GRACE TECHNICAL REPORTS

Beanbag: Operation-based Synchronization with Intra-Relations

Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, Song Hui, Hong Mei

GRACE-TR-2008–04 December 2008

Beanbag: Operation-based Synchronization with Intra-Relations[*]

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

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

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

[3]GRACE Center
National Institute of Informatics, Tokyo, Japan
hu@nii.ac.jp

December 13, 2008

Abstract

Modern development environment often involves data with complex relationships. When users update some part of the data, we need to synchronize the update to make all data consistent.

Bidirectional transformation supports synchronization by propagating updates between two replicas, from one replica to the other or vice versa. However, replicas are often constrained with intra-relations, and we may have to propagate updates within one replica. Such kind of propagation is not well supported by current bidirectional transformation.

In this paper we propose Beanbag, a new language for operation-based synchronization of data with intra-relations. Beanbag treats inter-relations and intra-relations in a unified way, and allows users to define these relations declaratively. A Beanbag program can be compiled into a synchronizer, which takes user updates as input and produces new updates to make all data consistent. We have implemented Beanbag in Java, applied it to several applications and tested its performance by experiments.

[*]This is a bug-fix version which is slightly different from the published version

1 Introduction

Modern development environment often involves data with complex relationships. For example, different diagrams in a UML model are related, and the UML model and its generated code are related. When users update some part of the data, we need to transform and propagate the update to other part to make all data consistent.

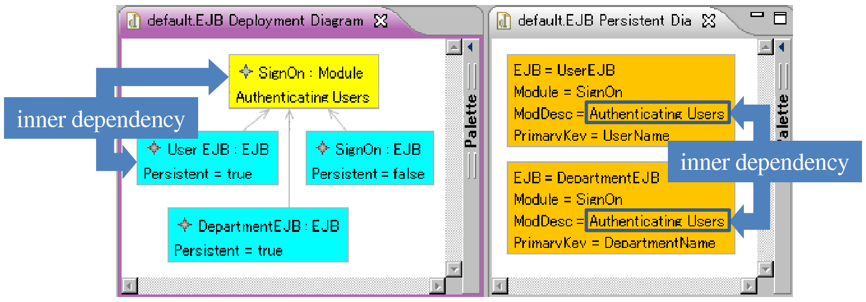

Figure 1: An EJB Modeling Tool

Figure 1 gives a concrete example, which depicts a simple Enterprise JavaBeans (EJBs) modeling tool. The tool provides two types of editable diagrams: the deployment diagram and the persistent diagram. The deployment view shows how EJBs are organized into modules, while the persistent diagram shows a list of persistent EJBs (entity beans). In Figure 1 there are three EJBs: `SignOnEJB`, `UserEJB`, and `DepartmentEJB`, all of which belong to a module `SignOn`. The persistent attributes of `UserEJB` and `DepartmentEJB` are `true`, indicating they are entity beans and are listed in the persistent diagram. In many places the two diagrams are related, such as the EJB names shared in the two diagrams. When users update an EJB name in one diagram, the corresponding name in the other diagram also need to be updated to make the two diagrams consistent. The process of propagating updates between heterogeneous data is called *heterogeneous synchronization* and the software components to perform such synchronization are called *synchronizers* [ACar].

Bidirectional transformation [Ste07, FGM$^+$07] provides support for heterogeneous synchronization. A bidirectional transformation contains a consistency relation $R \in A \times B$ between two heterogeneous data formats, a forward function $f : A \times B \to B$ and a backward function $g : A \times B \to A$ [Ste07]. Given two inconsistent replicas of data in A and B, bidirectional transformation changes one replica according to the other, or vice versa, to make them consistency. Typical bidirectional transformation languages include QVT Relations [Obj08] and TGGs [KW07].

Bidirectional transformation works well when the two replicas can be freely modified. However, in the real world many application data may also be constrained with intra-relations and update of one location may require updates of other location. Two examples of intra-relations are indicated with thick arrows in Figure 1. In the deployment diagram the `SignOn` module and its EJBs are connected with associations, and when the module is deleted, we need to either delete all EJBs, or at least, remove the invalid associations. The intra-relations in the persistent diagram concerns two related module description. When one is modified, we need to modify the other to make sure they are consistent. As bidirectional transformation only propagates updates between the two replicas, it cannot well handle data with intra-relations.

