On-Site Synchronization of Software Artifacts

Yingfei Xiong, Zhenjiang Hu, Masato Takeichi
Department of Mathematical Informatics
University of Tokyo, Tokyo, Japan
xiong@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp

Haiyan Zhao, Hong Mei
Key Laboratory of High Confidence
Software Technologies (Peking University)
Ministry of Education, Beijing, 100871, China
{zhhy,meih}@sei.pku.edu.cn

Abstract

Software development often involves multiple artifacts, such as feature models, UML models and code, which are in different formats but share a certain amount of information. When users change one artifact or change several artifacts simultaneously, we need to propagate these changes across all artifacts to ensure them consistent.

Existing approaches focus on off-site synchronization, that is, manipulating application data on external copies. However, in many software development tools, synchronization happens "on-site". The synchronization is tightly integrated into the tool and manipulates the internal data.

In this paper we propose a new approach to on-site synchronization, which takes modification operations on artifacts and produces new modification operations to make them consistent. The synchronization is incremental, ensuring short response time. We evaluate the performance of our approach by experiments.

1 Introduction

Software development often involves multiple artifacts, such as UML models, code and abstract views of code, which are in different formats but share a certain amount of information. When users change one artifact or change several artifacts simultaneously, we need to propagate these changes across all artifacts to ensure consistency among the artifacts. This process of propagating changes is called *synchronization* [4].

Existing approaches focus on off-site synchronization. That is, applications export their data on some intermediate formats, such as XML, and a synchronization system manipulates the intermediate data. Off-site synchronization is suitable for synchronizing off-the-shelf applications that were implemented without synchronization support. However, as many software engineering tools are designed with synchronization in mind from the beginning, in this paper

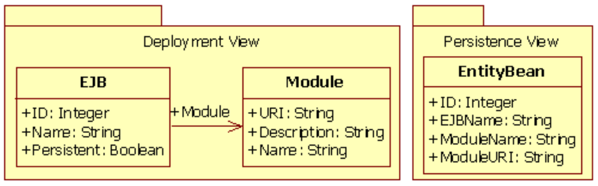


Figure 1. The Class Diagram of the EJB Tool

we argue that providing support for *on-site synchronization* is worth more investigation. On-site synchronization is tightly integrated into the applications and manipulates the internal data.

As an example, let us consider a simple Enterprise JavaBeans (EJBs) design tool that provides two editable views: the deployment view shows how EJBs are assembled into EJB modules, while the persistence view shows a list of persistent EJBs (EntityBeans). In a practical system, these two views may be stored in objects defined in Figure 1. If the `Persistent` attribute of an `EJB` object is *true*, there is a corresponding `EntityBean` object, where the `ID` attributes of the two objects are equal and the other attributes of the `EntityBean` object are equal to the related attributes of the `EJB` object and its `Module` object. When users modify, for instance, the `Name` attribute of an `EJB` object, the system should dynamically modify the `EJBName` attribute of the corresponding `EntityBean` object to make all objects consistent. That is, the synchronization is integrated into the tool and happens "on-site".

Off-site approaches have difficulties in supporting on-site synchronization. If we implement the synchronization between the two views using an off-site approach, we have to export all objects to the intermediate format, synchronize and import the objects again. It requires extra work to design the intermediate format and to implement the import and export, and also the system performance will be low. To support on-site synchronization, we need a new interface that can be tightly integrated into applications.

Besides the need for a new interface, on-site synchronization imposes some other new requirements. A notable one is that on-site synchronization has a strict requirement on response time. One way to achieve short response time is incremental synchronization, where the modified part is treated carefully to avoid recomputing on the unmodified parts. Although there is a bundle of work on incremental unidirectional on-site transformation [1] and incremental off-site synchronization [7], as far as we know, there is little discuss about incremental on-site synchronization.

As a matter of fact, on-site synchronization allows us to use more information such as modification operations, which makes room for solving the challenging problem of synchronizing artifacts with inner dependency. In the above example of the EJB design tool, there is inner dependency: the `EJB` objects depend on the `Module` objects that are in the same artifact. So if users delete a `Module` object, we should delete the `EJB` objects in the same view as well. Off-site synchronization has difficulties to cope with inner dependency because there is no information about which part has been modified. As will be seen later in Section 2 and Section 5, on-site synchronization can deduce the needed information from users' modification operations and propagate modification inside one artifact.

In this paper, we made the first attempt of defining a set of important *synchronizers* for on-site synchronization. Our synchronizers can be integrated into applications through modification operations. They take the modification operations on the application artifacts and produce new modification operations that can make the artifacts consistent. The features of our synchronizers are:

- Our synchronizers are constructed in a *compositional* way like lens [6]. A set of primitive synchronizers and a set of combinators provide users with a powerful mechanism to describe various synchronization behavior.
- Our synchronizers have *clear synchronization semantics*, satisfying the stability, preservation, propagation properties as proposed in our previous work [17] for off-site synchronization, ensuring predicable synchronization behavior [16].
- Our synchronizers can be *implemented efficiently*: they synchronize artifacts incrementally, ensuring a minimal response time. In addition, they can be used together with off-site synchronization, which may lead to a more general synchronization framework.

We have implemented all primitive synchronizers and combinators as an open source Java library[1] with which users can easily integrate our approach into their Java projects for on-site synchronization. In addition, we confirm the efficiency and practicality of our approach by testing our system on the EJB design tool, which we believe is a typical example capturing requirements in common synchronization applications.

[1]http://code.google.com/p/synclib/

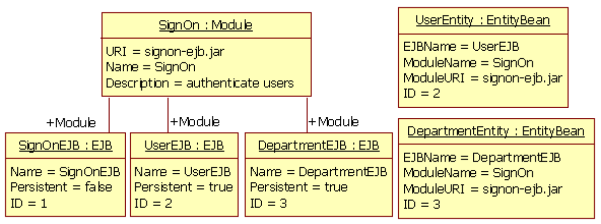


Figure 2. An Example of Objects

The rest of the paper is organized as follows. We first give an overview of our approach in Section 2. Then we establish some common notation in Section 3 and introduce the data types and modification operations we use in Section 4. Based on them we define a set of synchronizers and combinators in Section 5 and Section 6, and show how to synchronize the EJB design tool in Section 7. We evaluate the performance of our approach by experiments in Section 8. Finally, we discuss related work in Section 9 and conclude the paper in Section 10.

2 System Overview

Before explaining the detailed definition of synchronizers for on-site synchronization, we give a taste of synchronizers by showing some synchronization examples of the EJB design tool.

Figure 2 shows six objects (of the three types) that are consistent. The consistency relation is established and kept by a synchronizer s, which is defined in Section 7.

In order to identify an object that has been modified or to be modified, we assume each type of objects has an attribute called the *key attribute*, which is unique among all instances and should not be modified by users. For the `Module` objects, we may choose the `URI` attribute. For the `EJB` and `EntityBean` objects, we may choose the `ID` attribute.

Each synchronizer, including the synchronizer s, accepts modification operations as input and produces modification operations as output. When users modify, for example, the `ModuleName` attribute of `UserEntity` (whose `ID` is "2") to "SignOnModule", the following modification operations will be sent to the synchronizer s.

$$\langle \tau, \tau, \&\{2 \mapsto \&\{\texttt{ModuleName} \mapsto !\text{“\textit{SignOnModule}”}\}_\tau\}_\tau \rangle$$

The input is a triple, describing the modification operations on each type of objects. The first two τ's denote no modification on both `EJB` objects and `Module` objects. The third component describes the modification operation on the `EntityBean` objects: the `EntityBean` object with ID of 2 has its attribute `ModuleName` changed to *"SignOnModule"*.

