On-Site Synchronizers for Multi-View Applications*

Yingfei Xiong[1], Zhenjiang Hu[1,3], Song Hui[2], Masato Takeichi[1], Haiyan Zhao[2], Hong Mei[2]

[1]Department of Mathematical Informatics
University of Tokyo, Tokyo, Japan
xiong@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp

[2]Key Laboratory of High Confidence
Software Technologies (Peking University)
Ministry of Education, Beijing, 100871, China
{songhui06,zhhy,meih}@sei.pku.edu.cn

[3]Information Systems Architecture Research Division / GRACE Center
National Institute of Informatics

Many applications use multiple editable views to represent data in different formats. When users modify one view or change several views simultaneously, we need to synchronize the views by propagating these changes across all views. Different from existing studies on off-site synchronization which synchronizes data between applications, this kind of on-site synchronization has a strict requirement on response time, and usually has no predefined propagation direction.

In this paper we propose a new approach to on-site synchronization, which takes modifications on all data and produces new modifications to make them consistent. We deduce the propagation direction from the modifications, and achieve incremental synchronization by only computing the modified part. We have applied our approach to construct an EJB modeling tool and our experiments show that our synchronizers are hundred times faster than existing off-site synchronizers.

1 Introduction

It is common that one software application uses different views to represent its data. For example, a programming source code editor may show an outline of the code and some features [AC06] of the code as well as the code itself; a UML editor may use a class diagram to show the static structure of the system and use a sequence diagram to show the dynamic behavior.

In many cases, not only one view are editable. Since these views are representing shared information, when users modify one view, we need to transform and propagate the modification to other views to make all views consistent. Furthermore, multiple views may be modified at the same time, e.g., a group of designers are working one UML design models and a group of programmers are working on the implementing code simultaneously. In such situations, we need to synchronize the modifications on different views and detect possible conflicts.

For a concrete example, let us consider a simple Enterprise JavaBeans (EJBs) modeling tool, as shown in Figure 1. The tool provides two editable views: The deployment view and the the persistent view. The deployment view shows three EJBs:

*The research was supported in part by a grant from Japan Society for the Promotion of Science (JSPS) Grant-in-aid for Scientific Research (A) 19200002, the National Natural Science Foundation of China under Grant No. 60528006 and the National High Technology Research and Development Program of China (863 Program) under Grant No. of 2006AA01Z156.

`SignOnEJB`, `UserEJB`, and `DepartmentEJB`, all of which belong to a module `SignOn`. The persistent attributes of `UserEJB` and `DepartmentEJB` are `true`, and these persistent EJBs, called entity beans, are listed in the persistent view. Some of the information is shared between the two views, such as the module name and the EJB name, while some of the information is independently displayed on each view, such as the module description just below the module name and the primary keys of the entity beans. When users modify an element related to shared information, for instance, the EJB name of an entity bean on the persistent view, the system should dynamically modify the EJB name of the corresponding EJB on the deployment view. The process of propagating changes between data in different formats is called *heterogeneous synchronization* and the software component to perform such synchronization is called *synchronizers* [ACar].

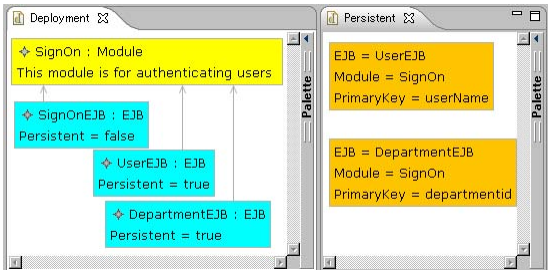

Figure 1. An EJB Modeling Tool

Existing synchronizers focus on *off-site* synchronization. That is, they synchronize data between off-the-shelf applications where these applications export their data in some intermediate formats like XML, and the synchronizers produce new XML files containing synchronized data. However, off-site synchronizers cannot well support synchronization in multi-view applications like the EJB modeling tool, where the data are synchronized *on-site* within one application. Not only exporting and importing the internal data requires extra programming work, but also the process takes too much time which is intolerable in most cases. We need new synchronizers to support this kind of *on-site synchronization*.

Compared to off-site synchronization, on-site synchronization imposes many new challenges. Here we highlight two challenges. The first one is response time. While off-site synchronization is often not sensitive to response time, on-site synchronization has a strict requirement on response time. When users modify one view, they are expected to see the modification of other views immediately.

The other one is about the direction of synchronization. Off-site synchronization often takes place between two applications, and the synchronization are performed with an explicit direction, i.e., change the data of one application according to the other or vice versa. However, in on-site synchronization the data are often mutually related and it is difficult to divide the data into two groups. For example, in the EJB application, when the attributes of an entity bean is modified by user, the application has to modify corresponding attributes of EJBs and modules. When a module is deleted by user, the application need to delete all EJBs and entity beans belonging to that module. It is not possible to divide the three types of objects into two groups where synchronization only takes place between groups but not inside a group.

In this paper, we made our first attempt of defining a set of on-site synchronizers to support on-site synchronization. Our synchronizers are incremental, ensuring a short response time, and are symmetric, treating every domain of data equally and assuming no specific direction in modification propagation.

The key factor in our design is modifications. When users modify the application data, the application capture users' modifications as an input to our synchronizers and our synchronizers produce new modifications which can make all data consistent. As no specific propagation direction is defined, the synchronizers use modifications as an indication and propagate the modification from modified parts to unmodified parts. Furthermore, because our synchronizers know users' modification, the synchronizers can perform an incremental synchronization where the modified part is treated carefully to avoid recomputing on the unmodified parts, ensuring a short response time.

```
EJBNameEqual = graph(ejb, entityBean) {
  vars = name;
  SGet<key="Name">(ejb, name);
  SGet<key="EJBName">(entityBean, name);
}
```

Figure 2. `ejb.Name = entityBean.EJBName`

To give a first taste of on-site synchronizers, let us consider a simple synchronizer which relates the name attribute of an EJB and the EJB name attribute of an entity bean. The code for the synchronizer is listed in Figure 2.