Intra-relations impose many new challenges and call for some non-trivial improvements on bidirectional transformation. We summarize the challenges as follows.

- First and foremost, intra-relations call for a new synchronization interface. Consider the intra relation in the persistent diagram, the system needs to know which part of the data is modified to correctly overwrite the unmodified data with the modified data, and this information is not available in the state of data. Furthermore, the synchronization directions are not limited to forward and backward, as the propagation along intra-relations can be of many directions once.
- Second, intra-relations often have mutual effects with inter-relations. Consider the intra-relation in the deployment diagram. If an EJB is deleted due to the deletion of a module, we should also delete the corresponding entity bean in the persistent diagram. The intra-relation in the persistent diagram is more interesting. From the persistent diagram alone, the two module descriptions are not related. The two descriptions are related because they correspond to the same module element on the deployment diagram. Because of such mutual effects, we need an unified way to connect intra-relations and inter-relations together.
- Third, the complexity of intra-relations usually leads multiple options in synchronization. For example, when a module is deleted in the deployment diagram we have options to remove only the associations or to also remove the child EJBs. We need provide means for users to specify these options and therefore precisely control the synchronization behavior.

In this paper, we study the language support of synchronization with intra-relations over dictionary-based data. We focus on dictionaries to make our language compact. Dictionaries can be used to represent many different data structures as studied by Pierce et al. [PSG03][1], and we will show how to encode object model like the EJB modeling tool in dictionaries.

In our study we adopt *operation-based synchronization*. Traditional bidirectional transformation can be classified as *state-based synchronization* [PSG03], as the current state of data are taken as input as input. Unlike the state-based way, the operation-based synchronization takes update operations applied on data and produces new update operations to make all data consistent. In this way we address the first challenge: the modified parts can be deduced from updates and propagation is not restricted to a specific direction.

[1]Dictionaries are known as edge-labeled trees in [PSG03]

The result of our study is called Beanbag, a new language for operation-based synchronization with intra-relations. The name of Beanbag is contrived from a traditional Asia game that keeps several beanbags consistent. Our contributions can be summarized as follows.

- We propose the Beanbag language for users to describe the consistency relation over data. The Beanbag language treats intra-relations and inter-relations in a unified way, and users can freely compose the relations using logical operators like conjunction and disjunction. In addition, we provide finer control over synchronization behavior, while keeping Beanbag program easy to write.
- We give a clear execution semantics to convert the consistency relations described in Beanbag to a synchronizer. The implemented synchronizer takes user updates as input, and produces new updates to make all data consistent. The synchronization satisfies the stability, preservation and consistency properties [XLH$^+$07], guaranteeing a correct synchronization behavior.
- We have implemented Beanbag in Java [Bea], evaluated its performance by experiments, and evaluated its practicability by several case studies. The experiments show that Beanbag is much faster than an incremental implementation of QVT relations[ikv], and the case studies show that Beanbag works well in practical applications. We also discovered an extra benefit of Beanbag during the development of applications: we can get rid of the key attributes used in many bidirectional transformation approaches [Obj08, BFP$^+$08] due to the tight integration between operation-based synchronizers and applications.

The rest of the paper is organized as follows. Section 2 gives an overview of Beanbag while Section 3 presents detailed semantics. Section 4 describes our implementation and the experiments we have conducted. Section 5 gives two case studies that we have conducted and explains how we get rid of the key attribute. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 Overview

To illustrate the features of Beanbag and give a concrete impression of what our system can do, let us consider how to develop a synchronization tool for the EJB example in Figure 1. To develop the tool, the only thing we need to do is to declare the consistency relations over diagrams in the Beanbag language. The execution semantics of Beanbag could turn the program into a synchronizer and synchronize the diagrams through updates. The full Beanbag program used in this section can be found in Appendix A and Appendix B.

