\{l_{\mathrm{C}}\}}_{\Rrightarrow} : \{\mathrm{C}\} \times \Psi \to (\{\mathrm{C}\} \times \{\mathrm{C}\}) \times (\Psi \times \Psi)$: given a set $\pi_{\mathrm{set}} \in \{\mathrm{C}\}$ of objects and $\psi \in \Psi$,

- let $\pi_{\prec} = \{i | i \in \pi_{\mathrm{set}} \wedge \forall i'(i' \in \pi_{\mathrm{set}} \wedge i' \neq i \Rightarrow i \not\prec i' \wedge i' \not\prec i)\}$, and $\pi_{+} = \pi_{\mathrm{set}} - \pi_{\prec}$; i.e., $\pi_{\prec}$ is a largest independent subset of π_{set};
- let $\psi_{\prec} = \{(i, i_E) | (i, i_E) \in \psi \wedge (i \in \pi_{\prec} \vee i' \xrightarrow{\mathsf{f}} i \in \pi_{\prec})\}$, and $\psi_{+} = \psi - \psi_{\prec}$; i.e., $\psi_{\prec}$ contains the correspondence information on the objects to be converted and their constructor parameters, where we assume that f is a constructor parameter of i';

then, $\mathrm{part}^{\{l_{\mathrm{C}}\}}_{\Rrightarrow}(\pi_{\mathrm{set}}, \psi) = ((\pi_{\prec}, \pi_{+}), (\psi_{\prec}, \psi_{+}))$.

(2) $\mathrm{part}^{\{l_{\mathrm{C}}\}}_{\Lleftarrow} : \{\mathrm{C}\} \times \Psi \times \{E\} \to (\{\mathrm{C}\} \times \{\mathrm{C}\}) \times (\Psi \times \Psi) \times (\{E\} \times \{E\})$: given a set $\pi_{\mathrm{set}} \in \{\mathrm{C}\}$ of objects, $\psi \in \Psi$, and a set $y_{\mathrm{set}} \in \{E\}$ of model elements ($y_{\mathrm{set}} \neq \epsilon$),

- let $y_{\prec} = \{i_E | i_E \in y_{\mathrm{set}} \wedge \forall i'_E(i'_E \in y_{\mathrm{set}} \wedge i'_E \neq i_E \Rightarrow i_E \not\prec i'_E \wedge i'_E \not\prec i_E)\}$, and $y_{+} = y_{\mathrm{set}} - y_{\prec}$; i.e., $y_{\prec}$ is a largest independent subset of y_{set};
- let $\pi_{\prec} = \{i | i \in \pi_{\mathrm{set}} \wedge i_E \in y_{\prec} \wedge (i, i_E) \in \psi\} \cup \{i | (i, i_E) \in \mathrm{new\_pairs} \wedge \nexists i'_E(i_E \neq i'_E \wedge (i, i'_E) \in \mathrm{new\_pairs}) \wedge \nexists i'(i \neq i' \wedge (i', i_E) \in \mathrm{new\_pairs})\}$, $\pi_{+} = \pi_{\mathrm{set}} - \pi_{\prec}$; intuitively, $\pi_{\prec}$ contains the objects mapped onto $y_{\prec}$;
- let $\psi_{\prec} = \{(i, i_E) | (i, i_E) \in \psi \wedge (i_E \in y_{\prec} \vee (i_E \in i'_E.\mathsf{f}_E \wedge i'_E \in y_{\prec}))\} \cup \{(i, i_E) | (i, i_E) \in \psi \wedge i \in \pi_{\prec}\}$, and $\psi_{+} = \psi - \psi_{\prec}$, where we assume that f and f_E are constructor parameters;

then, $\mathrm{part}^{\{l_{\mathrm{C}}\}}_{\Lleftarrow}(\pi_{\mathrm{set}}, \psi, y_{\mathrm{set}}) = ((\pi_{\prec}, \pi_{+}), (\psi_{\prec}, \psi_{+}), (y_{\prec}, y_{+}))$.

(3) If $y_{\mathrm{set}} = \epsilon$, then $\mathrm{part}^{\{l_{\mathrm{C}}\}}_{\Lleftarrow}(\pi_{\mathrm{set}}, \psi, y_{\mathrm{set}}) = ((\pi_{\mathrm{set}}, \epsilon), (\epsilon, \psi), (\epsilon, \epsilon))$.

Remark 14. It is not difficult to verify that the two partition functions meet the requirements imposed by recursion. Hence, according to Theorem 4, $\{l_{\mathrm{C}}\}^{+}$ is a well behaved system-model BX. Particularly, $\{l_{\mathrm{C}}\}^{+}$ takes the dependencies among objects into account and overcomes the limitation of $\{l_{\mathrm{C}}\}$ when being used to synchronize a set of objects.

Alignment predicate. In the partition and the pairing functions, we used a domain-specific predicate isAligned to establish new correspondence between objects and model elements. It is used to handle the case that a system is synchronized with a model without previous correspondence data. isAligned checks whether an object can be paired with an element by using domain knowledge. For instance, supposing that there is a file model that synchronizes with the file system, a file in the file system can be paired with a model element that denotes a file by using their full paths; For a running IoT system and a IoT model that represents this running IoT system, an object (i.e., a device) can be paired with a model element that denotes a device by using IP addresses. We expect that an object (an element) can only be paired with at most one element (one object), i.e., $\mathrm{isAligned}(i, i_E) \Rightarrow \forall i'(\neg\mathrm{isAligned}(i', i_E)) \wedge \forall i'_E(\neg\mathrm{isAligned}(i, i'_E))$. However, if this requirement cannot be satisfied at runtime, a runtime error arises.