The `SGet<key="Name">` is a primitive synchronizer which synchronizes an object (the first argument) and a value (the second argument), ensuring the `Name` attribute of the object is equal to the value. Similarly, `SGet<key="EJBName">` ensures the `EJBName` attribute of its first argument is equal to the second argument. The two primitive synchronizers are combined into `EJBNameEqual` by a graph combinator which connects them through variables. The variables `ejb` and `entityBean` are external variables and `name` is an internal variable. The external variables are visible to the client of the synchronizer while the internal variables are invisible to the client. The `Name` attribute of `ejb` is ensured to be equal to `name` and the `EJBName` attribute of `entityBean` is ensured to be equal to `name`.

When users modify, say, the `Name` attribute of the `ejb` object to `"UserEJB"`, an input of ⟨`{Name->"UserEJB"}, nomod`⟩ will be passed to `EJBNameEqual`. The first component describes the modification on `ejb` while the second component `nomod` indicates `entityBean` is not modified. `EJBNameEqual` will notice that `ejb` is modified, and will invoke `SGet<key="Name">` with ⟨`{Name->"UserEJB"}, nomod`⟩ where the second `nomod` indicates `name` is not modified. `SGet<key="Name">` will propagate modifications from modified part to unmodified part, and produce ⟨`{Name->"UserEJB"}, "UserEJB"`⟩. Then `EJBNameEqual` will notice that `name` is modified and invoke `SGet<key="EJBName">` to further propagate a modification `{EJBName->"UserEJB"}` to `entityBean`. Finally, `EJBNameEqual` will produce the synchronized modifications ⟨`{Name->"UserEJB"}, {EJBName->"UserEJB"}`⟩ on `ejb` and `entityBean`.

If users modify the `EJBName` attribute of the `entityBean` object, the modification will be propagated in the reverse direction. If both objects are modified, the synchronizer will check if the two modifications are consistent, and report a failure if they conflict.

We start our definition of on-site synchronizers by giving a precise requirement for on-site synchronization (Section 2). A fundamental difficulty of designing synchronizers concerns the balance between *expressiveness* and *robustness* [FGM+07, Ste07]. In this work we choose to stick to only a reasonable subset of robustness properties to gain the expressiveness for most common synchronization tasks. Specifically, our on-site synchronizers satisfy the *stability*, *preservation*, *propagation* properties [XLH+07] – they are ensured to produce consistent result and make no unnecessary modifications – but not *total* [FGM+07] – on-site synchronizers are not ensured to find out all possible solutions for a specific input. We hope our work could serve as a foundation upon which others can build domain-specific synchronization language that is total.

We then proceed to the definition of concrete synchronizers. To simplify and generalize our definitions, we use a small set of data types to represent in-memory objects and define modification types on the data types (Section 3). Based on the data types and modification types we define a set of primitive synchronizers and a set of combinators (Section 4), where the combinators are used to combine primitive synchronizers into bigger synchronizers for more complex consistency relations, such as the graph combinator we have seen.

We have applied our work to actually construct the EJB modeling tool to see how our synchronizer work in practice (Section 5). The result is satisfactory and moreover, we also discovered an extra benefit of using on-site synchronizers: we can get rid of the key attributes used in off-site synchronizations. In off-site synchronization, users often need to designate some key attributes [Obj05, FGM+07]

which uniquely identify each data item. However, based on our experience, many application data do not have a suitable candidate to be a key attribute [YKW+08]. Because on-site synchronizer is tightly integrated into the application, we can make use of in-memory addresses to identify items and get rid of key attributes.

We evaluate the performance of our synchronizers through a set of experiments (Section 6). The result shows that our synchronizers are hundreds times faster than medini QVT [ikv], an implementation of an off-site synchronization language. We also compare our work with a variety of related work (Section 7). Finally, we discuss the remaining problems of current work and point out possible future directions (Section 8).

2 On-site Synchronization

Before defining concrete synchronizers, we discuss generally about on-site synchronization without targeting any particular data types. Later we will define specific data types and define synchronizers on those data types. We use $\vec{v}$ to denote a n-tuple $\langle v_1, v_2, \ldots, v_n \rangle$. We assume no nested structure in tuples, that is, $\langle v_1, \langle v2, v_3 \rangle \rangle = \langle v_1, v_2, v_3 \rangle$.

We assume a data type is collection of values. A *modification* defined on the data type D is a idempotent function where $m \in D \to D$ and $m \circ m = m$. The idempotent property of a synchronizer can be used to check whether a modification has been applied or not. If we apply a modification to a data item and we get the same data item, the modification has been applied to the data item. A special modification `nomod` is used to denote that the data item is not modified, where $\texttt{nomod}(d) = d$.

Users usually do not apply only one modification on the data, but a sequence of modifications. In many cases, a sequence of modifications has the same effect of one modification. To avoid a complex model involving both single modifications and sequences, we introduce the concept of *modification set*. A modification set M defined on the data type D is a set of modification operation closed on composition, that is, for any $m_1, m_2 \in M$ we have $m_1 \circ m_2 \in M$. To be simple, we assume each data type D has a corresponding modification set, denoted by M_D, and $\texttt{nomod} \in M_D$.

If two modification operations affect different parts of an artifact, we say the two operations are *distinct*. Formally, we say m_1, m_2 are distinct if and only if they are commutative: $m_1 \circ m_2 = m_2 \circ m_1$. We write $m_1 \ominus m_2$ if m_1 and m_2 are distinct. If two modifications are not distinct, we say they *conflict* with each other.

Another important relation over modifications is the *sub modification* relation, which indicates one modification is included in another modification. If $m_1 \circ m_2 = m_2$, we say m_1 is a sub modification of m_2, denoted by $m_1 \sqsubseteq m_2$.