To define the consistency relation, we first need to define the data structure of the diagrams in Figure 1. The two diagrams consist of three types of objects:

module objects and EJB objects in the deployment view, and entity bean objects in the persistent view. By assigning each object with a unique key, we can represent the EJB objects in Figure 1 by the following *dictionary* (a mapping from keys to objects):

```
{1↦{"Name"↦"SignOnEJB",
     "Persistent"↦false,
     "Module"↦4},
 2↦{"Name"↦"UserEJB",
     "Persistent"↦true,
     "Module"↦4},
 3↦{"Name"↦"DepartmentEJB",
     "Persistent"↦true,
     "Module"↦4}}
```

It has three EJB objects with keys 1, 2, and 3. Each attribute of an object (`"Name"`, `"Persistent"`, or `"Module"`) points to either a primitive value or a key (4 in this example) to another object. It is worth noting that dictionary, though being simple, plays an important role in our language; it not only enables concise description of objects (and their relations) , but also provides a direct means to access and update each object.

Similarly, we can represent modules in Figure 1 as follows (we omit the definition of the entity beans in the persistent view).

```
{4↦{"Name"↦"SignOn",
     "Description"↦"This module is ..."}
```

After defining the data structure by dictionaries, we step to declare the consistency relation using our synchronization language Beanbag. We want to declare two types of relations: the inter-relations between diagrams and the intra-relations within one diagram. In Beanbag these two types are captured in a unified way. First, we describe the top relation among the EJBs, the modules, and the entity beans.

```
1 main(ejbs,modules,entitybeans) {
2  containmentRefs⟨attr="Module"⟩(ejbs,modules);
3  for[ejb,entitybean]in⟨ejbs,entitybeans⟩{
4   persistent(ejb,entitybean,modules) |
5   nonPersistent(ejb,entitybean) |
6   {ejb=null;entitybean=null}
7  }
8 }
```

It says that (1) EJBs have a containment relation with modules (line 2) in the sense that if a module is deleted then all EJBs belonging to the module should be deleted, (2) for each pair of `(ejb, entitybean)` in `(ejbs, entitybeans)` respectively, they have either the `persistent` relation (line 4) or the `nonPersistent` relation (line 5), or both of them are deleted at the same time (line 6). The relation `containmentRefs` is provided by the standard library and we will see its definition in Section 3.3. The `persistent` relation and `nonPersistent` relation are relations to be refined.

This small piece of code exposes some interesting features of Beanbag. First, the parameters of `main` (`ejb`, `modules` and `entitybeans`) are all treated equally. We do not have to specify `ejb` and `modules` are in the same diagram nor impose any propagation direction among them. Second, the intra-relations and the inter-relations are treated equally. Line 2 captures the intra-relation on the deployment diagram and the rest lines capture the inter-relations between the two diagrams. Both of them are described using the same set of constructs. The intra-relation on the persistent diagram is implicitly captured in the `persistent` relation. Let continue to declare the `persistent` relation.

```
 9 persistent(ejb,entitybean,modules){
10  var moduleRef,moduleName,module;
11  ejb."Persistent"=true;
12  entitybean."EJBName"=ejb."Name";
13  entitybean."ModuleName"=moduleName;
14  moduleRef=ejb."Module";
15  findBy⟨attr="Name"⟩(modules,moduleName,moduleRef);
16  module."Description"=entitybean."ModuleDescription";
17  module=!modules.moduleRef;
18 }
```

It says that the EJB should be persistent (line 11) with the same name as that of the entity bean (line 12), while the entity bean has an module name (line 13), and if we find out the module by the name from all modules (line 13), the result should just be the module referred by the EJB (line 14). The `findyBy` is also a standard library relation described in Section 3.3. The entity bean also has a module description, which is equal to the description of the module referred by the EJB (line 16-17). The “!” in line 17 is used to tune the synchronization behavior and will be introduced in Section 3.3. Note here we related the module description in the entity bean and that in the module. If several entity beans shares the same module name, the module descriptions in these entity bean will be indirectly related and result in the intra-relation in the persistence view.