5.2 Feature synchronizer

A feature synchronizer $l_{\mathsf{f}} : \partial\mathsf{f} \overset{\Psi}{\longleftrightarrow} \mathsf{f}_E$ synchronizes a feature value of f in the system with a feature value of f_E in the model, where $\partial\mathsf{f}$ consists of the feature edits defined in Subsection 3.2. The basic idea of a feature synchronizer is to ensure that a system feature value $i \xrightarrow{\mathsf{f}} v$ correspond to a model feature value $i_E \xrightarrow{\mathsf{f}_E} u$, such that i and v corresponds to i_E and u, respectively.

Depending on whether f and f_E are ordered, we have two generic feature synchronizers as follows.

Definition 10 (Generic unordered feature synchronizer)**.** Given unordered feature f and unordered feature f_E, the generic unordered feature synchronizer $l_{\mathsf{f}} : \partial\mathsf{f} \overset{\Psi}{\longleftrightarrow} \mathsf{f}_E$ is defined as follows.

(1) $l_{\mathsf{f}}.K = \{(i \xrightarrow{\mathsf{f}} v, \psi, i_E \xrightarrow{\mathsf{f}_E} u) | \psi(i) = i_E \wedge F(v) = u\}$, where the function F has the same meaning as that is described in class synchronizer;

(2) $l_{\mathsf{f}}.\!\Rrightarrow\!(i \xrightarrow{\mathsf{f}} v, \psi) = (\psi(i) \xrightarrow{\mathsf{f}_E} F(v), \psi)$, i.e., create a model feature value that is consistent with the system feature value;

(3) $l_{\mathsf{f}}.\!\Lleftarrow\!(i \xrightarrow{\mathsf{f}} v, \psi, i_E \xrightarrow{\mathsf{f}_E} u) =$

- $(\emptyset, \psi)$, if $(i \xrightarrow{\mathsf{f}} v, \psi, i_E \xrightarrow{\mathsf{f}_E} u) \in l_{\mathsf{f}}.K$;
- $(\{\mathrm{modify}_{\mathsf{f}}(i, v, F^{-1}(u))\}, \psi)$, if $(i \xrightarrow{\mathsf{f}} v, \psi, i_E \xrightarrow{\mathsf{f}_E} u) \notin l_{\mathsf{f}}.K \wedge i_E = \psi(i)$;

(4) Supposing $i_E^{-1} = \psi^{-1}(i_E)$ and $u^{-1} = F^{-1}(u)$, then $l.\!\Lleftarrow\!(\epsilon, \psi, i_E \xrightarrow{\mathsf{f}_E} u) =$

- $(\{\mathrm{insert}_{\mathsf{f}}(i_E^{-1}, u^{-1})\}, \psi)$, if f is a not containment reference or $u^{-1} \neq \mathsf{uuid}^{(v)}\ u$;

- $(\{\text{moveIn}_{\mathsf{f}}(i_E^{-1}, u^{-1}, i)\}, \psi)$, if f is a containment reference, and i is current container of u^{-1} or i is the current container of v where $u^{-1} = \mathsf{uuid}^{(v)}\ u$;

(5) $l_{\mathsf{f}}.\Lleftarrow(i \xrightarrow{\mathsf{f}} v, \psi, \epsilon) =$

- $(\{\text{remove}_{\mathsf{f}}(i, v)\}, \psi)$, if f is a not containment reference, or $F(v) = \bot \wedge \nexists u(F^{-1}(u) = \mathsf{uuid}^{(v)}\ u)$;
- $(\{\text{moveOut}_{\mathsf{f}}(i, v, i')\}, \psi)$, if f is a containment reference, and $F(i')$ is current container u, such that $u = F(v) \wedge u \neq \bot$ or $u = F(\mathsf{uuid}^{(v)}\ u)$.

In brief, in forward transformation (branch 2), the synchronizer creates a model feature value that is consistent with the given system feature value. Backwards, if the given system feature value is consistent with or can be changed based on the model feature value, then the synchronizer does nothing or produces a modify edit (branch 3); if no system feature value is provided, then the synchronizer produces either an insert edit or a moveIn edit to add a feature value (branch 4); if no model feature value is provided, the synchronizer produces either a remove edit or a moveOut edit to delete a feature value (branch 5).

Remark 15. After checking Definition 7 and Eqs. (4) and (5), it is not difficult to find that the generic unordered feature synchronizer is a well-behaved system-model BX.

Afterward, we can lift l_{f} to a set mapping $\{l_{\mathsf{f}}\}$ with a pairing function $\text{pair}^{\{l_{\mathsf{f}}\}}$. Given a system feature value set $\text{vals}_s \in \{\mathsf{f}\}$, a model feature value set $\text{vals}_m \in \{\mathsf{f}_E\}$, and $\psi \in \Psi$, let

(1) $P_\psi = \{(i \xrightarrow{\mathsf{f}} v, \psi, i_E \xrightarrow{\mathsf{f}_E} u) | i \xrightarrow{\mathsf{f}} v \in \text{vals}_s \wedge i_E \xrightarrow{\mathsf{f}_E} u \in \text{vals}_m \wedge (\psi^{-1}(i_E) = i \vee \psi^{-1}(i_E) = \mathsf{uuid}^{(i)}\ i_E) \wedge (F^{-1}(u) = v \vee F^{-1}(u) = \mathsf{uuid}^{(v)}\ u)\}$,

(2) $P_S = \{(i \xrightarrow{\mathsf{f}} v, \psi, \epsilon) | i \xrightarrow{\mathsf{f}} v \in \text{vals}_s \wedge \nexists i_E \xrightarrow{\mathsf{f}_E} u((i \xrightarrow{\mathsf{f}} v, \psi, i_E \xrightarrow{\mathsf{f}_E} u) \in P_\psi)\}$,

(3) $P_M = \{(\epsilon, \psi, i_E \xrightarrow{\mathsf{f}_E} u) | i_E \xrightarrow{\mathsf{f}_E} u \in \text{vals}_m \wedge \nexists i \xrightarrow{\mathsf{f}} v((i \xrightarrow{\mathsf{f}} v, \psi, i_E \xrightarrow{\mathsf{f}_E} u) \in P_\psi)\}$,

then $\text{pair}^{\{l_{\mathsf{f}}\}}(\text{vals}_s, \psi, \text{vals}_m) = P_\psi \cup P_S \cup P_M$.