Taking the input operations, the synchronizer s can automatically produce the following output.

$$\begin{aligned}\langle \tau, \&\{&\text{“\textit{signon-ejb.jar}”} \mapsto \&\{\texttt{Name} \mapsto !\text{“\textit{SignOnModule}”}\}_\tau\}_\tau,\\ &\&\{2 \mapsto \&\{\texttt{ModuleName} \mapsto !\text{“\textit{SignOnModule}”}\}_\tau,\\ &\quad 3 \mapsto \&\{\texttt{ModuleName} \mapsto !\text{“\textit{SignOnModule}”}\}_\tau\}\rangle\end{aligned}$$

which is the result as expected: the `Name` attribute of the `Module` object and the `ModuleName` attributes of all `EntityBean` objects in the same module are changed. There are neither unnecessary changes like modifying an `EJB` object, nor overwriting of the users' input modification operations like modifying `ModuleName` back to "SignOn".

For another example, if users delete the `Module` object with URI of *"signon-ejb.jar"*, the input will be:

$$\langle \tau, \&\{\textit{“signon-ejb.jar”} \mapsto \textit{del}\}_\tau, \tau \rangle$$

Because the `EJB` objects and the `EntityBean` objects are dependent on the `Module` object, all these objects should be deleted. After synchronization, the output will be as follows.

$$\begin{aligned}&\langle \&\{1 \mapsto \textit{del}, 2 \mapsto \textit{del}, 3 \mapsto \textit{del}\}_\tau,\\ &\ \&\{\textit{“signon-ejb.jar”} \mapsto \textit{del}\}_\tau,\\ &\ \&\{2 \mapsto \textit{del}, 3 \mapsto \textit{del}\}_\tau \rangle\end{aligned}$$

It is worth noting that there are often more than one way to synchronize objects and it is a challenge to provide a proper method for users to customize the system behavior. For the above example, it is also possible to set the attributes `EJB.Module`, `EntityBean.ModuleName` and `EntityBean.ModuleURI` to `null` to achieve consistent. We address this issue by introducing options to synchronizers, which will be explained in Section 5.

3 Notations

We start by introducing some common notations to be used in the rest of the paper. We use the lambda notations for functions, for instance, $\lambda a : a$ denotes an identity function and $f\ a$ indicates applying a function f to an argument a. When f is a partial function, we write $f\ a = \perp$ to mean f is undefined on a. We write $\textit{dom}(f)$ for the set of arguments on which f is defined. For any set S, we write $S_\perp$ to denote $S \cup \{\perp\}$. We write $\{\}$ for a function that maps anything to $\perp$. Suppose $f \in A \rightarrow B$, $a \in A$, $b \in B$. The update of a function, written as $f[a \leftarrow b]$, is defined as

$$f[a \leftarrow b]\ a' = \begin{cases} f\ a' & \text{if } a' \neq a \\ b & \text{if } a' = a \end{cases}$$

We write $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$ to denote $\{\}[k_1 \leftarrow v_1] \ldots [k_n \leftarrow v_n]$.

We use $\vec{v}$ to denote a n-tuple $\langle v_1, v_2, \ldots, v_n \rangle$. We write $\vec{v}.i$ for the ith component of the tuple $\vec{v}$. We write $\langle a \ldots a \rangle^n$ for a n-tuple where all elements are a. Let $\vec{a} = \langle a_1, \ldots, a_n \rangle$, $\vec{b} = \langle b_1, \ldots, b_n \rangle$ be two tuples, we write $\vec{a} \oplus \vec{b}$ for the concatenation of the two tuples $\langle a_1, \ldots, a_n, b_1, \ldots, b_n \rangle$. Let $\vec{a}$ be a n-tuple. We write $\vec{a}[i \leftarrow x]$ for a new tuple that is obtained by replacing the ith element of $\vec{a}$ with x. Let $\vec{f} \in (A \rightarrow B)^n$ be a tuple of functions and $\vec{v} \in A^n$ be a tuple, we use $\vec{f}\ \vec{v}$ to denote $\langle f_1\ v_1, f_2\ v_2, \ldots, f_n\ v_n \rangle$. Let $f \in A \rightarrow B$ be a function and $\vec{v} \in A^n$ be a tuple, we write $f \triangleright \vec{v}$ for $\langle f\ v_1, f\ v_2, \ldots, f\ v_n \rangle$.

4 Data Representation and Modification Operations

Data Representation We define synchronizers on a set of data types. We treat numbers and strings as unstructured *primitive values*, denoted by $\mathcal{P}$. We also denote the set of integers as $\mathcal{I}$, where $\mathcal{I} \subset \mathcal{P}$. *Dictionaries* map keys in $\mathcal{P}$ to values, denoted by $\mathcal{D}$. We use a universal set $\mathcal{U}$ to denote all primitive values and dictionaries. The dictionary set $\mathcal{D}$ and the universal set $\mathcal{U}$ are recursively defined as follows.

$$\boxed{\begin{aligned}\mathcal{D} = &\quad \{f \mid f \in \mathcal{P} \rightarrow \mathcal{U} \text{ and } \textit{dom}(f) \text{ is finite }\},\\ \mathcal{U} = &\quad (\mathcal{P} \cup \mathcal{D})_\perp .\end{aligned}}$$

Being simple, this data representation based on dictionaries can be used to describe various types of artifacts. For the two views in our EJB editor example, we can describe the objects of the two views using the above data representation[2]. For all instances of a class, we represent them as a dictionary mapping from the values of the key attributes to the objects. Each object is also represented as a dictionary mapping from the non-key attribute names to the attribute values. If the attribute value is a primitive value, we use the primitive value. If the attribute value is a reference to another object, we use the value of the key attribute of the referenced object.

As an example, the objects in Figure 2 can be represented by the following three dictionaries.

$$\begin{aligned}
\{1 \mapsto\ & \{\texttt{Name} \mapsto \textit{“SignOnEJB”},\\
& \ \texttt{Persistent} \mapsto \textit{false},\\
& \ \texttt{Module} \mapsto \textit{“signon-ejb.jar”}\},\\
2 \mapsto\ & \{\texttt{Name} \mapsto \textit{“UserEJB”},\\
& \ \texttt{Persistent} \mapsto \textit{true},\\
& \ \texttt{Module} \mapsto \textit{“signon-ejb.jar”}\},\\
3 \mapsto\ & \{\texttt{Name} \mapsto \textit{“DepartmentEJB”},\\
& \ \texttt{Persistent} \mapsto \textit{true},\\
& \ \texttt{Module} \mapsto \textit{“signon-ejb.jar”}\}\};\\
\{\textit{“signon-ejb.jar”} & \mapsto \{\texttt{Name} \mapsto \textit{“SignOn”},\\
\ \texttt{Description} & \mapsto \textit{“authenticate users”}\}\};\\
\{2 \mapsto\ & \{\texttt{EJBName} \mapsto \textit{“UserEJB”},\\
& \ \texttt{ModuleName} \mapsto \textit{“SignOn”},\\
& \ \texttt{ModuleURI} \mapsto \textit{“signon-ejb.jar”}\},\\
3 \mapsto\ & \{\texttt{EJBName} \mapsto \textit{“DepartmentEJB”},\\
& \ \texttt{ModuleName} \mapsto \textit{“SignOn”},\\
& \ \texttt{ModuleURI} \mapsto \textit{“signon-ejb.jar”}\}\}.
\end{aligned}$$

Modification Operations Modification operations are also defined on the data types. In general, users can perform three types of modification operations on the data: replacing a primitive value by a new value v, deleting a value,

[2]This representation is just conceptual. In actual implementation, we do not need to transform objects into the data types because we rely on modification operations to synchronize data.

and making modification inside a dictionary according to a structure ω mapping keys to modification operations. We view each modification operation as a function mapping from old values to new values and use $!v, \mathit{del}, \&\omega$ to denote the three types of operations, respectively. We also use a special function τ to indicate no modification.