The `nonPersistent` relation is similarly defined as follows.

```
18 nonPersistent(ejb,entitybean) {
19  ejb."Persistent"=false;
20  entitybean=null;
21 }
```

As seen above, our synchronization program is of high modularity in the sense that small synchronization relations can be glued to form a big one. Beanbag is simple to use; the basic synchronization statements in Beanbag are the *equality* “=” to describe primitive relation, and synchronization statements are combined together with “;” (called *conjunction*) and “|” (called *disjunction*), and the *for* statement to apply a relation inside dictionaries. The details about the language are given in Section 3.3.

That is all we have to do. After the synchronization relations are given, our system can automatically produce a synchronizer for keeping the two views consistent. To be concrete, let us see how the synchronizer interacts with users in action.

A synchronizer has two procedures: `initialize` and `synchronize`. Before synchronization, we need to initialize the synchronizer so that it knows what data will be synchronized. The `initialize` procedure is also useful in the execution of disjunction and the for statement, as will be seen in Section 3.3. We invoke the `initialize` procedure with three initial values and three initial updates for the three parameters of `main`.

```
Initial values:  [{}, {}, {}]
Initial updates: [void, void, void]
```

The initial values are all empty dictionaries, indicating no objects at the beginning. The initial updates are used to control the behavior of initialization and will be explained in details in Section 3. Here we just pass void, indicating no initial update. After invocation, the `initialize` procedure will return the following result to indicate that we do not have to modify the initial values.

```
Output updates: [void, void, void]
```

After initialization, we use the `synchronize` procedure to synchronize user updates. Suppose users have added a new entity object. This update can be described as follows. The dictionary structure indicates the location of updates and “!” indicates the value at the location is replaced by the value following “!”.

```
Input: [void,
        void,
        {1↦{"EJBName"↦!"UserEJB",
             "ModuleName"↦!"SignOn"}}]
```

To synchronize the updates, we invoke the `synchronize` procedure of the synchronizer with the updates as input, and the synchronizer will produce the following output updates. The output updates are expected to be applied on data to make the data consistent. In the updates an EJB object and a module object are created and their attributes are properly set. Note the input update is also kept in the output.

```
Output: [{2↦{"Name"↦!"UserEJB",
              "Persistent"↦!true,
              "Module"↦!3}},
         {3↦{"Name"↦!"SignOn"}},
         {1↦{"EJBName"↦!"UserEJB",
              "ModuleName"↦!"SignOn"}}]
```

Suppose users have modified the name of the EJB to `"User"`. The input and output of the synchronization are as follows. The EJB name of the corresponding entity bean is changed.

```
Input:  [{2↦{"Name"↦!"User"},void,void}]
Output: [{2↦{"Name"↦!"User"},void,
         {1↦{"EJBName"↦!"User"}}]
```

For another example, suppose users want to delete the module. The input and output of the synchronization are as follows. The EJB belonging to the module is deleted, and so does the corresponding entity bean.

```
Input:  [void,{3↦!null},void]
Output: [{2↦!null},{3↦!null},{1↦!null}]
```

As discussed in Section 1, along intra-relations usually we have multiple options in synchronization. When users delete the module, we can also only remove the associations to achieve consistency. This can be done by replacing the statement in line 2 with the following statement.

```
  nullableRefs⟨attr="Module"⟩(ejbs, modules);
```

That is, we use `nullableRefs` to replace `containmentRefs`, and `nullableRefs` is also a relation defined in the standard library. It changes the `Module` reference in EJBs to `null` when the referred module is deleted. As now the `Module` reference can be `null`, we also need to change the `persistent` to handle `null` references.

```
persistent(ejb, entitybean, modules) {
 var moduleRef, moduleName, module;
 ejb."Persistent" = true;
 entitybean."EJBName" = ejb."Name";
 {
  {ejb."Module"=null;
   entitybean."ModuleName"=null;
   entitybean."ModuleDescription"=null;} |
  {moduleName=entitybean."ModuleName";
   moduleName≠null;
   moduleRef=ejb."Module";
   findBy⟨attr="Name"⟩(modules,moduleName,moduleRef);
   module."Description"=entitybean."ModuleDescription";
   module=!modules.moduleRef;}
 }
}
```