Remark 16. It is trivial to prove that the above pairing function satisfies the conditions required by set mapping. Because any edit to a certain feature value never touches other feature values, a set of feature values can be viewed as a set of disjoint sub-systems, each of which contains a single feature value. According to Theorem 6, $\{l_{\mathsf{f}}\}$ is a well-behaved system-model BX. Besides, the set edits for $\{l_{\mathsf{f}}\}$ are, in fact, $\partial\mathsf{f}$. Since l_{f} already produces those edits, there is no need for $\{l_{\mathsf{f}}\}$ to generate extra set edits.

The case that feature f (and f_E) is ordered is very similar to the unordered case. We define a generic ordered feature synchronizer as follows.

Definition 11 (Generic ordered feature synchronizer)**.** Given ordered feature f and ordered feature f_E, the generic ordered feature synchronizer $l_{\mathsf{f}} : \partial\mathsf{f} \overset{\Psi}{\longleftrightarrow} \mathsf{f}_E$ is defined as follows.

(1) $l_{\mathsf{f}}.K = \{(i \xrightarrow{\mathsf{f}[p]} v, \psi, i_E \xrightarrow{\mathsf{f}_E[q]} u) | \psi(i) = i_E \wedge F(v) = u \wedge p = q\}$, where the function F has the same meaning as that is described in class synchronizer;

(2) $l_{\mathsf{f}}.\Rrightarrow(i \xrightarrow{\mathsf{f}[p]} v, \psi) = (\psi(i) \xrightarrow{\mathsf{f}_E[p]} F(v), \psi)$, i.e., create a model feature value that is consistent with the system feature value, including their posistions;

(3) $l_{\mathsf{f}}.\Lleftarrow(i \xrightarrow{\mathsf{f}[p]} v, \psi, i_E \xrightarrow{\mathsf{f}_E[q]} u) =$

- $(\{\text{modify}_{\mathsf{f}}(i, F^{-1}(u), p, q)\}, \psi)$, if $i_E = \psi(i)$; note that if $\text{cur}(p^o) = \text{cur}(q^n) \wedge v = F^{-1}(u)$ at runtime, then this modify changes nothing; in such a case, we still view the produced edit set as $\emptyset$;

(4) Supposing $i_E^{-1} = \psi^{-1}(i_E)$ and $u^{-1} = F^{-1}(u)$, then $l.\Lleftarrow(\epsilon, \psi, i_E \xrightarrow{\mathsf{f}_E[q]} u) =$

- $(\{\text{insert}_{\mathsf{f}}(i_E^{-1}, u^{-1}, q)\}, \psi)$, if f is a not containment reference or $u^{-1} \neq \mathsf{uuid}^{(v)}\ u$;
- $(\{\text{moveIn}_{\mathsf{f}}(i_E^{-1}, u^{-1}, q, i)\}, \psi)$, if f is a containment reference, and i is current container of u^{-1} or i is the current container of v where $u^{-1} = \mathsf{uuid}^{(v)}\ u$;

(5) $l_{\mathsf{f}}.\Lleftarrow(i \xrightarrow{\mathsf{f}[p]} v, \psi, \epsilon) =$

- $(\{\text{remove}_{\mathsf{f}}(i, p)\}, \psi)$, if f is a not containment reference, or $F(v) = \bot \wedge \nexists u(F^{-1}(u) = \mathsf{uuid}^{(v)}\ u)$;
- $(\{\text{moveOut}_{\mathsf{f}}(i, p, i')\}, \psi)$, if f is a containment reference, and $F(i')$ is current container u, such that $u = F(v) \wedge u \neq \bot$ or $u = F(\mathsf{uuid}^{(v)}\ u)$.

Remark 17. In short, the ordered feature synchronizer ensures that a system feature value exists if and only if there is model feature value that can be paired with the system value. By checking Definition 7 and Eqs. (4) and (5), we can prove that the above synchronizer is a well-behaved BX.

Similarly, l can be lifted to a list mapping $[l_{\mathsf{f}}]$ with a pairing function $\text{pair}^{[l_{\mathsf{f}}]}$. Given a system feature value list $\text{vals}_s \in \{\mathsf{f}\}$, a model feature value list $\text{vals}_m \in \{\mathsf{f}_E\}$, and $\psi \in \Psi$, let

(1) $P_\psi = \{(i \xrightarrow{\mathsf{f}[p]} v, \psi, i_E \xrightarrow{\mathsf{f}_E[q]} u) | i \xrightarrow{\mathsf{f}[p]} v \in \mathrm{vals}_s \wedge i_E \xrightarrow{\mathsf{f}_E[q]} u \in \mathrm{vals}_m \wedge (\psi^{-1}(i_E) = i \vee \psi^{-1}(i_E) =^{\mathsf{uuid}^{(i)}} i_E) \wedge (F'^{-1}(u) = v \vee F'^{-1}(u) =^{\mathsf{uuid}^{(v)}} u) \wedge (\mathrm{count}(i \xrightarrow{\mathsf{f}[p]} v) = \mathrm{count}_E(i_E \xrightarrow{\mathsf{f}_E[p]} u))\}$;