To discuss the properties of modification operations, we formally define the set of modification operations $\mathcal{M}$ as follows.

$$\boxed{\begin{array}{l}
\mathcal{M} = \mathcal{PM} \cup \mathcal{DM} \cup \{\tau, \mathit{del}\} \\
\quad \text{where} \\
\quad \mathcal{PM} = \{!v \mid v \in \mathcal{P}\}; \\
\quad \mathcal{DM} = \{\&\omega \mid \omega \in \mathcal{P} \to \mathcal{M}\}; \\
\quad \ \tau \in \mathcal{U} \to \mathcal{U}, \tau\ a = a; \\
\quad \ \mathit{del} \in U \to \{\bot\},\ \mathit{del}\ a = \bot; \\
\quad \ !v \in \mathcal{P}_\bot \to \mathcal{P},\ !v\ a = v; \\
\quad \ \&\omega \in \mathcal{D}_\bot \to \mathcal{D}, \\
\quad \ \&\omega\ \bot = \&\omega\ \{\}, \\
\quad \ \&\omega\ d = d' \text{ where } \forall k \in \mathcal{P} : d'\ k = \omega\ k\ (d\ k).
\end{array}}$$

We use $\{\}_\tau$ to denote a special function that maps any thing to τ. Similarly, we write $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}_\tau$ to denote $\{\}_\tau[k_1 \leftarrow v_1] \ldots [k_n \leftarrow v_n]$.

If we have applied a modification operation to an artifact, applying it again will not change anything. This property can be used to check if a modification operation has been applied or not:

Property 1 $\forall m \in \mathcal{M},\ v \in \mathcal{U} :\ m\ (m\ v) = m\ v$

If two modification operations affect different parts of an artifact, we say the two operations are *distinct*. Formally, we say $m_1, m_2 \in \mathcal{M}$ are distinct if and only if they are commute: $m_1 \circ m_2 = m_2 \circ m_1$. We write $m_1 \ominus m_2$ if m_1 and m_2 are distinct.

For distinct operations, we have the following property:

Property 2 $\forall m_1, m_2 \in \mathcal{M} : (m_1 \ominus m_2 \Rightarrow m_1 \circ m_2 \in \mathcal{M})$

This property plays an important role in our design of synchronizers which map modification operations to modification operations. From this property we know that for a sequence of distinct modification operations, there is a modification operation in $\mathcal{M}$ that has the same effect. On the other hand, if users perform two operations that are not distinct, e.g. $!v_1$ followed by $!v_2$, it is sufficient to use the latter to represent the whole sequence. As a result, it suffices to consider modification operations in $\mathcal{M}$ instead of considering a sequence of operations.

If a modification operation m_1 is included in another modification operation m_2, we say m_1 is a *sub modification* of m_2, denoted as $m_1 \sqsubseteq m_2$. Formally, $m_1 \sqsubseteq m_2$ if and only if $m_1 \circ m_2 = m_2$.

5 Synchronizers

Before introducing our synchronizers, we should be more precise about on-site synchronization. Given a consistency relation R over n artifacts, an *on-site synchronizer* based on R takes users' modification operations on the artifacts and produces new modification operations which are expected to be applied to the artifacts to make them satisfy R. To perform incremental synchronization, we also need to keep some information like trace information, and update the information after synchronization. We call this information the state of the synchronizer, and denote the set of all states as Θ.

Formally, a *synchronizer* s synchronizing n artifacts consists of two parts, a consistency relation $s.R \subseteq \mathcal{U}^n$ and a triple $\langle s.\Theta, s.\vartheta, s.\mathit{sync}\rangle$, where $s.\vartheta \in s.\Theta$ is the start state and $s.\mathit{sync} \in s.\Theta \to \mathcal{M}^n \to s.\Theta \times \mathcal{M}^n$ is the synchronization function. Initially, the synchronizer is in the $s.\vartheta$ state and assume all artifacts are $\bot$ (each artifact contains no content in the initial stage). We also write $s.len$ for the number of artifacts that s synchronizes.

As argued in Section 2, the output modification operations should meet some requirements. We formalize the requirements by the following three properties.

Property 3 (Stability) *Let $\vec{\tau} = \langle \tau \ldots \tau \rangle^{s.len}$, we have* $\forall \theta \in s.\Theta :\ s.\mathit{sync}\ \theta\ \vec{\tau} = (\theta, \vec{\tau})$

Property 4 (Preservation) $s.\mathit{sync}\ \theta\ \vec{m} = \langle \theta', \vec{m}' \rangle \implies \forall i \in \{1, 2, \ldots, s.len\} :\ \vec{m}.i \sqsubseteq \vec{m}'.i$

Property 5 (Propagation) *If we construct a $s.\mathit{data}$ function using only the following two rules:*

1. $\langle \bot \ldots \bot \rangle^{s.len} \in s.\mathit{data}\ s.\vartheta$
2. $s.\mathit{sync}\ \theta\ \vec{m} = \langle \theta', \vec{m}' \rangle$
   $\implies \forall \vec{v} \in (s.\mathit{data}\ \theta) :\ (\vec{m}'\ \vec{v}) \in (s.\mathit{data}\ \theta')$

we have:

$\forall \theta \in s.\Theta :\ s.\mathit{data}\ \theta \subseteq s.R$

The three properties were originally proposed in our previous work [17] for off-site synchronization. Here we adapt them to on-site synchronization. Intuitively, the stability property is to prohibit the synchronizer making unnecessary changes. The preservation property requires the synchronizer to preserve all users' modification operations. The propagation property states that if we apply the modification operations produced by the synchronizer, the artifacts should be in the consistency relation.

In this section, we propose a powerful mechanism for users to specify synchronizers with various kinds of synchronization behavior. To achieve this, we adopt the compositional method used in many off-site synchronization approaches [6, 12]. As will be seen later, we shall provide some primitive synchronizers that synchronize artifacts according to some predefined primitive relations. We also provide combinators, where a combinator can combine the synchronizers into a new synchronizer to synchronize artifacts according to a new relation that is combined from the consistency relations of the inner synchronizers. In this way users can get their synchronizers by only considering combining relations.

In addition, all primitive synchronizers and synchronizers combined by combinators are ensured to satisfy the three properties. This can be formally proved and users can know their synchronizers work correctly once they have combined them. Due to space limit, we omit the formal proof and only state the result:

Theorem 1 *All synchronizers, either directly introduced in the paper or combined by combinators introduced in the paper, satisfy the stability property, the preservation property and the propagation property.*

Sometimes we have different ways to achieve consistency. To customize the behavior, we further parameterize primitive synchronizers and combinators with *options*. Options are a tuple of boolean values, often denoted by ψ. A different assignment to the options leads to a different synchronizer or combinator, with its own way to achieve consistency. By adjusting the behavior of local parts, users can customize the global behavior of their synchronizers. The number of options can be large. In this paper we show a small set of options as an example, and demonstrate how to customize the behavior when deleting a `Module` object.

We denote the set of all synchronizers as $\mathcal{X}$.

5.1 Primitive Synchronizers

Basic Synchronizers We start from a simple synchronizer *id* that is to keep two artifacts identical.

$$\begin{array}{l}
\textit{id}.R = \{(a,a) \mid a \in \mathcal{U}\} \\
\textit{id}.\Theta = \{\epsilon\} \\
\textit{id}.\vartheta = \epsilon \\
\textit{id.sync}\ \epsilon\ \langle m_1, m_2\rangle = \begin{cases} \langle \epsilon, \langle m_1 \circ m_2, m_1 \circ m_2\rangle\rangle & m_1 \ominus m_2 \\ \bot & \text{else} \end{cases}
\end{array}$$

The *id* synchronizer keeps the relation defined by $\textit{id}.R$, has only one constant state ϵ that does not change during synchronization, and has an important synchronization function *id*.*sync* saying that given two modifications m_1 and m_2 on two artifacts respectively, it returns a pair of the same modification operation $m_1 \circ m_2$ to be applied to the two artifacts if m_1 and m_2 are distinct, and returns error otherwise.