It says either the module reference in the EJB and the module-related attributes in the entity bean are all `null`, or, like the original one, the attributes in the entity bean are related to the module referred by the EJB. Now if we repeat the above steps again, the synchronizer will produce the following output which remove only the association.

```
Input:  [void,{3↦!null},void]
Output: [{2↦{"Module"↦!null}},
         {3↦!null},
         {1↦{"ModuleName"↦!null}}]
```

Another interesting example of multiple options in synchronization is that an entity bean is deleted, we can either remove the corresponding EJB or just set its persistent attribute to `false`. The current Beanbag program will set the persistent attribute to `false`, but we can change this behavior by simply swapping the two statements in line 5 and line 6. The new program will give higher priority to deletion over the `nonPersistent` relation, and will delete the corresponding EJB when an entity bean is deleted.

3 The Beanbag Language

We have seen how Beanbag works in Section 2, and this is summarized in Figure 2. A Beanbag program is compiled into a synchronizer which consists of two

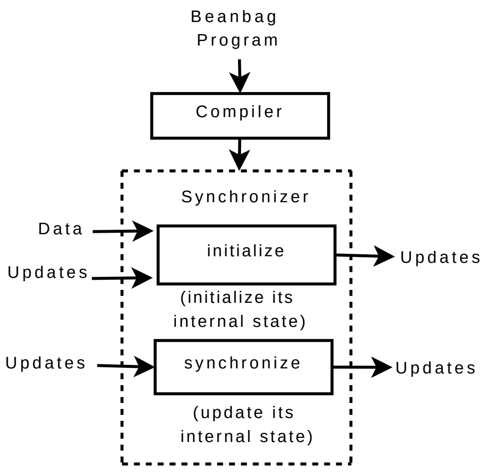

Figure 2: How Beanbag works

procedures. The `initialize` procedure initializes the synchronizer. It takes initial data and updates as input and produces new updates to make the input data consistent. The `synchronize` procedure synchronizes data. It takes user updates and produces new updates to make data consistent.

To ensure synchronizers behave in a reasonable way, we propose three properties to constrain the behavior of synchronizers. The three properties are adapted from our previous work [XLH$^+$07] on state-based synchronization. The first property, *stability*, requires that when users modify no data item, the synchronizer should modify no data item, either. The second property, *preservation*, requires the output updates should contain the input updates. In other words, when users modify a data item, for example, to 2, the synchronizer should not modify the data item to any other value different from 2. The third property, *consistency*, requires the synchronizer to produce correct updates to make all data consistent. After we apply updates produced by a synchronizer to data, we can know the data are consistent. This property is formerly defined as *propagation* in our previous publication [XLH$^+$07].

The `synchronize` procedure satisfies all the three properties. The `initialize` procedure satisfies preservation and consistency. It does not satisfy stability because the input data may not be consistent.

3.1 Data

Beanbag uses a small but general set of data types to represent data. In this way our language remains compact, but many other data types can also be mapped to our data types and synchronized by our synchronizer. Figure 3 shows the definition of data in BNF. The *italic symbols* are non-terminals and the `Sans Serif` symbols are terminals. `STRING`, `NUMBER` and `BOOLEAN` are lexical tokens of common meanings.

There are two types of data. The rule `primitive` generates unstructured *prim-*

$$\begin{array}{lll}
\textit{value} & ::= & \textit{primitive} \mid \textit{dictionary} \\
\textit{primitive} & ::= & \mathsf{null} \mid \mathsf{STRING} \mid \mathsf{NUMBER} \mid \mathsf{BOOLEAN} \\
\textit{dictionary} & ::= & \{\textit{entries}\} \\
\textit{entries} & ::= & \textit{entry} \mid \textit{entry}, \textit{entries} \\
\textit{entry} & ::= & \textit{primitive} \mathrel{-\!\!>} \textit{value}
\end{array}$$

Figure 3: Definition of data

$$\begin{array}{lll}
\textit{update} & ::= & \mathsf{void} \mid \textit{prim\_update} \mid \textit{dic\_update} \\
\textit{prim\_update} & ::= & !\textit{primitive} \\
\textit{dic\_update} & ::= & \{\textit{update\_entries}\} \\
\textit{update\_entries} & ::= & \textit{update\_entry} \mid \textit{update\_entry}, \textit{update\_entries} \\
\textit{update\_entry} & ::= & \textit{primitive} \mathrel{-\!\!>} \textit{update}
\end{array}$$

Figure 4: Definition of updates

itive values, including numbers, strings, booleans and a `null` value. The rule `dictionary` generates structured dictionaries, which map primitive values (keys) to other values. A key-value pair is called an *entry*. If a key is mapped to `null`, it means that the key does not exist in the dictionary. That is, `{"a"↦null}` and `{}` are both empty dictionaries. We write $d.k$ for the value to which the dictionary d maps the key k if no confusion will be caused.