- if f (f_E) is a reference (EReference), then $F'^{-1} = F^{-1}$; otherwise $F'^{-1}(\_) = \bot$, i.e., when pairing primitive values, we only check their positions;

- $\mathrm{count}(i \xrightarrow{\mathsf{f}[p]} v) = |\{p' | p' \leqslant p \wedge i \xrightarrow{\mathsf{f}[p']} v \in \mathrm{vals}_s\}|$, and $\mathrm{count}_E(i_E \xrightarrow{\mathsf{f}_E[p]} u)$ is analogous;

(2) $P_S = \{(i \xrightarrow{\mathsf{f}[p]} v, \psi, \epsilon) | i \xrightarrow{\mathsf{f}[p]} v \in \mathrm{vals}_s \wedge \nexists i_E \xrightarrow{\mathsf{f}_E[q]} u((i \xrightarrow{\mathsf{f}[p]} v, \psi, i_E \xrightarrow{\mathsf{f}_E[q]} u) \in P_\psi)\}$;

(3) $P_M = \{(\epsilon, \psi, i_E \xrightarrow{\mathsf{f}_E[q]} u) | i_E \xrightarrow{\mathsf{f}_E[q]} u \in \mathrm{vals}_m \wedge \nexists i \xrightarrow{\mathsf{f}[p]} v((i \xrightarrow{\mathsf{f}[p]} v, \psi, i_E \xrightarrow{\mathsf{f}_E[q]} u) \in P_\psi)\}$,
then $\mathrm{pair}^{[l_\mathsf{f}]}(\mathrm{vals}_s, \psi, \mathrm{vals}_m) = P_\psi \cup P_S \cup P_M$.

Remark 18. We can prove that the above pairing function satisfies the conditions required by list mapping. Especially, P_ψ ensures that for the object and the element at position i, if their former objects/elements are mutually paired and they are consistent, then they will also be paired. According to Theorem 7, $[l_\mathsf{f}]$ is a well-behaved system-model BX. Besides, the list edits for $[l_\mathsf{f}]$ are still $\partial\mathsf{f}$.

5.3 Building system-model synchronizer

Given a system module Π, we can split Π into many disjoint sub-system modules, i.e., the sub-system modules $\Pi_{\textsc{c}_1}, \Pi_{\textsc{c}_2}, \ldots, \Pi_{\textsc{c}_m}$ of all class $\textsc{c}_1, \textsc{c}_2, \ldots, \textsc{c}_m$ and the sub-system modules Π_{f_1}, Π_{f_2},..., Π_{f_n} of structural features $\mathsf{f}_1, \mathsf{f}_2, \ldots, \mathsf{f}_n$ that are not constructor parameters, such that $\Pi = \Pi_\textsc{c} \oplus \Pi_{\mathsf{f}_1} \oplus \Pi_{\mathsf{f}_2} \oplus \cdots \oplus \Pi_{\mathsf{f}_n}$. For the model type Y, the case is similar, and we also have $Y = Y_{E_1} \oplus Y_{E_2} \oplus \cdots \oplus Y_{E_m} \oplus Y_{\mathsf{f}_{E,1}} \oplus Y_{\mathsf{f}_{E,2}} \oplus \cdots \oplus Y_{\mathsf{f}_{E,n}}$.

Given a model $y \in Y$, it is very easy to split y into many sub-models according to the corresponding sub-model types by traversing the entire model. However, to split a system $\pi \in \Pi$, we need a root object $i_{\mathrm{root}} \in \pi$ as the starting point and call the getter methods of all features (i.e., get_f declared in Subsection 3.2) repeatedly to traverse the system.

Supposing that we have realized the class synchronizers $l_{\textsc{c}_i} : \partial\textsc{c}_i \overset{\Psi}{\longleftrightarrow} E_i$ $(i = 1, \ldots, m)$ and the feature synchronizers $l_{\mathsf{f}_{E_j}}$ $(j = 1, \ldots, n)$, we can combine them into a system-model synchronizer $l_\Pi : \partial\Pi \overset{\Psi}{\longleftrightarrow} Y$ as

$$l_\Pi \equiv \{l_{\textsc{c}_1} \oplus l_{\textsc{c}_2} \oplus \cdots \oplus l_{\textsc{c}_m}\}^+; (l_{\mathsf{f}_1} \otimes l_{\mathsf{f}_2} \otimes \cdots \otimes l_{\mathsf{f}_n}), \tag{6}$$

where $l^*_{\mathsf{f}_i}$ is either $\{l_{\mathsf{f}_i}\}$ or $[l_{\mathsf{f}_i}]$. In brief, we synchronize all objects first with $\{l^*_{\textsc{c}_1} \oplus l^*_{\textsc{c}_2} \oplus \cdots \oplus l^*_{\textsc{c}_m}\}^+$ (as described in Subsection 5.1), and then synchronize feature values with $l^*_{\mathsf{f}_1} \otimes l^*_{\mathsf{f}_2} \otimes \cdots \otimes l^*_{\mathsf{f}_n}$[13].

Remark 19. l_Π has two parts, which are combined by sequential union. The former part is the synchronizer for objects, which has been discussed in Subsection 5.1. The later part is the parallel union of all $l^*_{\mathsf{f}_i}$, which is also a well-behaved system-model BX according to Theorem 1, because the edit to a feature value should not affect the value of another feature. Due to the fact that feature synchronizers never change the correspondence data, l_Π is a well-behaved BX according to Theorem 2.

We need the following necessary knowledge to create a concrete system-model synchronizer.