Two other basic synchronizers are *remove* and *equal*$[\![v]\!]$. The synchronizer *remove* removes an artifact by setting it to $\bot$. The synchronizer *equal*$[\![v]\!]$ forces an artifact to be equal to a primitive value v. The definitions of the two synchronizers are as follows.

$$\begin{array}{l}
\textit{remove}.R = \{\bot\} \\
\textit{remove}.\Theta = \{\epsilon\} \\
\textit{remove}.\vartheta = \epsilon \\
\textit{remove.sync}\ \epsilon\ \langle m\rangle = \begin{cases} \langle \epsilon, \langle \tau\rangle\rangle & \text{if} \quad m = \tau \\ \langle \epsilon, \langle \textit{del}\rangle\rangle & \text{elif } m \ominus \textit{del} \\ \bot & \text{else} \end{cases}
\end{array}$$

$$\begin{array}{l}
v \in \mathcal{P} \\
\textit{equal}[\![v]\!].R = \{\langle a\rangle \mid a = v \vee a = \bot\} \\
\textit{equal}[\![v]\!].\Theta = \{\epsilon\} \\
\textit{equal}[\![v]\!].\vartheta = \epsilon \\
\textit{equal}[\![v]\!].\textit{sync}\ \epsilon\ \langle m\rangle = \begin{cases} \langle \epsilon, \langle \textit{del}\rangle\rangle & m = \textit{del} \\ \textit{id.sync}\ \bot\ \langle !v, m\rangle & \text{else} \end{cases}
\end{array}$$

Algorithm 1: *dget*$^\psi$*.sync*

```
   Input: $\langle k, d, v\rangle, \langle m_k, m_d, m_v\rangle$
1  if $\textit{dget}^\psi.\textit{sync}'\ \langle k,d,v\rangle\ \langle m_k, m_d, m_v\rangle = \bot$ then return $\bot$;
2  $\langle m'_k, m'_d, m'_v\rangle \leftarrow \textit{dget}^\psi.\textit{sync}'\ \langle k,d,v\rangle\ \langle m_k, m_d, m_v\rangle$;
3  return $\langle\langle m'_k\ k, m'_d\ d, m'_v\ v\rangle, \langle m'_k, m'_d, m'_v\rangle\rangle$;
```

Synchronizers on Dictionaries The dynamic get synchronizer *dget*$^\psi$ synchronizes a dictionary $d \in \mathcal{D}$, a key $k \in \mathcal{P}$ and a value $v \in \mathcal{U}$, and ensures that v is the value obtained by querying d with k. It has three options: `d_over_v`, `prop_v_del` and `k_exists`. The option `d_over_v` determines whether we use d to update v or use v to update d when only k is modified. The option `prop_v_del` determines whether we delete d or delete $d.k$ when v is deleted. The option `k_exists` determines whether we ensure $d\ k \neq \bot$ by setting k to $\bot$ when $d\ k = \bot$.

$$\begin{array}{l}
\psi = \langle \texttt{d\_over\_v}, \texttt{prop\_v\_del}, \texttt{k\_exists}\rangle \\
\textit{dget}^\psi.R = \{(k,d,v) \mid d \in \mathcal{D}_\bot, v \in \mathcal{U}_\bot, k \in \mathcal{P}_\bot, \\
\quad d\ k = v \vee (d = \bot \wedge v = \bot) \vee k = \bot\} \\
\textit{dget}^\psi.\Theta = \textit{dget}^\psi.R \\
\textit{dget}^\psi.\vartheta = \langle \bot, \bot, \bot\rangle
\end{array}$$

The state of *dget*$^\psi$ is a tuple that records the current values of the three artifacts. Every time we synchronize, the state will change to new values.

The function *dget*$^\psi$*.sync* is a bit complex to write in mathematical notations, so we present it in pseudo code, as shown in Algorithm 1. The function just calls *dget*$^\psi$*.sync*$'$ defined in Algorithm 2 and applies the returned modification operations to the state. In Algorithm 2, first the algorithm deals with the situation where k and d are modified to $\bot$, respectively (Line 1-5). Then the algorithm changes $\bot$ to $\{\}$ and changes τ to $\&\{\}_\tau$ for later processing (Line 6-9). After that, the algorithm merges the value in the dictionary, the value artifact, and their modification operations using a *merge* function defined below (Line 10-13). If v is deleted, the algorithm further deletes d or deletes k according to the semantics of options (Line 14-20). Finally, the algorithm restores the modification operation on d to τ if the original one is τ (Line 21), and then returns the result (Line 22).

$$\begin{array}{l}
\textit{merge} \in \mathcal{U} \to \mathcal{U} \to \mathcal{M} \to \mathcal{M} \to \mathcal{M} \\
\textit{merge}\ v_1\ v_2\ m_1\ m_2 = \textit{merge}'\ v_1\ v_2\ (m_1 \circ m_2) \\
\textit{merge}'\ v_1\ v_2\ m = \\
\begin{cases}
\tau & \text{if} \quad v_1 = v_2 \wedge m = \tau \\
\textit{del} & \text{elif } m = \textit{del} \\
!(m\ v_1) & \text{elif } v_1 \in \mathcal{P}_\bot \wedge v_2 = \bot \wedge m \in \mathcal{PM} \cup \{\tau\} \\
!(m\ v_2) & \text{elif } v_1 \in \mathcal{P}_\bot \wedge v_2 \in \mathcal{P} \wedge m \in \mathcal{PM} \cup \{\tau\} \\
\textit{merge}'\ \{\}\ v_2\ m & \text{elif } v_1 = \bot \wedge v_2 \in \mathcal{D} \\
\textit{merge}'\ v_1\ \{\}\ m & \text{elif } v_1 \in \mathcal{D} \wedge v_2 = \bot \\
\&\omega & \text{elif } v_1, v_2 \in \mathcal{D} \wedge m = \&\omega' \\
\quad \text{where } \omega\ k = \textit{merge}'\ (v_1\ k)\ (v_2\ k)\ (\omega'\ k) & \\
\&\omega & \text{elif } v_1, v_2 \in \mathcal{D} \wedge m = \tau \\
\quad \text{where } \omega\ k = \textit{merge}'\ (v_1\ k)\ (v_2\ k)\ \tau & \\
\bot & \text{else}
\end{cases}
\end{array}$$

The static get synchronizer *sget*$^\psi[\![k]\!]$ is similar to *dget*$^\psi$, but the key k is statically determined at the beginning. The

Algorithm 2: *dget*$^{\psi}$*.sync*$'$

```
Input: $\langle k, d, v\rangle, \langle m_k, m_d, m_v\rangle$
$k' \leftarrow m_k\ k$;
if $k' = \bot$ then return $\langle m_k, m_d, m_v\rangle$;
if $m_d = del$ then
    if $m_v \ominus del \wedge m_k \ominus del$ then return $\langle m_k, del, del\rangle$;
    else return $\bot$;
$m'_d \leftarrow m_d$;
if $d = \bot$ then $d \leftarrow \{\}$;
if $m'_d = \tau$ then $\omega \leftarrow \{\}_\tau$;
else $\&\omega \leftarrow m'_d$;
if not $(\omega\ k) \ominus m_v$ then return $\bot$;
if $\psi$.d_over_v then $m'_v \leftarrow merge\ (d\ k')\ v\ (\omega\ k')\ m_v$;
else $m'_v \leftarrow merge\ v\ (d\ k')\ m_v\ (\omega\ k')$;
$m'_d \leftarrow \&\omega[k' \leftarrow m'_v]$;
if $m'_v\ v = \bot$ then
    if $\psi$.prop_v_del then
        if $m_d \ominus del \wedge m_v \ominus del$ then return $\langle m_k, del, del\rangle$;
        else return $\bot$;
    else if $\psi$.k_exists then
        if $m_k \ominus del \wedge m_v \ominus del$ then return $\langle del, m_d, del\rangle$;
        else return $\bot$;
if $m'_d = \&\{\}_\tau$ then $m'_d = m_d$;
return $\langle !k', m'_d, m'_v\rangle$;
```