Many other data structures can be mapped to this dictionary-based data types. We have seen how to map object models to the data types in Section 2. For more examples, a set can be mapped to a dictionary by assign each value a unique key or using the in-memory address of each value as its key. A sequence can be mapped to a dictionary by simulating the implementation of a linked list, that is, `["a", "b", "c"]` can be represented as `{"value"↦"a", "next"↦{"value"↦"b", "next"↦{"value"↦"c"}}}`. More detailed discussion on mapping different data types to dictionaries can be found in [PSG03].

3.2 Updates

We capture users' updates by locations of updates and results of updates. The syntax of the language for describing updates is shown in Figure 4. The non-terminal `primitive` has been defined in Figure 3.

The syntax is similar to that of values. An update can be either *prim_update* – an update on primitive values, *dic_update* – an update on dictionaries, or void – an update indicating that nothing has been changed by users. An update on primitive values is just to replace old value with a new value. For example, if we apply `!2` to a value `1`, the value `1` will change to `2`. An update on dictionaries maps keys to updates, where the updates are expected to be applied on the values mapped by the same keys. If a key does not exist in a *dic_update*, we consider the key is mapped to void. For example, if we apply an update `{1↦!null, 2↦!"a", 3↦{"x"↦!"y"}}` to a dictionary `{1↦"n", 2↦"m", 4↦"z"}`, we will get `{2↦"a", 3↦{"x"↦"y"}, 4↦"z"}`.

If two updates change the same location to different values, we say that the two updates conflict. For example, `{1↦!"a"}` and `{1↦!"b"}` conflict but `{1↦!"a"}`

Table 1: The rules of detecting conflicts

	void	*prim_update*$_2$	*dic_update*$_2$
void	false	false	false
prim_update$_1$	false	*result*$_1$	true
dic_update$_1$	false	true	*result*$_2$

where $result_1 \equiv (prim\_update_1 \neq prim\_update_2)$
$result_2 \equiv (\exists k : dic\_update_1.k$ and $dic\_update_2.k$ conflict$)$

Table 2: The rules of merging updates

	void	*prim_update*$_2$	*dic_update*$_2$
void	void	*prim_update*$_2$	*dic_update*$_2$
prim_update$_1$	*prim_update*$_1$	*prim_update*$_2$	*dic_update*$_2$
dic_update$_1$	*dic_update*$_1$	*prim_update*$_2$	*dic_update*$_3$

where $\forall k : dic\_update_3.k \equiv (dic\_update_1.k$ merges $dic\_update_2.k)$

and `{2↦!"b"}` do not. The complete rules for detecting conflicts are summarized in Table 1. The cells in the table show whether the update from the left conflicts with the update from the top. Here we write *dic_update*.k for the update to which *dic_update* maps k.

In many cases, users perform a sequence of updates instead of a single one, which requires us to merge a sequence of updates into a single update. Table 2 shows the rules for merging two updates. The left of the table shows the update applied earlier, the top of the table shows the update applied later, and the cells show the merged results. For example, merging `{1↦"a", 2↦"b"}` with `{1↦"c", 3↦"d"}` results in `{1↦"c", 2↦"b", 3↦"d"}`. Non-conflicting updates are commutative under merging, e.g., merging `{1↦"a"}` with `{2↦"b"}` has the same result as merging `{2↦"b"}` with `{1↦"a"}`.