(1) The definition of a system module, including

- the classes in the system, and the implementation of their constructors and destructors;
- the features in the system, and the implementation of their getter methods and edits (for non constructor parameters);
- the domain constraints (i.e., the contracts) of system edits (see Subsection 3.3);
- optionally, the domain-specific predicate isAlive (see Subsection 3.2).

(2) The definition of a metamodel that consists of a set of EClasses and EStructuralFeatures.

(3) Information needed by concrete class synchronizers and feature synchronizers, including

- the mapping between classes and EClasses and the mapping between system features and EStructuralFeatures;
- the implementation of the domain-specific alignment predicate isAligned (see Subsection 5.1);
- optionally, the implementation of a bijective function (or a BX) F for a feature synchronizer that converts between an attribute and an EAttribute (see Subsection 5.2), i.e., F converts free data.

[13] All feature synchronizers share the same correspondence data generated by class synchronizers.

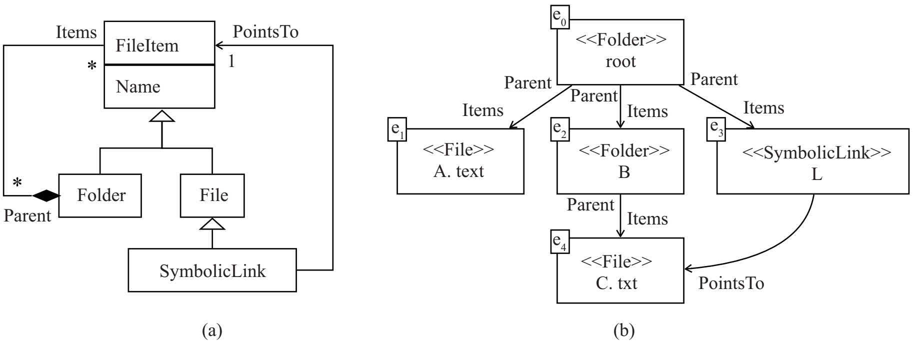


Figure 7 A file system example. (a) File metamodel; (b) a simple file model.

5.4 Example: file system synchronizer

This subsection demonstrates how to develop a file system synchronizer by concretizing the generic system-model synchronizer. In this example, a file system consists of a root directory and all the contents within this root directory. We use the Java class java.io.File to represent the system objects in a file system. A file model conforms to the metamodel (i.e., the model type) as shown in Figure 7(a). The metamodel defines three concrete EClasses, i.e., Folder, File, and SymbolicLink that is a subclass of File. A Folder contains many FileItems, and a SymbolicLink points to a FileItem. For simplicity, we assume that a SymbolicLink must point to a FileItem in the same file system. Figure 7(b) shows the expected file model that is consistent with the simple file system in Figure 2.

There are several difficulties in the development of the file system synchronizer. First, in Java, java.io.File is the class to denote files and folders. It means that java.io.File is mapped onto File, Folder, and SymbolicLink, so the mapping between Java classes and EClasses is not bijective. Second, java.io.File does not define a field to denote the target of the symbolic link, when the file object is a symbolic link file. Besides, java.io.File does not provide a setter method to change the symbolic link. Third, a file system imposes many domain constraints that determine the execution order of the system edits. For example, if a folder is deleted from disk, then its contents are also removed; we cannot insert a file into a folder that has not been created; two files (folders) should not have the same name.

File system module. There is only one class java.io.File in our example. However, we split all objects of java.io.File into the following three groups that are regarded as three subclasses.

(1) $\mathrm{c_{dir}}$ is the set $\{o|o \in \text{java.io.File} \wedge o.\text{isDirectory}()\}$ of folders. $\mathrm{c_{dir}}$ has the following features:

- $\mathrm{f_{dir\_name}}$ is the name of this folder, which is a constructor parameter;
- $\mathrm{f_{dir\_par}}$ is the parent folder of this folder, which is a constructor parameter;
- $\mathrm{f_{dir\_items}}$ is the content of this folder, which is an unordered containment reference.

(2) $\mathrm{c_{file}}$ is the set $\{o|o \in \text{java.io.File} \wedge o.\text{isFile}() \wedge o \text{ is not a symbolic link}\}$ of normal files; note that we can call API java.nio.file.Files.isSymbolicLink() to determine whether a file is a symbolic link. $\mathrm{c_{file}}$ is equipped with the following features:

- $\mathrm{f_{file\_name}}$ is the name of this file, which is a constructor parameter;
- $\mathrm{f_{file\_par}}$ is the parent folder of this file, which is a constructor parameter.

(3) $\mathrm{c_{link}}$ is the set $\{o|o \in \text{java.io.File} \wedge o.\text{isFile}() \wedge o \text{ is a symbolic link}\}$ of symbolic links. $\mathrm{c_{link}}$ is equipped with the following features:

- $\mathrm{f_{link\_name}}$ is the name of this symbolic link, which is a constructor parameter;
- $\mathrm{f_{link\_par}}$ is the parent folder of this symbolic link, which is a constructor parameter;
- $\mathrm{f_{link\_pointsTo}}$ is the content of this symbolic link, which is a singleton non-containment reference.

The constructors $\mathrm{con_{C_{dir}}}$, $\mathrm{con_{C_{file}}}$, and $\mathrm{con_{C_{link}}}$ can be realized by creating a new instance of java.io.File with a parent folder and a name, i.e., using new java.io.File(parent,name). Regarding destructors, since they are not required in Java, we omit them in this example.