A synchronizer synchronizes n data items that are defined in data types $D_1, D_2, \ldots, D_n$ is denoted as $s \in D_1 \leftrightarrow D_2 \leftrightarrow \cdots \leftrightarrow D_n$. We write $s \in\!\!\leftrightarrow D_1$ when $n = 1$. A synchronizer consists of four components. The first one is a consistency relation $s.R \subseteq D_1 \times D_2 \times \cdots \times D_n$ which the synchronizer tries to establish over the data items. The second component is a state set $s.\Theta$. To perform incremental synchronization, a synchronizer may need to keep some information like trace information, and update the information after synchronization. This kind of information is called states of the synchronizer and is defined by $s.\Theta$. The third component is an incremental synchronization function $s.sync \in M_{D_1} \times \cdots \times M_{D_n} \times s.\Theta \to M_{D_1} \times \cdots \times M_{D_n} \times s.\Theta$. This function takes a sequence of modifications on the data items, produces a new sequence of modifications, and updates the state of the synchronizer at the same time. The forth component is a non-incremental synchronization function $s.sync \in M_{D_1} \times \cdots \times M_{D_n} \times D_1 \times \cdots \times D_n \to M_{D_1} \times \cdots \times M_{D_n} \times s.\Theta$. This function is used at the initial stage when we do not have a state of the synchronizer, and it takes a sequence of data items instead of the state and constructs a state after synchronization. If the input modifications are conflicting or the synchronizer has failed to synchronize with the modifications, the $s.sync$ and $s.resync$ will return $\perp$.

To ensure synchronizers behave in a sensible way, researchers have proposed different

kinds of properties to constrain the behavior of synchronizers. Here we adopt three properties we proposed in our previous work for off-site synchronization[XLH+07] and adapt them for on-site synchronization.

Suppose s is a synchronizer of the type $s \in D_1 \leftrightarrow D_2 \leftrightarrow \cdots \leftrightarrow D_n$. The first property, stability, is to prevent synchronizers making unnecessary modifications. It requires that when users modify no data items, the synchronizer will also modify no data items.

Property 1 (Stability) $\forall \theta \in s.\Theta :$
$s.sync(\texttt{nomod}, \ldots, \texttt{nomod}, \theta) = \langle \texttt{nomod}, \ldots, \texttt{nomod}, \theta \rangle$

The second property, preservation, requires all user modifications should be kept. Each input modification should be a sub modification of the corresponding output. Here we write $\vec{v}.i$ for the ith component of the tuple $\vec{v}$.

Property 2 (Preservation) $s.sync(\vec{m}, \theta) = \langle \vec{m}', \theta' \rangle \Longrightarrow \forall i \in \{1, 2, \ldots, n\} : \ \vec{m}.i \circ \vec{m}'.i = \vec{m}'.i$

The third property, propagation, requires that the synchronizer should correctly propagate the modification to make all data consistent. Suppose $\vec{m}_1$ and $\vec{m}_2$ are two tuples of modifications, we write $\vec{m}_1 \circ \vec{m}_2$ for $\langle \vec{m}_1.1 \circ \vec{m}_2.1, \ldots, \vec{m}_1.n \circ \vec{m}_2.n \rangle$. Similarly, we write $\vec{m}(\vec{d})$ for a tuple constructed by applying each item of the modification tuple $\vec{m}$ to each item of the data sequence $\vec{d}$.

Property 3 (Propagation) *For any sequence of invocations:*

$$\begin{aligned} s.resync(\vec{m}_0, \vec{d}) &= \langle \vec{m}'_0, \theta_0 \rangle, \\ s.sync(\vec{m}_1, \theta_0) &= \langle \vec{m}'_1, \theta_1 \rangle, \\ &\cdots \\ s.sync(\vec{m}_n, \theta_{n-1}) &= \langle \vec{m}'_n, \theta_n \rangle; \end{aligned}$$

we have:
$\forall i \in \{0, \ldots, n\} : (\vec{m}'_i \circ \vec{m}'_{i-1} \circ \cdots \circ \vec{m}'_0)(\vec{d}) \in s.R.$

3 Data and Modification Types

To simplify our definition of synchronizers, in this section we define a small set of data types. We try to make this set of data types general so that other data types can be mapped to our data type and be synchronized by our synchronizers. When f is a partial function, we write $f(a) = \perp$ to mean f is undefined on a.

We treat booleans, integers, strings and $\perp$ as unstructured *primitive values*, denoted by `Prim`. For modifications on `Prim`, we consider only replacing a value by a new value. If $v \in$ `Prim`, we write $!v$ for a modification on `Prim`, where $!v(a) = v$. The modification set on `Prim` is $M_{\texttt{Prim}} = \{\texttt{nomod}\} \cup \{!v \mid v \in \texttt{Prim}\}$. We sometimes ignore the ! symbol if no confusion will be caused.

The only structured data type we consider is dictionaries. A dictionary in type `Dict < T >` maps keys in `Prim` to values in T, where T can be primitive values or other dictionary types. If we use a universal type `Object` to denote all primitive values and dictionaries, the typing rule for a dictionary type is as follows.

$$\frac{T \subseteq U, f \in \texttt{Prim} \to T \text{ and the domain of } f \text{ is finite}}{f \in \texttt{Dict} < \texttt{T} >}$$

The modifications on dictionaries are also structured. A dictionary modification on dictionary applies different modifications to values mapped by different keys. Formally, if $\omega \in \texttt{Prim} \to M_T$ is a function mapping keys to modifications, a dictionary modification $\&\omega$ is defined as: $\&\omega(d) = d'$ where $\forall k \in \texttt{Prim} : d'(k) = \omega(k)(d(k))$ The modification set on a dictionary type `Dict < T >` is defined as $M_{\texttt{Dict<T>}} = \{\texttt{nomod}\} \cup \{\&\omega \mid \omega \in \texttt{Prim} \to M_T\}$. We sometimes ignore the & symbol if no confusion will be caused.

A lot of other data structures can be mapped to this dictionary-based data types. For example, a set can be mapped to a dictionary by giving each value a random key or using the in-memory address of each value as its key. A sequence can be mapped to a dictionary by using the index as key, that is, `⟨"a", "b", "c"⟩` can be represented as `{1->"a", 2->"b", 3->"c"}`.

For the in-memory objects, we can also map them to the data types. For all instance of a class, we represent them as a dictionary mapping from the in-memory addresses to the instances. Each instance object is represented as a dictionary mapping from attribute names to 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 corresponding key of the referenced object.