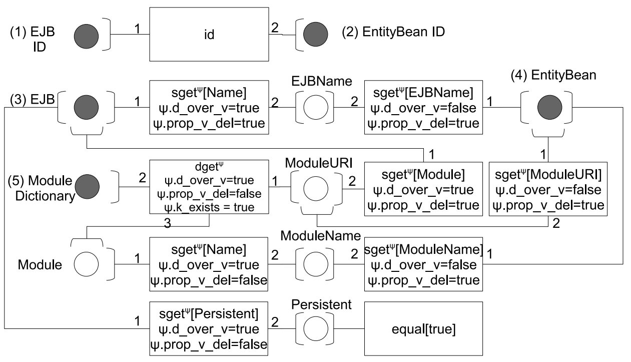
sget$^{\psi}[\![k]\!]$ synchronizer can be defined using the *dget*$^{\psi}$ synchronizer, as follows.

$$\boxed{\begin{array}{l}
\psi = \langle \texttt{d\_over\_v}, \texttt{prop\_v\_del}\rangle \\
\psi' = \langle \psi.\texttt{d\_over\_v}, \psi.\texttt{prop\_v\_del}, \mathit{false}\rangle \\
k \in \mathcal{P} \\
\mathit{sget}^{\psi}[\![k]\!].R = \{\langle d, v\rangle \mid \langle k, d, v\rangle \in \mathit{dget}^{\psi'}.R\} \\
\mathit{sget}^{\psi}[\![k]\!].\Theta = \mathit{sget}^{\psi}[\![k]\!].R \\
\mathit{sget}^{\psi}[\![k]\!].\mathit{sync}\ \langle d, v\rangle\ \langle m_d, m_v\rangle = \\
\quad \mathit{dget}^{\psi'}.\mathit{sync}\ \langle k, d, v\rangle\ \langle !k, d, v\rangle
\end{array}}$$

5.2 Combinators

In this section we introduce two useful combinators for constructing a bigger synchronizer by gluing smaller synchronizers statically and dynamically. The graph combinator statically combines synchronizers by organizing them into a *synchronizer graph*. The dictionary map combinator dynamically uses an inner synchronizer to synchronize a list of dictionaries.

Graph Combinator Figure 3 shows an example of a synchronizer graph which synchronizes an `EJB` object, an `EntityBean` object, their `ID` attributes and all `Module` objects. In the figure, the rectangles are *synchronizer holders* and the circles are *variables*. Synchronizer holders, as the name suggests, store synchronizers. Variables are artifacts being synchronized or artifacts for intermediate data. We distinguish the artifacts being synchronized and the artifacts for intermediate data by denoting the former with solid circles and the latter with hollow circles. We also call the former *parameters*. If a synchronizer connects to some variables by hand-like lines, we consider that the synchronizer synchronizes the variables. A number on a line shows the



(1) EJB ID
1
id
2
(2) EntityBean ID
(3) EJB
1
sget$^{\psi}$[Name] ψ.d_over_v=true ψ.prop_v_del=true
2
EJBName
2
sget$^{\psi}$[EJBName] ψ.d_over_v=false ψ.prop_v_del=true
1
(4) EntityBean
1
1
(5) Module Dictionary
2
dget$^{\psi}$ ψ.d_over_v=true ψ.prop_v_del=false ψ.k_exists = true
1
ModuleURI
2
sget$^{\psi}$[Module] ψ.d_over_v=true ψ.prop_v_del=true
sget$^{\psi}$[ModuleURI] ψ.d_over_v=false ψ.prop_v_del=true
3
2
Module
1
sget$^{\psi}$[Name] ψ.d_over_v=true ψ.prop_v_del=false
2
ModuleName
2
sget$^{\psi}$[ModuleName] ψ.d_over_v=false ψ.prop_v_del=true
1
1
sget$^{\psi}$[Persistent] ψ.d_over_v=true ψ.prop_v_del=false
2
Persistent
equal[true]

Figure 3. The Synchronizer Graph $\vec{g}_1$

index in the tuple of artifacts that the synchronizer synchronizes. A number on a parameter shows the index of the parameter.

Formally, we define a synchronizer graph $\vec{g}$ as a 5-tuple $\langle \mathcal{V}, \mathcal{H}, \vec{p}, \gamma, \varphi\rangle$:

- $\mathcal{V}$ is a set of variables.
- $\mathcal{H}$ is a set of synchronizer holders. $\mathcal{V}$ and $\mathcal{H}$ form the vertexes of the graph.
- $\vec{p} \in \mathcal{V}^+$ is a sequence of variables that are used as the parameters of the synchronizer graph. $\vec{p}$ does not contain duplicated elements.
- $\gamma : \mathcal{H} \to \mathcal{V}^+$ is a function mapping each synchronizer holder to a sequence of variables. For any $h \in \mathcal{H}$, $(\gamma\ h)$ does not contain duplicated elements. γ denotes the edges of the graph.
- $\varphi \in \mathcal{H} \to \mathcal{X}$ maps each synchronizer holder to a synchronizer.

Given a synchronizer graph $\vec{g}$, we turn it into a synchronizer by the *graph combinator*. The resulted synchronizer is denoted by *gc*$[\![\vec{g}]\!]$. The consistency relation, the state set and the start state of *gc*$[\![\vec{g}]\!]$ is defined as follows.

$$\boxed{\begin{array}{l}
\vec{g} = \langle \mathcal{V}, \mathcal{H}, \vec{p}, \gamma, \varphi\rangle \\
\mathit{gc}[\![\vec{g}]\!].R = \{\vec{v} \mid \exists f \in \mathcal{V} \to \mathcal{U} : f \rhd \vec{p} = \vec{v}\ \wedge \\
\qquad \forall h \in \mathcal{H} : f \rhd (\gamma\ h) \in (\varphi\ h).R\} \\
\mathit{gc}[\![\vec{g}]\!].\Theta = \{\theta \mid \forall h \in \mathcal{H} : (\theta\ h) \in (\varphi\ h).\Theta\} \\
\mathit{gc}[\![\vec{g}]\!].\vartheta = \lambda h \in \mathcal{H} :\ (\varphi\ h).\vartheta
\end{array}}$$

The consistency relation of *gc*$[\![\vec{g}]\!]$ is the conjunction of consistency relations of inner synchronizers. The state of *gc*$[\![\vec{g}]\!]$ is a function that maps each synchronizer holder to a state of the corresponding synchronizer, and the start state maps each synchronizer holder to the start state.

The algorithm of *gc*$[\![\vec{g}]\!]$*.sync* is given in Algorithm 3. The algorithm uses a function f_m to map each variable to the modification operation on the variable. At the beginning, all synchronizers are put into a set H (Line 2). In every iteration, the algorithm chooses a synchronizer in H that has the highest priority (explained below) and uses the synchronizer to synchronize its connected variables (Line 6-8). If the synchronizer modifies a variable, then all synchronizers connecting to the variable are added to H (Line 10-11). Then the process repeats until H is empty.