The feature getters can be realized as follows. $\mathrm{get_{f_{dir\_name}}}$, $\mathrm{get_{f_{file\_name}}}$, and $\mathrm{get_{f_{link\_name}}}$ are implemented by calling API java.io.File.getName(). $\mathrm{get_{f_{dir\_par}}}$, $\mathrm{get_{f_{file\_par}}}$, and $\mathrm{get_{f_{link\_par}}}$ are implemented by calling API java.io.File.getParentFile(). $\mathrm{get_{f_{dir\_items}}}$ is implemented by calling API java.io.File.listFiles(). $\mathrm{get_{f_{link\_pointsTo}}}$ is implemented by calling API java.io.File.getCanonicalFile() that will return the link target if the file object is a symbolic link.

The feature edits for $\mathrm{f}_{\mathrm{dir\_items}}$ are realized as follows. $\mathrm{insert}_{\mathrm{f}_{\mathrm{dir\_items}}}$ is implemented by calling APIs java.io.File.mkdir() and java.io.File.createNewFile(), depending on whether a subfolder or a file will be inserted into this folder. $\mathrm{remove}_{\mathrm{f}_{\mathrm{dir\_items}}}$ is implemented by calling API java.io.File.delete(). $\mathrm{modify}_{\mathrm{f}_{\mathrm{dir\_items}}}$ is implemented by calling API java.io.File.renameTo(). $\mathrm{moveOut}_{\mathrm{f}_{\mathrm{dir\_items}}}$ is implemented by moving the original file/subfolder into a temporary folder and renaming it with a temporary name using API java.io.File.renameTo(). $\mathrm{moveIn}_{\mathrm{f}_{\mathrm{dir\_items}}}$ is implemented by moving the file/subfolder from the temporary folder into this folder and renaming it to the required name using API java.io.File.renameTo().

The feature edits for $\mathrm{f}_{\mathrm{link\_pointsTo}}$ is implemented as follows. $\mathrm{insert}_{\mathrm{f}_{\mathrm{link\_pointsTo}}}$ is implemented by calling API java.nio.file.Files.createSymbolicLink. $\mathrm{remove}_{\mathrm{f}_{\mathrm{link\_pointsTo}}}$ is implemented by turning the link target to a predefined invalid path using API java.nio.file.Files.createSymbolicLink. $\mathrm{modify}_{\mathrm{f}_{\mathrm{link\_pointsTo}}}$ is implemented by chaining $\mathrm{remove}_{\mathrm{f}_{\mathrm{link\_pointsTo}}}$ and $\mathrm{insert}_{\mathrm{f}_{\mathrm{link\_pointsTo}}}$.

Constraints of edits. A file system imposes many constraints that can be encoded as the preconditions of system edits. We list some examples as follows:

P0 Any object must be created before being used;

P1 To remove/move out a file/sub-folder from a parent folder, or to insert/move a file/sub-folder into a parent folder, the parent folder must be present, i.e., the preconditions of $\mathrm{insert}_{\mathrm{f}_{\mathrm{dir\_items}}}$, $\mathrm{remove}_{\mathrm{f}_{\mathrm{dir\_items}}}$, $\mathrm{modify}_{\mathrm{f}_{\mathrm{dir\_items}}}$, $\mathrm{moveIn}_{\mathrm{f}_{\mathrm{dir\_items}}}$, and $\mathrm{moveOut}_{\mathrm{f}_{\mathrm{dir\_items}}}$.

P2 To delete/rename/move out a folder/file, the folder/file must be present, i.e., the preconditions of $\mathrm{remove}_{\mathrm{f}_{\mathrm{dir\_items}}}$, $\mathrm{rename}_{\mathrm{f}_{\mathrm{dir\_items}}}$, and $\mathrm{moveOut}_{\mathrm{f}_{\mathrm{dir\_items}}}$.

P3 To insert/rename (i.e., modify)/move in a file/folder, file/folder name collision is not allowed, i.e., the preconditions of $\mathrm{insert}_{\mathrm{f}_{\mathrm{dir\_items}}}$, $\mathrm{modify}_{\mathrm{f}_{\mathrm{dir\_items}}}$, and $\mathrm{moveIn}_{\mathrm{f}_{\mathrm{dir\_items}}}$;

P4 To move a file/folder into a target folder, this file/folder must be present in the temporal folder, i.e., the precondition of $\mathrm{moveIn}_{\mathrm{f}_{\mathrm{dir\_items}}}$.

Class and feature mapping. The mapping from system classes and features to the EClasses and EStructuralFeatures defined in Figure 7(a) is straightforward, as follows:

- $\mathrm{c}_{\mathrm{dir}}$, $\mathrm{c}_{\mathrm{file}}$, and $\mathrm{c}_{\mathrm{link}}$ are mapped onto Folder, File, and SymbolicLink, respectively;
- $\mathrm{f}_{\mathrm{dir\_name}}/\mathrm{f}_{\mathrm{file\_name}}/\mathrm{f}_{\mathrm{link\_name}}$ and $\mathrm{f}_{\mathrm{dir\_par}}/\mathrm{f}_{\mathrm{file\_par}}/\mathrm{f}_{\mathrm{link\_par}}$ are mapped onto name and parent of Folder/File/SymbolicLink (inherited from FileItem), respectively;
- $\mathrm{f}_{\mathrm{dir\_items}}$ is mapped onto item of Folder, and $\mathrm{f}_{\mathrm{link\_PointsTo}}$ is mapped onto pointsTo of SymbolicLink.

Alignment predicate. The last step to develop the file system synchronizer is to define isAligned that can compute the missing correspondences between objects of java.io.File and model elements of FileItem. The pair function is quite simple—comparing the full path of an object and an element.

Now, we have defined all necessary information that is needed to derive the file system synchronizer from the generic system-model synchronizer that is defined in Eq. (6).