As an example, the deployment view of the EJB tool can be represented by the following two dictionaries. In a practical system, the keys of the dictionaries can be replaced by in-memory addresses.

```
{1->{Name->"SignOnEJB",
     Persistent->false,
     Module->1},
 2->{Name->"UserEJB",
     Persistent->true,
     Module->1},
 3->{Name->"DepartmentEJB",
     Persistent->true,
     Module->1}
},
{1->{Name->"SignOn",
 Description->"This module is for ..."
}
```

4 Synchronizers

Based on the data and modification types, we are ready to define some concrete synchronizers and combinators. For the sake of brevity, we will only give the formal definitions of some synchronizers, and introduce the rest synchronizers and combinators informally. For all formal definitions, please refer to [XHT+08].

4.1 Basic Synchronizers

Identity We start from a simple synchronizer `Id<v1OverV2=true>` $\in$ `Object` $\leftrightarrow$ `Object` that keeps two data items identical. This synchronizer is parameterized on a boolean parameter `v1OverV2`, where a different assignment to the parameter leads to a different synchronization behavior. The value `true` after the equal sign is a default assignment to the parameter. Here the parameter `v1OverV2` defines the behavior of the *resync* function, and is explained below.

The `Id` synchronizer keeps no information in its state. Its *sync* function composites the two input modifications, and produce a pair of the composite result when the two input modifications are distinct. When the two inputs conflict, the function will fail (returning $\perp$). The *resync* acts according to the parameter `v1OverV2`. When two different values with no modification are inputted to the *resync* function, `Id` has to modify one data item according to the value of the other. The parameter `v1OverV2` determines whether we modify the second according to the first ($v1$ is over $v2$) or we modify the first according to the second ($v2$ is over $v1$). When `v1OverV2=true`, the function first apply the input modifications on the first data item, and use a function $findmod$ to find the smallest modification that can change the second data item into the first. The result of $findmod$ is composited with the original modifications and is returned. When `v1OverV2=false`, *resync* will try to modify the second data item into the first instead.

`Id<v1OverV2=opt>` $\in$ `Object` $\leftrightarrow$ `Object`

$R = \{\langle a, b\rangle \mid a = b\}$

$\Theta = \{\epsilon\}$

$$sync(m_1, m_2, \epsilon) = \begin{cases} \langle m_1 \circ m_2, m_1 \circ m_2, \epsilon\rangle & m_1 \ominus m_2 \\ \perp & \text{else} \end{cases}$$

$$resync(m_1, m_2, v_1, v_2) = \begin{cases} \langle m, m, \epsilon\rangle & m_1 \ominus m_2 \\ \perp & \text{else} \end{cases}$$

where $m =$
$\begin{cases} findmod(v_2, (m_1 \circ m_2)(v_1)) \circ m_1 \circ m_2 & \texttt{opt} = \texttt{true} \\ findmod(v_1, (m_1 \circ m_2)(v_2)) \circ m_1 \circ m_2 & \texttt{opt} = \texttt{false} \end{cases}$

SetMember, Equality and NotNull The second synchronizer `SetMember<tester, default>` $\in\leftrightarrow$ `Object` is used to ensure a data item is belonging to a specific set or not. The first parameter is a function `tester` $\in$ `Object` $\rightarrow$ `Boolean` which defines the set. If a data item is in the set, `tester` returns `true`, otherwise `tester` returns `false`. The second parameter `default` is a value in the set, which is used to produce a value in *resync* when the input value is not in the set.

The `SetMember` synchronizer keeps the current value of the data item in its state. Its *sync* function applies the modification to its state, and report a failure if the new state is not in the set. Its *resync* function tries to modify the input value to the default if the input is not in the set, and report a failure when the modification found conflicts with the input. The definition of `SetMember` is as follows.

`SetMember<tester=f, default=v0>` $\in \leftrightarrow$ `Object`
$R = \{a \mid f(a) = true\}$
$\Theta =$ `Object`

$$sync(m, v) = \begin{cases} \langle m, m(v)\rangle & f(m(v)) = true \\ \bot & \text{else} \end{cases}$$

$$resync(m, v) = \begin{cases} \langle m, m(v)\rangle & f(m(v)) = true \\ \langle findmod(v, v_0) \circ m, v_0\rangle & f(m(v)) = false \wedge \\ & findmod(v, v_0) \ominus m \\ \bot & \text{else} \end{cases}$$

By using `SetMember`, we can construct several useful synchronizers. For example, if we define a function `equal<v>` as

$$\texttt{equal} < \texttt{v} > (a) = \begin{cases} \texttt{true} & a = v \\ \texttt{false} & \text{else} \end{cases},$$

We can define a synchronizer `Equal` that ensures a data item is equal to a specific value.

```
Equal<v>=
  SetMember<tester=equal<v>, default=v>
```

Similarly, we can define a `NotNull` synchronizer that ensures a data item is not null.

$$\texttt{notNull}(a) = \begin{cases} \texttt{true} & a \neq \texttt{null} \\ \texttt{false} & \text{else} \end{cases}$$

```
NotNull<default>=
  SetMember<tester=notNull, default=default>
```

4.2 Synchronizers on Dictionaries

We define two primitive synchronizers on dictionaries. The two synchronizers both synchronize a dictionary and a data item, and ensure the data item is equal to the value mapped by a key in the dictionary. The difference is that the key of the static get synchronizer is statically determined at programming time and the key of the dynamic get synchronizer is dynamically obtained from a third data item.

Static Get The static get synchronizer `SGet<key, dictOverValue=true>`$\in$ `Dict` $<$ `T` $>$ $\leftrightarrow$ `T` synchronizes a dictionary with a data item and ensures the data item is equal to the value mapped by `key` in the dictionary. The second parameter `dictOverValue` is similar to `v1OverV2` in `Id`: in the *resync* function, whether the synchronizer try to modify the dictionary according to the data item or try to modify the data item according to the dictionary.

The implementation of `SGet` is similar to `ID`, except that it synchronizes an item in a dictionary with the other item. Because the modification on dictionaries is a function mapping keys to modifications, we can get the inner modification mapped by `key` and proceed as `Id`.

Dynamic Get The dynamic get synchronizer `DGet<dictOverValue=true, changeKey=false, keyFactory=null>`$\in$ `Prim` $\leftrightarrow$ `Dict` $<$ `T` $>$ $\leftrightarrow$ `T` ensures the third data item is equal to the value mapped by the first data item in the second data item, a dictionary. The first parameter `dictOverValue` has the same meaning as the `dictOverValue` parameter of the static get synchronizer: whether we modify dictionary according to the third data item or we modify the data item according to the dictionary.