Algorithm 3: *gc*$[\![\vec{g}]\!]$.*sync*

```
    Input: $\theta, \vec{m}$
    Data: $f_m \in \mathcal{V} \rightarrow \mathcal{M},\ H \subseteq \mathcal{H}$
 1  $f_m \leftarrow \{\}_\tau$;
 2  $H \leftarrow \mathcal{H}$;
 3  foreach $i \in \{1, 2, \ldots, gc[\![\vec{g}]\!].len\}$ do
 4      $f_m \leftarrow f_m[\vec{p}.i \leftarrow \vec{m}.i]$;
 5  while $H \neq \emptyset$ do
 6      $h \leftarrow$ the synchronizer holder $h' \in \mathcal{H}$ where $\varphi\ h'$ has the
        highest priority;
 7      if $(\varphi\ h).sync\ (\theta\ h)\ (f_m \triangleright (\gamma\ h)) = \bot$ then return $\bot$;
 8      $\langle \theta_h, \vec{m}_h \rangle \leftarrow (\varphi\ h).sync\ (\theta\ h)\ (f_m \triangleright (\gamma\ h))$;
 9      foreach $j \in \{1, 2, \ldots, (\varphi\ h).len\}$ do
10          if $\vec{m}_h.j \neq f_m\ (\gamma\ h).j$ then
11              $H \leftarrow H \cup \{h \mid h$ connects to $v$ by $\gamma\}$;
12          $f_m \leftarrow f_m[(\gamma\ h).j \leftarrow \vec{m}_h.j]$ ;
13      $\theta \leftarrow \theta[h \leftarrow \theta_h]$;
14      $H \leftarrow H \backslash \{h\}$;
15  $\vec{m} \leftarrow f_m \triangleright \vec{p}$;
16  return $\langle \theta, \vec{m} \rangle$;
```

We use priority to ensure that we invoke more determined synchronizer first to reduce unnecessary failures. First, a synchronizer has the lowest priority if it does need to be invoked (e.g. no modification on its variables). Second, the more choices the synchronizer $(\varphi\ h)$ has to synchronize the artifacts, the lower the priority is. For example, *dget*$^\psi\ \theta\ \langle !k, \tau, !v \rangle$ has higher priority than *dget*$^\psi\ \theta\ \langle \tau, \tau, !v \rangle$ because the latter has choices to modify k while the former does not have. In this way we reduce the chance that an inner synchronizer make wrong choice and therefore reduce the chance of failure. The priority can be calculated in $O(1)$ time by examining the type of the synchronizer and the types of modification operations.

Note in the algorithm, an inner synchronizer can be invoked again when the modification operations on the connected variables are changed. In this way it is possible to propagate modification back into the same artifact, addressing inner dependency.

Dictionary Map Combinator The *dictionary map combinator* synchronizes several dictionaries using an inner synchronizer s. In the simplest situation, we synchronize two dictionaries d, d' by matching the items in the two dictionaries by key and synchronizing the matched pairs using s.

In a more complex situation, the two domains of keys are not equal, and we use a bijective mapping R_k to relate the keys of the two dictionaries. That is, we use s to synchronize $d\ k$ and $d'\ k'$ for each $\langle k, k' \rangle \in R_k$.

Additionally, the inner synchronizer may need some "global artifacts" that cannot be obtained from the two dictionaries. For example, we need the `Module` dictionary when we are synchronizing an `EJB` object and a `PersistentEJB` object in Figure 3. In this case, we restart the whole synchronization whenever the "global artifacts" are modified by the inner synchronizer s, so that all items in the dictionaries are consistent with the "global artifacts".

Furthermore, sometimes we want to use the keys in the inner synchronization. For example, we may need to use the `URI` attributes when we are synchronizing a `Module` object. However, we do not want the inner synchronizer to modify the keys because this will make the matched keys not in the relation R_k, so we put a $!k$ on the key k before invoking the inner synchronizer.

Finally, we sometimes synchronize more than two dictionaries and we achieve this by extending R_k over any number of domains. Formally, a *symmetric relation* $R_k \subset \mathcal{P}^n$ satisfies $\forall \vec{a}, \vec{b} \in R_k, i \in \mathcal{I} : \vec{a}.i = \vec{b}.i \implies \vec{a} = \vec{b}$ and $\forall a \in \mathcal{P}, i \in \{1, \ldots, n\} : \exists \vec{k} \in R_k : a = \vec{k}.i$. We write $||R_k||$ for n, the number of domains.

Now putting the above all together, we get the dictionary map combinator. We use *dc*$[\![s, R_k]\!]$ to denote the synchronizer obtained by combining an inner synchronizer s and a symmetric relation R_k using a dictionary map combinator. The inner synchronizer is assumed to synchronize a set of keys, the values mapped by the keys in the dictionaries, and "global artifacts" in turn. The consistency relation, the state set and the start state of *dc*$[\![s, R_k]\!]$ are defined below.

$$\boxed{\begin{array}{l}
s \in \mathcal{X}, R_k \text{ is a symmetric relation} \\
dc[\![s, R_k]\!].R = R_1 \cup \{\langle \bot \ldots \bot \rangle^{s.len - ||R_k||}\} \\
\quad R_1 = \{\vec{d} \oplus \vec{v} \mid \vec{d} \in \mathcal{D}^{||R_k||} \wedge \vec{v} \in \mathcal{U}^{s.len - 2||R_k||} \\
\qquad \wedge \forall \vec{k} \in R_k : \vec{k} \oplus (\vec{d}\,\vec{k}) \oplus \vec{v} \in s.R\} \\
dc[\![s, R_k]\!].\Theta = R_k \rightarrow s.\Theta \\
dc[\![s, R_k]\!].\vartheta = \lambda\, \vec{a} : s.\vartheta
\end{array}}$$

In the consistency relation R, the artifacts being synchronized are divided into two groups. $\vec{d}$ are dictionaries and $\vec{v}$ are "global artifacts" used by the inner synchronizer. Their relationships are defined by the consistent relation of the inner synchronizer. The state of *dc*$[\![s, R_k]\!]$ is a function mapping each $\vec{k} \in R_k$ to a state of the inner synchronizer. Initially the function maps any $\vec{k}$ to the start state.

The algorithm of the *dc*$[\![s, R_k]\!]$.*sync* function is described in Algorithm 4. The algorithm first deals with the situation where all artifacts are deleted (Line 1-5). Then the algorithm initializes a set K to store all key tuples that need to be synchronized (Line 7-9). For each key tuple in K, the algorithm invokes the *sync* function of the inner synchronizer and stores the result (Line 12-22). If the "global artifacts" are changed, the algorithm resynchronizes from the beginning to ensure items in the dictionaries are consistent with the "global artifacts" (Line 23-26).

6 Bridging On-Site and Off-Site

When performing an on-site synchronization, we assume the synchronizer is tightly integrated into the system and its state is always compatible with the artifacts. However, this assumption does not always hold. One exception occurs

Algorithm 4: The algorithm of $dc[\![s, R_k]\!].sync$

```
Input: $\langle \theta, \vec{m}_d \oplus \vec{m}_v \rangle$, where $|\vec{m}_d| = ||R_k||$
if $\vec{m}_d$ contains del then
    if $\vec{m}_d \oplus \vec{m}_v$ only contains del and $\tau$ then
        return $\langle dc[\![s, R_k]\!].\vartheta, \langle del \dots del \rangle^{dc[\![s,R_k]\!].len} \rangle$;
    else
        return $\perp$;
replace all $\tau$ in $\vec{m}_d$ with $\&\{\}_\tau$;
$K \leftarrow \{\vec{k} \mid \vec{k} \in R_k \wedge \vec{m}_d\, \vec{k} \neq \langle \tau \dots \tau \rangle^{||R_k||}\}$;
if $\vec{m}_v \neq \langle \tau \dots \tau \rangle^{|\vec{m}_v|}$ then
    $K \leftarrow K \cup dom(\theta)$;
resync_all:
foreach $\vec{k} \in K$ do
    $\vec{m}_k \leftarrow (\lambda v :!v) \triangleright \vec{k}$;
    $\vec{m}_s \leftarrow \vec{m}_k \oplus (\vec{m}_d\, \vec{k}) \oplus \vec{m}_v$;
    $\theta_s \leftarrow \theta\, \vec{k}$;
    if $s.sync\ \theta_s\ \vec{m}_s = \perp$ then return $\perp$;
    $\langle \theta_s, \vec{m}_s \rangle \leftarrow s.sync\ \theta_s\ \vec{m}_s$;
    $\theta \leftarrow \theta[\vec{k} \leftarrow \theta_s]$;
    foreach $i \in \{1, \dots, ||R_k||\}$ do
        $\vec{m}_d \leftarrow \vec{m}_d[i \leftarrow \vec{m}_d.i[\vec{k}.i \leftarrow \vec{m}_s.(i + ||R_k||)]]$;
    $\vec{m}'_v \leftarrow \vec{m}_v$;
    foreach $i \in \{1, \dots, dc[\![s, R_k]\!].len - ||R_k||\}$ do
        $\vec{m}'_v \leftarrow \vec{m}'_v[i \leftarrow \vec{m}_s.(2||R_k|| + i)]$;
    if $\vec{m}'_v \neq \vec{m}_v$ then
        $\vec{m}_v \leftarrow \vec{m}'_v$;
        $K \leftarrow K \cup dom(\theta)$;
        goto resync_all;
    $\vec{m}_v \leftarrow \vec{m}'_v$;
replace $\&\{\}_\tau$ in $\vec{m}_d$ with $\tau$ if it is originally $\tau$;
return $\langle \theta, \vec{m}_d \oplus \vec{m}_v \rangle$;
```