Runtime behavior. We have implemented this file system synchronizer in our prototype tool support. Figure 8 shows the execution of this synchronizer. The original file system is a super set of Figure 2, where the link target, i.e., feature pointsTo, is denoted as a dashed arrow (which is also a superset of Figure 7(b)). The forward transformation of the file system synchronizer successfully generates a file model that reflects the folder structure.

Then, we change the file model programmatically as follows to simulate an automated model conversion: (1) remove Folder B, (2) move File C.txt to Folder root, (3) rename File A.txt to C.txt, (4) rename the original File C.txt to A.txt, (5) rename Folder D to renamed-D, (6) remove Folder E, (7) move File F.txt to renamed-D, and (8) change the target of pointsTo of SymbolicLin L to File F.txt. Please note that the above sequence of changes to the file model is invalid for a real file system because

- we cannot move a file from a folder that has been deleted, i.e., (1) and (2), and (6) and (7),
- and we cannot rename a file to a name that is occupied, i.e., (3) since we have moved the original File C.txt to the root folder in (2).

That is to say, if we try to propagate these model changes with the state-of-the-art change-driven approaches (e.g., SM@RT [12]), then we are unable to obtain the expected result.

The key of system-model BX is that it calculates the system edits according to the current state of the system and the model, rather than the changes/delta of the model (no matter whether they are valid to the system). By comparing the changed file model and the current file system, the file system synchronizer determines the following system edits:

e_1 remove root/B from root;

e_2 remove root/D/E from D;

e_3 set the link target of root/L to root/renamed-D/F.txt;

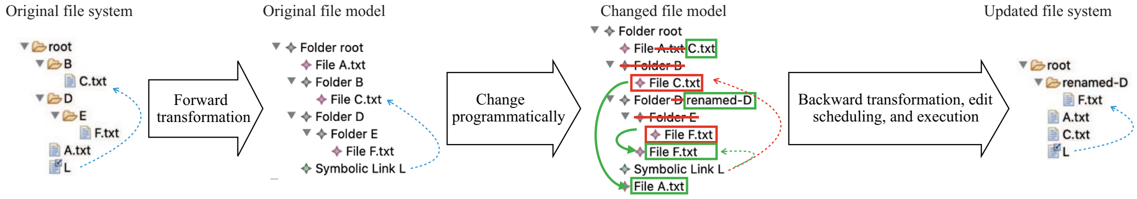


Figure 8 Execution of the file system synchronizer.

e_4 create a file object pointing to root/A.txt;
e_5 create a file object pointing to root/C.txt;
e_6 create a file object pointing to root/renamed-D;
e_7 create a file object pointing to root/renamed-D/F.txt;
e_8 move root/D/E/F out of root/D/E;
e_9 move root/D/E/F into root/renamed-D;
e_{10} move root/B/C.txt out of root/B;
e_{11} move root/B/C.txt into root and renamed it to root/A.txt;
e_{12} modify the original root/A.txt into root/C.txt in root;
e_{13} modify root/D into root/renamed-D in root.

After collecting all those edits, the file system synchronizer is able to plan an execution order as follows. According to P2, e_8 and e_{10} must run before e_2 and e_1, respectively. According to P1, e_2 must run before e_{13}. According to P0, e_4, e_5, e_6, and e_7 must run before e_{11}, e_{12}, e_{13}, and e_3, e_9, respectively. According to P4, e_8 and e_{10} must run before e_9 and e_{11}, respectively, because moveOuts must run before moveIns. According to P3, e_{12} must run before e_{11} to avoid file name collision. Finally, the file system synchronizer produces an edit sequence as follows during execution of backward transformation:

$$[e_{10}, e_5, e_6, e_8, e_7, e_4, e_1, e_{12}, e_2, e_3, e_{11}, e_{13}, e_9].$$

By applying this edit sequence, we obtain the updated file system as shown in Figure 8, which is consistent with the updated file model.

6 Conclusion and future work

This paper investigated how to synchronize a system and a model, and proposed system-model BX as a theoretical solution to this problem. This paper also proposed a set of combinators for system-model BX and proved their correctness. System-model BX is different from existing BX technologies in the following aspects: (1) the system-model BX does not treat a system as free data and requires BX developers to explicitly define system read/write operations; (2) the system-model BX is fully aware of domain knowledge and constraints on system edits; (3) the system-model BX computes a set of system edits during the backward transformation, and then plans a proper execution order according to the domain knowledge, rather than blindly executing them.

Regarding the future work, we plan to improve our work in the following aspects. First, we will continue enriching the formal definition of system-model BX and defining more useful combinators. Second, we notice that some system edits computed by a backward transformation can be merged to reduce the total number of edits to be executed. We will investigate how to automatically infer mergeable edits. Third, we will study how to automatically check the correctness of system-model BXs, such as the verification of Eqs. (4) and (5), the disjoint property required by many combinators, and the well-formedness conditions of pairing functions and partition functions. At last, we plan to enhance our tool support and develop a programming environment for system-model BX, and will conduct more case studies with the help of our tool.

Acknowledgements This work was supported by Beijing Natural Science Foundation (Grant No. 4192036).