The parameter `changeKey` is also a boolean value. If `changeKey=false`, then the dynamic get synchronizer will act the same behavior as the static get dictionary. Suppose the current values of the three data items are $\langle$`1, {1->"a", 2->"b"}, "a"`$\rangle$ and the input modifications are $\langle$`nomod, nomod, "b"`$\rangle$, the synchronizer will produce $\langle$`{1->"b"}, nomod, "b"`$\rangle$.

If `changeKey=true` and the first modification is `nomod`, the dynamic get synchronizer will first apply the input modifications to the dictionary and the data item, and then try to find a value in the changed dictionary whose value is equal to the changed data item. If there is such a value, the dynamic get will change the first data item to the key mapping to the value. If there is no such a value and `keyFactory=null`, or if the first modification is not `nomod`, the synchronzier acts the same behavior as `changeKey=false`.

For example, if the current values of the data items and the input modifications are the same as the above, but `changeKey=true`, the result will be $\langle$`nomod, 2, "b"`$\rangle$.

The third parameter `keyFactory` is effective only when `changeKey=true`. It is a function `keyFactoyr` $\in$ `Dict` $<$ `T` $>$ $\rightarrow$ `Prim` used to generate a new key that is different from all existing keys in the dictionary. When `keyFactory` is not `null`,

the synchronizer will create a new key if it cannot find a proper value in the dictionary. This parameter is useful when the dictionary is a set of objects and sometimes we want to create new objects from other modifications.

4.3 Combinators

We have seen a set of primitive synchronizers. In practice, the consistency relationship over data are often much more complex than the primitive synchronizers can cover. To specify these relationships, we adopt the compositional method used in many off-site synchronization approaches [FGM+07, LHT07]. These approaches define a set of combinators where a combinator can combine synchronizers into new a 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 combine the consistency relationships as they needed.

Graph Combinator We have seen graph combinator in the introduction. The synchronizer in Figure 2 ensures the `Name` attribute of `ejb` is equal to the `EJBName` attribute of `entityBean`. The graph combinator combines synchronizers using a graph. The nodes of the graph are variables (`ejb`, `entitybean`, and `name`), and the edges of the graph are synchronizers (the two `SGet` synchronizers). For example, the statement of `SGet<key="Name">(ejb, name)` connects the `SGet` synchronizer to the variable `ejb` and the variable `name`, enforcing the consistency relationship `SGet`.R over `ejb` and `name`. Variables are further classified into external variables and internal variables. Only external variables are expose to the synchronizers outside the graph. Input/output modifications are assumed to be applied on the external variables in the same order where the external variables appear.

The *sync* function of the graph combinator first puts the input modifications on the external variables, and invokes the inner synchronizers connecting to the external variables whose modifications are not `nomod`. Once an inner synchronizer is invoked, the output modification of the inner synchronizer will be put on the variables, and the *sync* function further invokes all synchronizers connecting to a variable whose modification is changed after the invocation. The *resync* function is similar to the *sync* function, except that *sync* invokes the *sync* function of inner synchronizers, while *resync* invokes the *resync* function of inner synchronizers.

It is possible that more than one inner synchronizers can be invoked at the same time. For example, a changed external variable is connected to more than one synchronizers. In such situations, the graph combinator will invoke the inner synchronizers in the order that they appear in the declaration of the synchronizer. For example, if both `ejb` and `entityBean` are changed, `EJBNameEqual` will invoke `SGet<key="Name">` first.

Switch Combinator The switch combinator represents the union of relations. The following code shows a synchronizer constructed using a switch combinator:

```
switch {
 graph{moduleRef, modules}{
  var=module;
  DGet(moduleRef, modules, module);
  NotNull<defaultValue=0>(module);
 };
 graph(moduleRef, modules){
  Equal<value=null>(moduleRef);
 };
}
```

This synchronizer ensures a reference to a module being valid. It contains two inner synchronizers, both of which are constructed using the graph combinator. The first synchronizer ensures that if we query the dictionary `modules` with `moduleRef`, we are ensured to get a `module` object which is not `null`. The second synchronizer ensures `moduleRef` is `null`. The union of the two synchronize ensures the reference `moduleRef` is always valid, that is, either `moduleRef` is `null` or `moduleRef` exists in `modules`.

The switch combinator works by invoking the inner synchronizers one by one. The state of the switch synchronizer includes the current values of the data items and the index of the synchronizer invoked the last time. The *sync* function of the switch combinator first find the synchronizer used in the last time and invokes its *sync* function. If

the function succeeds, the switch combinator returns. Otherwise it will invoke the *resync* functions of the other inner synchronizers in the order they appear in the declaration, and return the result of the first succeeded function. If all inner synchronizer fails, the function returns $\perp$ to report a failure. The *resync* function of the switch combinator just invokes the *resync* functions of the inner synchronizers one by one and return the result of the first succeeded function.

For example, suppose the current value of `moduleRef` is 1, and the input modification is ⟨`nomod, {1->null}, nomod`⟩. That is, the reference is no longer in the dictionary and is invalid. The switch combinator first invokes *sync* function of the first inner synchronizer, which is the synchronizer used in the last synchronization. This inner synchronizer first invokes `DGet` to propagate a `!null` modification to `module`, and then fails because `NotNull` fails. After that, the *resync* function of the second inner synchronizer is invoked, and `moduleRef` is set to `null`.

In MOF [OMG02] there is a special reference called *containment reference*, which indicates one object is contained in another object respect to the reference. When the referenced object is deleted, we should also delete the contained object. This kind of reference can be simulated using the following synchronizer.

```
ContainmentReference<attribute> = switch {
  graph(childObj, parentObjs) {
    var=ref, parentObj;
    SGet<key=attribute>(childObj, ref);
    DGet(ref, parentObjs, parentObj);
    NotNull<default=0>(parentObj);
  };
    graph(childObj, parentObjs) {
      Equal<value=null>(childObj);
  };
}
```

Map Combinators The combinators we have seen so far are static, which means the data items to be synchronized by inner synchronizers are statically determined. Sometimes we need to dynamically synchronize newly created data items or remove existing items according to modifications on the data items. For instance, we may want to synchronize two dictionaries by using an inner synchronizer s to synchronize every two items in the two dictionaries. We introduce two map combinators for this task.