when we allow disjunction of synchronizers, that is, the artifacts can either be synchronized by a synchronizer a or by a synchronizer b. For example, when the `Persistent` property of an `EJB` object is `true`, we may use the synchronizer in Figure 3 to synchronize. When users modify the `Persistent` property to `false`, we may want to switch to the synchronizer in Figure 4. When we switch to the latter synchronizer, the state of the synchronizer may not be compatible with the artifacts.

To deal with such a situation, we add an off-site synchronization function $s.resync \in \mathcal{U}^{s.len} \rightarrow \mathcal{M}^{s.len} \rightarrow s.\Theta \times \mathcal{M}^{s.len}$ to a synchronizer s. The function, like $s.sync$, takes and produces modification operations. However, it also takes all artifacts as input and constructs a new state based on the artifacts. In this sense, we say that it is an off-site synchronization.

We would expect the $s.resync$ synchronization to exhibit a consistent behavior like the $s.sync$ function. To ensure this, we extend the properties of synchronizers to constrain the behavior of $s.resync$. Specifically, we add the following construction rule to $s.data$ in the propagation property to ensure the artifacts are consistent after synchronization:

$$\forall \vec{v} : (s.resync\ \vec{v}\ \vec{m} = \langle \theta, \vec{m}' \rangle \Longrightarrow (\vec{m}'\ \vec{v}) \in (s.data\ \theta))$$

Besides, we add the rule below to the preservation property to ensure users' modification operations are preserved:

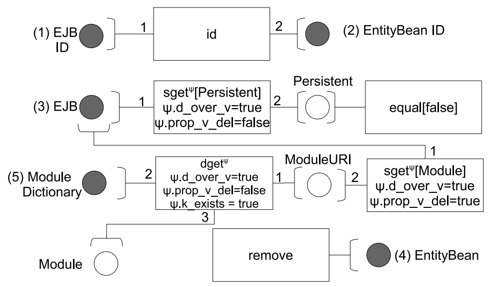


Figure 4. The Synchronizer Graph $\vec{g}_2$

$$s.resync\ \vec{v}\ \vec{m} = \langle \theta, \vec{m}' \rangle \Longrightarrow \forall i \in \{1, \dots, s.len\} :\ \vec{m}.i \sqsubseteq \vec{m}'.i$$

The *resync* function of existing primitive synchronizers and combinators can be similarly defined like the *sync* function. Here we give the definitions of *id*.*resync* and $dget^\psi$.*resync* as examples:

$$id.resync\ \langle v_1, v_2 \rangle\ \langle m_1, m_2 \rangle = \begin{cases} \langle \epsilon, \langle m, m \rangle \rangle & m_1 \ominus m_2 \\ \perp & \text{else} \end{cases}$$
$$\text{where } m = merge\ v_2\ v_1\ m_2\ m_1$$
$$dget^\psi.resync\ \vec{v}\ \vec{m} = dget^\psi.sync\ \vec{v}\ \vec{m}$$

The Switch Combinator Based on the *resync* function, we can achieve disjunction of synchronizations through the *switch combinator*. The switch combinator allows users to choose between a set of inner synchronizers. If an inner synchronizer fails to synchronize the artifacts (returning $\perp$), the switch combinator will try another synchronizer until the artifacts are synchronized or all synchronizers fail.

Let $\vec{s}$ be a tuple of synchronizers. We use $sc[\![\vec{s}]\!]$ to denote the synchronizer obtained by combining $\vec{s}$ using a switch combinator. $sc[\![\vec{s}]\!]$ is defined as follows.

$$m \in \mathcal{I} \wedge m > 1$$
$$\vec{s} \in \mathcal{X}^m \wedge \forall i \in \{1, \dots, m\} : \vec{s}.i.len = sc[\![\vec{s}]\!].len$$
$$sc[\![\vec{s}]\!].R = \bigcup_{k \in \{1,\dots,m\}} \vec{s}.k.R$$
$$sc[\![\vec{s}]\!].\Theta = \{\langle \vec{v}, i, \theta \rangle \mid i \in \mathcal{I} \wedge \vec{v} \in \vec{s}.i.data\ \theta \wedge \theta \in \vec{s}.i.\Theta\}$$
$$sc[\![\vec{s}]\!].\vartheta = \langle \langle \perp \dots \perp \rangle^{sc[\![\vec{s}]\!].len}, 1, \vec{s}.1.\vartheta \rangle$$
$$sc[\![\vec{s}]\!].sync\ \langle \vec{v}, i, \theta \rangle\ \vec{m} = \begin{cases} \langle \langle \vec{m}'\ \vec{v}, i, \theta' \rangle, \vec{m}' \rangle & \text{if} \quad \vec{s}.i.sync\ \theta\ \vec{m} \neq \perp \\ \quad \text{where } \langle \theta', \vec{m}' \rangle = \vec{s}.i.sync\ \theta\ \vec{m} & \\ \langle \langle \vec{m}'\ \vec{v}, k, \theta' \rangle, \vec{m}' \rangle & \text{elif } \begin{array}{l} \exists k \leq m : \vec{s}.k.resync\ \vec{v}\ \vec{m} \neq \perp \wedge \\ \forall j \in \{1, \dots, k-1\} : \vec{s}.j.resync\ \vec{v}\ \vec{m} = \perp \end{array} \\ \quad \text{where } \langle \theta', \vec{m}' \rangle = \vec{s}.k.resync\ \vec{v}\ \vec{m} & \\ \perp & \text{else} \end{cases}$$
$$sc[\![\vec{s}]\!].resync\ \vec{v}\ \vec{m} = \begin{cases} \langle \langle \vec{m}'\ \vec{v}, k, \theta \rangle, \vec{m}' \rangle & \text{if } \begin{array}{l} \exists k \leq m : \vec{s}.k.resync\ \vec{v}\ \vec{m} \neq \perp \wedge \\ \forall j \in \{1, \dots, k-1\} : \vec{s}.j.resync\ \vec{v}\ \vec{m} = \perp \end{array} \\ \quad \text{where } \langle \theta, \vec{m}' \rangle = \vec{s}.k.resync\ \vec{v}\ \vec{m} & \\ \perp & \text{else} \end{cases}$$

Intuitively, a switch combinator remembers in its state the current values of the artifacts, the index of the inner synchronizer used in the last synchronization and the state of the inner synchronizer. In $sc[\![\vec{s}]\!].sync$, the switch combinator first invokes the *sync* function of the inner synchronizer used in the last time. If the function fails, the switch combinator invokes the *resync* function of inner synchronizers one by one. In $sc[\![\vec{s}]\!].resync$, the switch combinator invokes the *resync* function of inner synchronizers one by one until one synchronizer synchronizes the artifacts.