References

1 Fischer S, Hu Z J, Pacheco H. The essence of bidirectional programming. Sci China Inf Sci, 2015, 58: 052106
2 Foster J N, Greenwald M B, Moore J T, et al. Combinators for bidirectional tree transformations. ACM Trans Program Lang Syst, 2007, 29: 17

3 Ko H S, Hu Z J. An axiomatic basis for bidirectional programming. In: Proceedings of the ACM on Programming Languages, Los Angeles, 2018. 1–29

4 Pacheco H, Zan T, Hu Z J. BiFluX: a bidirectional functional update language for XML. In: Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014), Canterbury, 2014. 1–12

5 Tran V D, Kato H, Hu Z J. Programmable view update strategies on relations. In: Proceedings of VLDB Endow, Tokyo, 2020. 726–739

6 Hofmann M, Pierce C B, Wagner D. Edit lenses. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Philadelphia, 2012. 495–508

7 Diskin Z, Gholizadeh H, Wider A, et al. A three-dimensional taxonomy for bidirectional model synchronization. J Syst Software, 2016, 111: 298–322

8 He X, Hu Z J. Putback-based bidirectional model transformations. In: Proceedings of the 26th ACM Joint Meeting on European Software, Lake City, 2018. 434–444

9 Xiong Y F, Song H, Hu Z J, et al. Synchronizing concurrent model updates based on bidirectional transformation. Softw Syst Model, 2013, 12: 89–104

10 Hermann F, Ehrig H, Orejas F, et al. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Softw Syst Model, 2015, 14: 241–269

11 Bencomo N, Götz S, Song H. Models@run.time: a guided tour of the state of the art and research challenges. Softw Syst Model, 2019, 18: 3049–3082

12 Song H, Huang G, Chauvel F, et al. Supporting runtime software architecture: A bidirectional-transformation-based approach. J Syst Software, 2011, 84: 711–723

13 Cánovas I J L, Jouault F, Cabot J, et al. API2MoL: automating the building of bridges between APIs and model-driven engineering. Inf Software Tech, 2012, 54: 257–273

14 Boronat A. Code-first model-driven engineering: on the agile adoption of MDE tooling. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, 2019. 874–886

15 Antkiewicz M, Czarnecki K, Stephan M. Engineering of framework-specific modeling languages. IIEEE Trans Software Eng, 2009, 35: 795–824

16 Bruneliére H, Cabot J, Dupé G, et al. MoDisco: a model driven reverse engineering framework. Inf Software Tech, 2014, 56: 1012–1032

17 Yu Y, Lin Y, Hu Z J, et al. Maintaining invariant traceability through bidirectional transformations. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 540–550

18 Reimann J, Seifert M, Aßmann U. Role-based generic model refactoring. In: Model Driven Engineering Languages and Systems. Berlin: Springer, 2010. 6395: 78–92

19 Noguera C, Duchien L. Annotation framework validation using domain models. In: Model Driven Architecture-Foundations and Applications. Berlin: Springer, 2008. 5095: 48–62

20 Eichberg M, Schäfer T, Mezini M. Using annotations to check structural properties of classes. In: Fundamental Approaches to Software Engineering. Berlin: Springer, 2005. 3442: 237–252

21 Kawanaka S, Hosoya H. biXid: a bidirectional transformation language for XML. ACM SIGPLAN Not, 2006, 41: 201–214

22 Bohannon A, Foster J N, Pierce B C, et al. Boomerang: resourceful lenses for string data. ACM SIGPLAN Not, 2008, 43: 407–419

23 Giese H, Wagner R. From model transformation to incremental bidirectional model synchronization. Softw Syst Model, 2009, 8: 21–43

24 Semeráth O, Debreceni C, Horváth Á, et al. Incremental backward change propagation of view models by logic solvers. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, San Malo, 2016. 306–316

25 Macedo N, Cunha A. Least-change bidirectional model transformation with QVT-R and ATL. Softw Syst Model, 2016, 15: 783–810

26 Eramo R, Pierantonio A, Rosa G. Managing uncertainty in bidirectional model transformations. In: Proceedings of ACM SIGPLAN International Conference on Software Language Engineering, Pittsburg, 2015. 49–58

27 Fritsche L, Kosiol J, Schürr A, et al. Efficient model synchronization by automatically constructed repair processes. In: Fundamental Approaches to Software Engineering. Cham: Springer, 2019. 11424: 116–133

28 Weidner M, Miller H, Meiklejohn C. Composing and decomposing op-based CRDTs with semidirect products. In: Proceedings of the ACM on Programming Languages, Virtual, 2020. 94

29 Mahdavi-Hezavehi S, Durelli V H S, Weyns D, et al. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. Inf Software Tech, 2017, 90: 1–26

30 Krupitzer C, Roth F M, VanSyckel S, et al. A survey on engineering approaches for self-adaptive systems. Pervasive Mobile Comput, 2015, 17: 184–206

31 Cheng B H C, de Lemos R, Giese H, et al. Software engineering for self-adaptive systems: a research roadmap. In: Software Engineering for Self-Adaptive Systems. Berlin: Springer, 2009. 5525: 1–26

32 Macías-Escrivá F D, Haber R, del Toro R, et al. Self-adaptive systems: a survey of current approaches, research challenges and applications. Expert Syst Appl, 2013, 40: 7267–7279

33 Zee K, Kuncak V, Rinard M. Full functional verification of linked data structures. ACM SIGPLAN Not, 2008, 43: 349–361

34 Boyapati C, Khurshid S, Marinov D. Korat: automated testing based on Java predicates. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Roma, 2002. 123–133

35 Aichernig B K. Contract-based testing. In: Formal Methods at the Crossroads. Berlin: Springer, 2003. 2757: 34–48

36 He X, Chen X, Cai S B, et al. Testing bidirectional model transformation using metamorphic testing. Inf Software Tech, 2018, 104: 109–129

37 Jackson D. Software Abstractions: Logic, Language, and Analysis. Cambridge: MIT Press, 2012

38 Jouault F, Allilaire F, Bézivin J, et al. ATL: a model transformation tool. Sci Comput Programm, 2008, 72: 31–39

39 Xiong Y F, Hu Z J, Zhao H Y, et al. Supporting automatic model inconsistency fixing. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Amsterdam, 2009. 315–324