The two map combinators synchronize a sequence of dictionaries both by matching the items in the dictionaries and synchronizing the matched tuple with an inner synchronizer. The difference is how they match the items. The `emap` combinator matches items by key. Items are matched only if they are mapped by the same key in the dictionaries. This is suitable for dictionaries whose keys containing significant information, like sequences. The `smap` combinator matches two dictionaries by the inner synchronizer s. Two items are matched if they are successfully synchronized by s. This is suitable for dictionaries whose keys containing insignificant information, like sets.

For example, the following code construct synchronizers using `emap` and `smap`, respectively.

```
s1=emap<sync=Id, dicts=2>
s2=smap<sync=Id, factories=[f1, f2]>
```

The inner synchronizers for both combinators are `Id`. The combinator *emap* also takes a parameter to specify the number of dictionaries to synchronize. The combinator *smap* also takes a list of key factories in case that it need to create new keys when a suitable match cannot be found for some keys. The combinator *emap* does not need a factory list because it just copies keys from one dictionary to the other and does not create new keys.

Suppose the current values for the two dictionaries are both empty dictionaries. With an input modifications of `{1->"a"}`, `{2->"a", 3->"b"}`, synchronizer $s1$ will produce `{1->"a", 2->"a", 3->"b"}`, `{1->"a", 2->"a", 3->"b"}` where the value mapped by the same key are made identical. Synchronizer $s2$ will produce `{1->"a", 2->"b"}`, `{2->"a", 3->"b"}`, where the keys 1 and 2 are matched because they are modified to the same value, and a new key 2 is created in the first dictionary because no proper match can be found for the key 3 in the second dictionary.

Sometimes the inner synchronizer may need some shared data items that cannot be obtained

from the two dictionaries. For example, when we are synchronizing an EJB and an entity bean, we need the module dictionary to obtain information for the `moduleName` attribute. In this case, we resynchronize all data items whenever the shared data items are modified by an inner synchronization, so that all items in the dictionaries are consistent with the shared data item.

For example, the following synchronizer maintains a containment reference between two dictionaries using the `ContainmentReference` synchronizer we have created:

```
ReferenceMaintainer<attribute> = emap {
  sync = ContainmentReference
    <attribute=attribute>;
  dicts = 1;
}
```

The `dicts` parameter is set to 1, which means the first data item synchronized by `ContainmentReference` is a dictionary to be iterated and the second data item is a shared data item. All objects in the first dictionary should have a valid containment reference, or the objects is deleted from the dictionary by being set to `null`.

4.4 The Synchronizer for the EJB Modeling Tool

Now we have seen all the primitive synchronizers and the combinators. Let us construct the synchronizer for the EJB Modeling Tool. The code for the synchronizer is shown in Figure 3.

The final synchronizer is `Main`, which synchronizes the three dictionaries of EJBs, modules and entity beans using three inner synchronizers. The first inner synchronizer `ModulesAndNames` mapped the modules dictionary into a dictionary of module names to be used by `TwoViewMaintainer`. The second inner synchronizer `ReferenceMaintainer` maintains the containment reference `Module` of the EJB objects. The last synchronizer `TwoViewMaintainer` maintains the mappings between the two views. This synchronizer maps between the EJB dictionary and the entity bean dictionary while the module name dictionary is used as a shared item. The inner synchronizer is further a switch between three synchronizers.

`Persistent` deals with persistent EJBs. The `Persistent` attribute of the EJB object is required to be `true` while other attributes are mapped between the EJB object and the entity bean. `NonPersistent` deals with non-persistent EJBs. The `Persistent` attribute of the EJB object is ensured to be `false` while the entity bean is ensured to be `null`. That is, no entity bean is created for non-persistent EJBs. `Deletion` deals with deleted objects. Both the EJB object and the entity bean are ensured to be equal to `null`. Note in the switch synchronizer `NonPersistent` appears before `Deletion`, which means when an entity bean is deleted, we first try to change the EJB into a non-persistent one, rather than deleting the EJB.

The synchronization behavior can be easily changed by changing the composition of synchronizers and the parameters on synchronizers. For example, if we want the synchronizer to delete an EJB when an entity bean is deleted, we just put `Deletion` before `NonPersistent` in the `switch` combinator. For another example, when users change the `ModuleName` attribute of an entity bean, the current synchronizer will try to find a module whose name is equal to the modified name and move the corresponding EJB to that module. If we want the synchronizer instead to change the name of the original module, we can just set the `changeKey` parameter of the `DGet` synchronizer to `false`.

5 Case Study

As we have constructed the synchronizer for the EJB modeling tool, we would like to see how the synchronizer works in actions. In this section, we use Eclipse Graphics Modeling Framework (GMF) and on-site synchronizers to actually construct the EJB modeling tool. The result interface of the editor has been shown in Figure 1. Besides the two views in Figure 1 there is also a “synchronize” button. When users press the button, the modifications on the two views are synchronized.

GMF is a framework for generating graphical editors. In GMF, users define a meta model and a set of shapes, together with a mapping between elements of the meta model and the shapes, and GMF

```
Persistent = graph(ejb, entitybean, modules) {
  var = moduleRef, ejbName, moduleName, persistent;
  SGet<key="Persistent", dictOverValue=false>(ejb, persistent);
  Equal<value=true>(persistent);
  SGet<key="Name">(ejb, ejbName);
  SGet<key="EJBName", dictOverValue=false>(entitybean, ejbName);
  SGet<key="ModuleName", dictOverValue=false>(entitybean, moduleName);
  SGet<key="Module">(ejb, moduleRef);
  DGet<changeKey=true, dictOverValue=false, keyFactory=MaxInteger>
    (moduleRef, modules, moduleName);
}

NonPersistent = graph(ejb, entitybean, modules) {
  var = persistent;
  SGet<key="Persistent", dictOverValue=false>(ejb, persistent);
  Equal<value=false>(persistent);
  Equal<value=null>(entitybean);
}

Deletion = graph(ejb, entitybean, modules) {
  Equal<value=null>(ejb);
  Equal<value=null>(entitybean);
}

TwoViewMaintainer = smap <
  sync = switch{
    Persistent;
    NonPersistent;
    Deletion;
  },
  factories = [MaxInteger, MaxInteger]
>

ModulesAndNames = emap <
  sync = switch {
    SGet<key = "Name", dictOverVal=dictOverVal>;
    graph(a, b){Equal<value=null>(a);Equal<value=null>(b);};
  },
  dicts = 2
>

Main = graph(ejbs, modules, entitybeans) {
  var = modulenames;
  ModulesAndNames(modules, modulenames);
  ReferenceMaintainer<attribute="Module">(ejbs, modules);
  TwoViewMaintainer(ejbs, entitybeans, modulenames);
}
```