7 Application: Synchronizing the EJB Design Tool

As we have introduced all synchronizers, we can use them to construct the synchronizer s for the EJB design tool. The definition of s is shown below.

$$\begin{array}{llll} s & = dc[\![s', =]\!] & s' & = sc[\![\langle s_1, s_2 \rangle]\!] \\ s_1 & = gc[\![\vec{g}_1]\!] & s_2 & = gc[\![\vec{g}_2]\!] \end{array}$$

That is, we first use the dictionary map combinator to match the `EJB` objects and the `EntityBean` objects and synchronize each pair with all `Module` objects using s'. The synchronizer s' switching between s_1 and s_2 to deal with the situations where `EJB.Persistent` $=$ *true* and `EJB.Persistent` $=$ *false*, respectively. The synchronizers s_1 and s_2 are constructed using the graph combinator and the two synchronizer graphs have already been shown in Figure 3 and Figure 4.

As mentioned before, when users delete a `Module` object, we may not want to remove objects but want to set the related attributes to `null`. We can achieve this by changing the `prop_v_del` options to `false` on $sget^{\psi}[\![$`Module`$]\!]$, $sget^{\psi}[\![$`ModuleName`$]\!]$, and $sget^{\psi}[\![$`ModuleURI`$]\!]$ in Figure 3 and on $sget^{\psi}[\![$`Module`$]\!]$ in Figure 4. In this way we can customize the global behavior by adjusting the behavior of local synchronizers.

8 Performance Evaluation

We evaluate the performance of our approach by experimenting with the synchronizer for the EJB design tool. We also compare our results with an off-site incremental synchronizer, medini QVT v1.1.2[3], which is a state-of-art implementation of the model transformation standard [14]. Our experiments are carried out on a laptop with an 1.70 GHz Intel(R) Pentium(R) M processor and 1.25 GB RAM.

Before carrying out the experiments, we prepare some common data. We first construct a large number of `EJB` and `Module` objects, where every 100 `EJB` objects belong to a `Module` object and the attributes are randomly assigned. Then we synchronize to get a consistent set of `EntityBean` objects.

In the first set of experiments we randomly choose a set of `EJB` objects, set their `Name` attributes to new values, and record the synchronization time. The result is shown in Table 1. The third column shows the time our tool takes and the forth column shows the time medini QVT takes. To be fair, we exclude the time during which medini QVT loads and saves XMI files, and only use the in-memory evaluation time reported by medini QVT.

From the table we can see, the synchronization time of our tool is a linear function of the modification size. The time remains constant when we increase the number of `EJB` objects and increases linearly when we increase the size of modification operations.

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

Table 1. Modifying the Name Attribute

Modification Size	Number of `EJB` objects	Time(ms) (Synchronizers)	Time(ms) (QVT)
500	1000	20	901
500	2000	20	2083
500	3000	20	6048
500	4000	20	10155
500	5000	20	16594
500	6000	20	23785
1000	6000	40	23894
1500	6000	60	24706
2000	6000	90	24575
2500	6000	130	25427

The time of medini QVT is much longer than our approach and is mainly related to the number of `EJB` objects. This is probably because QVT works off-site. When synchronizing, medini QVT has to re-check whether all applied rules are still valid and the number of rules is related to the number of `EJB` objects.

However, the synchronization time cannot always be a linear function of modification operations. In the second set of experiments we randomly choose a `EntityBean` object and modify its `ModuleName` attribute to a new value, just like the first example in Section 2. To synchronize, we have to iterate all `EntityBean` objects to find the objects in the same module for modifying their `ModuleName` attributes, and the time of the iteration is related to the number of objects. The result of the experiment is shown in Table 2. Note medini QVT cannot synchronize this modification because it does not handle inner dependency.

Table 2. Modifying the ModuleName Attribute

Number of `EJB` objects	Time(ms)
1000	50
2000	80
3000	101
4000	190
5000	250

From the table we can see that the synchronization time increases as the number of EJB objects increases. Nevertheless, the synchronization time is still short and we believe that it is efficient enough to support real applications.

9 Related Work

The mainstream work of off-site synchronization is work on bidirectional transformation [6, 10, 11, 12, 14]. In these approaches, a bidirectional language is used to describe a consistency relation R between two artifacts $a \in A, b \in B$, a forward function $f : A \times B \to B$ and a backward function $g : A \times B \to A$ at the same time [16]. Bidirectional transformation cannot directly support modifying the two artifacts at the same time. Benjamin and et al. [15] proposed a framework, Harmony, to support this with bidirectional transformation by designing a common artifact and use a reconciler to reconcile different versions of the common artifact.

Some bidirectional transformation approaches also target at synchronizing software artifacts. Two among them are Triple Graph Grammars (TGGs) [11], a model transformation approach applying early work in graph grammars to modeling environments [13, 2], and QVT relations [14], the standard of model transformation. The two approaches have been proved structurally similar by Greenyer and Kindler [8]. They are both rule-based, and can support incremental synchronization by re-checking applied rules. However, the TGGs approach is based on non-deleting graph grammars, and so far there is no well-defined semantics for object deletion and attribute modification in TGGs. On the other hand, QVT defines a clear semantics for object deletion and attribute modification by allowing only direct mapping between attributes. As a result, complex synchronization, like those involving string concatenation, cannot be supported by QVT. On the contrary, our approach can support string concatenation by defining a new synchronizer for concatenating strings.

Some researchers focus on the view consistency problem, which is a typical application of on-site synchronization. Amor and et al. [3] design a declarative language which focuses on bijective mappings between views and provides powerful support to expressions. Some other work [9, 5] provides general frameworks for view consistency, where users write code for identifying and handling inconsistency. Compared to these frameworks, our approach only requires users to composite synchronizers once and users automatically get the ability of handling inconsistency.

Our work started from our previous attempt [17] on synchronizing models from a forward transformation program. However, later we found that it is difficult to fully present the semantics of synchronization just in a forward program, and then we designed synchronizers, to provide a precise and flexible foundation for synchronization.

10 Conclusion and Future Work

In this paper we have proposed a compositional approach to on-site synchronization. Our approach is incremental, handles inner dependency and allows users to customize the behavior, making our approach suitable for on-site synchronization of software artifacts.

The set of synchronizers we proposed is growable in the sense that users can add new synchronizers, combinators, even new data types and new modification operations when the problem cannot be well tackled by existing ones. If the new synchronizers satisfy the three properties, they can work well with existing ones. We are actively exploring more synchronization scenarios in software engineering and designing new synchronizers when necessary.

References

[1] U. A. Acar. *Self-adjusting computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In *Proc. 2nd ECMDA*, pages 361–375, 2006.

[3] R. Amor, J. Hosking, and W. Mugridge. A declarative approach to inter-schema mappings. In *Modelling of Buildings Through Their Life-Cycle: Proc CIB W78/TG10 Conference*, 1995.

[4] M. Antkiewicz and K. Czarnecki. Design space of heterogeneous synchronization. In *Proc. 2nd GTTSE*, to appear.

[5] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Trans. Softw. Eng.*, 20(8):569–578, 1994.

[6] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

[7] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *Proc. 9th MoDELS*, pages 543–557, 2006.

[8] J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In *Proc. 10th MoDELS*, pages 16–30, 2007.

[9] J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, 1998.

[10] S. Kawanaka and H. Hosoya. biXid: a bidirectional transformation language for XML. In *Proc. 11th ICFP*, pages 201–214, 2006.

[11] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, University of Paderborn, June 2007.

[12] D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of XQuery. In *Proc. PEPM*, pages 21–30, 2007.

[13] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *In Proc. 22nd ICSE*, pages 742–745, 2000.

[14] Object Management Group. MOF QVT final adopted specification. http://www.omg.org/docs/ptc/05-11-01.pdf, 2005.

[15] B. C. Pierce, A. Schmitt, and M. B. Greenwald. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003.

[16] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proc. 10th MoDELS*, pages 1–15, 2007.

[17] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proc. 22nd ASE*, pages 164–173, 2007.