Figure 3. The Synchronizer for the EJB Modeling Tool

generates a graphical editor that uses the shapes to edit models defined by the meta model. A simplified structure of GMF-generated editors is shown in Figure 4. The model is responsible for maintaining the data defined by the meta model and the view is responsible for displaying the data using the predefined shapes. When users make modifications on the view, the view modifies the model, and when model is modified, it updates the view.

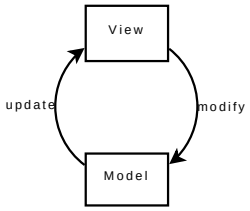

Figure 4. The Structure of a GMF Editor

GMF requires the mappings between meta model elements and shapes to be of one-to-one mapping, that is, the graphical view and the model cannot be heterogeneous. As a result, although we can generate multiple heterogeneous views using GMF, the data of these views cannot be synchronized.

We fill the missing synchronization part using on-site synchronizers, and the structure of our editor is shown in Figure 5. We define different models for different views, and uses GMF to generate multiple independently running single-view applications. The synchronization part is shared among all these applications. When a view is modifying its model, the modification is captured by a model listener. This can be achieved through the notification mechanism provided by GMF. When users press the "synchronize" button, the modifications recorded in the model listener will be inputted into the synchronizer, and the synchronizer will produce the synchronized modifications. Finally, the synchronized modifications are applied to the models through a modification applier.

As mentioned in Section 1, the synchronization part can directly use the internal addresses of objects to interact with other part, rather than defining key attributes for objects. Here we have two choices to convert objects into dictionaries. The first one is that we directly use the addresses of

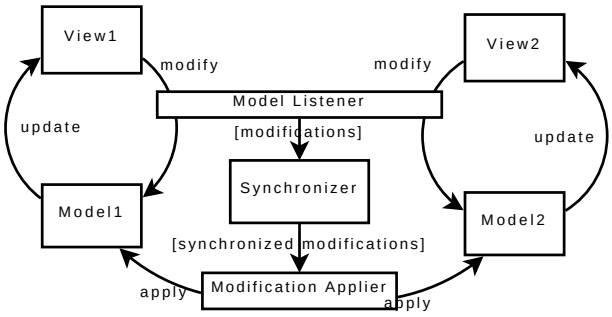

Figure 5. Synchronizing Multiple Views in GMF Editors

objects as keys of the objects in the dictionaries. The second one is that we use integers as the keys for objects, and keep a bijective mapping between the integers and the addresses of objects. The first method is more efficient, but it requires the objects and the states of synchronizers be serialized in the same way when we save the file. Because GMF uses XMI to store objects, we choose the second method in our construction of the EJB tool.

The EJB editor is constructed using the structure in Figure 5. A deployment view editor and a persistent view editor are generated from two sets of definitions using GMF. The synchronizer between the two views is constructed using the code in Figure 3, and the model listener and the modification applier are manually constructed from 340 lines of Java code. The whole development takes no more than ten hours.

To test our synchronizers work correctly, we have performed dozens of experiments on the editor, and all experiments gave desirable results. In the following we describe a few experiments related to the important properties of on-site synchronizers.

In the first experiment, we created an entity bean on the persistent view with `EJBName="User"` and `ModuleName="SignOn"`. After synchronization, an EJB named `"User"` and a module named `"SignOn"` was added to the deployment view. The EJB belongs to the module and its `Persistent` attribute is set to `"true"`. The two views are now consistent, embodying the propagation property.

In the second experiment, we pressed the synchronization button without modifying anything.

The two views remained the same, satisfying the stability property.

In the third experiment, we modified the description text of the `SignOn` module. After synchronization, the modified text, together with other parts of the views, remained the same, satisfying the preservation property.

In the forth experiment we tested the ability of handling conflicting modifications. We modified the `EJBName` attribute of the entity bean to `"x"` and renamed the EJB to `"y"`. When we synchronized, we got a message saying that there were conflicting modifications. We solved the conflict by renaming the EJB to `"x"`, and synchronized again. This time the synchronization passed without errors.

6 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 offsite incremental synchronizer, medini QVT v1.1.2 [ikv], which is a state-of-art implementation of the model transformation standard [Obj05]. 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 EJBs and modules, where every 100 EJBs belong to a module and the attributes are randomly assigned. Then we synchronize to get a consistent set of entity beans.

In the first set of experiments we randomly choose a set of EJBs, 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 EJBs and increases linearly when we increase the size of modifications.

Table 1. Modifying the `Name` Attribute

Mod. Size	Number of EJBs	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 EJBs. This is probably because QVT works offsite. When synchronizing, medini QVT has to recheck 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 first changes the `changeKey` parameter of the `DGet` synchronizer to `false`, that is, when we modify the `ModuleName` attribute of an entity bean, the name of the corresponding module will be changed. Then we randomly choose an entity bean object and modify its `ModuleName` attribute to a new value. Because more than one entity bean can belong to the same module, 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 propagate modifications within one model.

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 synchroniza-

tion 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.

7 Related Work

The mainstream work of off-site synchronization is work on bidirectional transformation [FGM+07, KH06, KW07, LHT07, Obj05]. 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 [Ste07]. Bidirectional transformation cannot directly support modifying the two artifacts at the same time. Benjamin and et al. [PSG03] 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 data in software applications. Two among them are Triple Graph Grammars (TGGs) [KW07], a model transformation approach applying early work in graph grammars to modeling environments [NNZ00, AKRS06], and QVT relations [Obj05], the standard of model transformation. The two approaches have been proved structurally similar by Greenyer and Kindler [GK07]. They are both rule-based, and can support incremental synchronization by re-checking applied rules. Part of our design of on-site synchronizers, like the `smap` combinator, is inspired by TGGs. On the other hand, we try to stay less declarative than TGGs, so that we can provide finer control over synchronization behavior to users.

Some researchers focus on the on-site maintenance of view consistency, which is a typical application of on-site synchronization. Amor and et al. [AHM95] design a declarative language which focuses on bijective mappings between views and provides powerful support to expressions. Some other work [GHM98, FGH+94] 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 [XLH+07] 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.

8 Discussion and Future Work

In this paper we have defined several on-site synchronizers and combinators, and have used them to construct an EJB modeling application. However, several issues still need attention before on-site synchronizers can be widely used. Here we discuss four of these issues.

Memory Consumption To achieve an efficient synchronization of application data, on-site synchronizers keep some information as their internal states. Keeping these states requires extra memory. Currently, to synchronize 6000 EJBs, we need about 110m memory, which is about twenty times higher than just keeping the objects without synchronizers. The memory consumption may be reduced by sharing of data. The current states of synchronizers are mainly the copies of data they synchronize. If we just keep references to the original data instead of keeping duplicated copies, a lot of memory can be saved. We leave this engineering task for future work.

Totality The current synchronizers are not total, which means, synchronizers may fail even if there is no conflicts among modifications. To see how this happens, consider the following synchronizer.

```
graph(k, d, v) {
  vars=k0;
  DGet<changeKey=true>(k0, d, v);
  Id(k, k0);
}
```

Suppose the input modifications are ⟨`2, nomod, "newV"`⟩. Since both inner synchronizers are con-

nected to modifications, they are invoked in the order they appear in the declaration. When `DGet` is invoked, it will try to find a new key for the changed value. However, if the new key is different from 2, the synchronization will fail when the `Id` is invoked.

This failure can be avoided by changing the order of `Id` and `DGet`. As a result, the current synchronizers require programmers to carefully consider the synchronization behavior when they define synchronizers, and sometimes the synchronization behavior is not easy to analyze.

We plan to overcome this problem by designing some high level languages that is total, so that users can program synchronizers by considering only the consistency relations over application data.

Termination The current synchronizers are not always ensured to terminate. Suppose `AddOne` is a synchronizer over two integers a and b, and ensures $a + 1 = b$. One example of a non-terminating synchronizer is as follows.

```
graph{d1, d2}{
  map<sync=AddOne, dicts=2>(d1, d2);
  map<sync=AddOne, dicts=2>(d2, d1);
}
```

When users add a new integer x to `d1`, the first `map` will insert a new integer $x + 1$ to `d2`, and then the second `map` will be invoked and insert $x + 2$ to `d1` and $x - 1$ to `d2`. After that, the first `map` will be invoked again, and will insert two more items to the dictionaries. This process will repeat and will not stop.

Termination is a more fundamental problem to programming, and non-terminating programs can also be constructed in other synchronization approaches [KW07, Obj05] and common programming languages. If we want to keep synchronizers always terminating, we may have to greatly reduce the expressiveness of synchronizers, which conflicts with our original goal. Therefore we choose not to ensure the termination of all synchronizers, and rely on programmers to create terminating synchronizers.

Conflicts Reporting The current synchronizers only report the existence of conflicts. A more preferable way is to report the locations of conflicts so that users know where to solve the conflicts. This can be possibly achieved by extending a modification with a source location, which records the original location of the modification. When a synchronizer propagates a modification from one data item to another data item, it also copies the source location part to the new modification. However, for complex synchronizers it is sometimes difficult to tell where the modification is propagated from. For example, for a `DGet` synchronizer, when the key and the dictionary are both modified, it is difficult to say the modification propagated to the third data item is from the key or from the dictionary. We plan to further investigate this issue and design an algorithm for propagating the source locations of modifications.

References

[AC06] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *Proc. 9th MoDELS*, pages 692–706, 2006.

[ACar] Michal Antkiewicz and Krzysztof Czarnecki. Design space of heterogeneous synchronization. In *Proc. 2nd GTTSE*, to appear.

[AHM95] Robert Amor, John Hosking, and Warwick Mugridge. A declarative approach to inter-schema mappings. In *Modelling of Buildings Through Their Life-Cycle: Proc CIB W78/TG10 Conference*, 1995.

[AKRS06] 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.

[FGH+94] 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.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

[GHM98] John Grundy, John Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, 1998.

[GK07] Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In *Proc. 10th MoDELS*, pages 16–30, 2007.

[ikv] ikv++ technologies. medini QVT homepage. `http://www.ikv.de/index.php?option=com_content&task=view&id=75&Itemid=77`.

[KH06] Shinya Kawanaka and Haruo Hosoya. biXid: a bidirectional transformation language for XML. In *Proc. 11th ICFP*, pages 201–214, 2006.

[KW07] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, University of Paderborn, June 2007.

[LHT07] Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of XQuery. In *Proc. PEPM*, pages 21–30, 2007.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *In Proc. 22nd ICSE*, pages 742–745, 2000.

[Obj05] Object Management Group. MOF QVT final adopted specification. http://www.omg.org/docs/ptc/05-11-01.pdf, 2005.

[OMG02] OMG. MetaObject Facility specification. http://www.omg.org/docs/formal/02-04-03.pdf, 2002.

[PSG03] Benjamin C. Pierce, Alan Schmitt, and Michael B. Greenwald. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003.

[Ste07] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proc. 10th MoDELS*, pages 1–15, 2007.

[XHT+08] Yingfei Xiong, Zhenjiang Hu, Masato Takeichi, Haiyan Zhao, and Hong Mei. On-site synchronization of software artifacts. Technical Report Report METR 2008-21, Department of Mathematical Informatics, University of Tokyo, April 2008.

[XLH+07] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *Proc. 22nd ASE*, pages 164–173, 2007.

[YKW+08] Yijun Yu, Haruhiko Kaiya, Hironori Washizaki, Yingfei Xiong, and Zhenjiang Hu. Enforcing a security pattern in stakeholder goal models. In *Proc. 4th QoP Workshop*, 2008.