3.3 The Beanbag Language

Every Beanbag program has two types of semantics. First, the *relation semantics* defines the consistency relation over data. Second, the *execution semantics* defines how to propagate updates to satisfy the relations. In many cases, users only need to know the relation semantics to understand a Beanbag program. However, if they want to know more about how updates are synchronized, or want to write a Beanbag program to control the synchronization behavior, they need also to understand the execution semantics. In this section we introduce the two types of semantics.

3.3.1 The Relation Semantics

Table 3 summarizes the relation semantics of the core language constructs by examples. The core language constructs are divided into five groups. Named relations allow us to organize programs modularly, and `main` declares the entry relation of a program. Expressions are the primitive relations we can use in Beanbag programs, and most of their relation semantics is as same as what we can expect from their

Table 3: The relation semantics of the core language constructs

Named relations		
0	`get⟨key="age"⟩(d,k) {d.key=k}`	declaring a named relation `get`, where `key` is a generic parameter with a default value `"age"`.
1	`main(d,k){get(d,k)}`	defining the entry relation of the program.
Expressions		
2	`a=b`	the two variables, `a` and `b`, are equal.
3	`a=1`	the variable `a` is equal to the constant 1.
4	`d."k"=v`	`v` is equal to the value to which the dictionary `d` maps the constant `"k"`.
5	`!d.k=v`	`v` is equal to the value to which the dictionary `d` maps the variable `k`.
6	`d.!k=v`	`v` is equal to the value to which the dictionary `d` maps the variable `k`.
7	`!d.!k=v`	`v` is equal to the value to which the dictionary `d` maps the variable `k`.
8	`a==b`	the two variables, `a` and `b`, are equal.
9	`a≠b`	the two variables, `a` and `b`, are not equal.
10	`get⟨key="name"⟩(a, b);`	calling a named relation `get`
Conjunction		
11	`{var c; c=a."name";b."name"=c;}`	exists `c`, where `c=a."name"` and `b."name"=c` both hold.
Disjunction		
12	`{a=`**`true`**`|a=`**`null`**`}`	either `a=true` or `a=null`
For statement		
13	**`for`**`[a,b]`**`in`**`[dictA,dictB] {a=b}`	for any key `k`, we have `dictA.k=dictB.k`.
14	**`for`**`[⟨k1,v1⟩,⟨k2,v2⟩]` **`in`** `⟨d1,d2⟩ {rel(k1,v1,k2,v2);}`	exits a bijective relation R over keys, where $\forall$ `⟨k1, k2⟩` $\in R$, the inner relation `rel(k1, d1.k1, k2, d2.k2)` holds.

syntax. Conjunction, disjunction and the `for` statement allow us to compose small relations using common logical operators. Conjunction represents the logical operator "and". Disjunction represents the logical operator "or". The `for` statement applies a relation to each sequence of entries in a sequence of dictionaries. We can require the sequence of entries to have the same key (line 13) or allow them to be freely matched (line 14). In addition, we can choose to capture the key component (line 14) or not (line 13).

Some language constructs have different syntax but have the same relation semantics. One example is `a=b` and `a==b`. Another example is `!d.k=v`, `d.!k=v` and `!d.!k=v`. This is because for the same consistency relation, we may have more than one way to synchronize the updates. The different language constructs represent different ways of synchronization, which allows us to finely control the synchronization behavior. The details of the difference will be introduced in the next section.

3.3.2 The Execution Semantics

When compiled, a beanbag program is converted into a synchronizer, and this is achieved by converting each language construct in the program into a synchronizer and combining them together. The expressions are turned into primitive synchronizers. Conjunctions, disjunctions and the `for` statements are turned into composite synchronizers where small synchronizers are glued together to form a bigger synchronizer. Every synchronizer, either primitive or composite, has the `initialize` procedure and the `synchronize` procedure. In this section we define the execution semantics of Beanbag by describing how these language constructs are turned into synchronizers and how the two procedures of these synchronizers behave.

Expressions Expressions are converted to primitive synchronizers in compilation. In synchronization, these synchronizers try to produce updates that change minimal parts on data and also satisfy the three properties.

As an example, let us consider a simple synchronizer, `a=b`, shown in line 2 of Table 3. This synchronizer synchronizes variables `a` and `b` by keeping them equal. The variables `a` and `b` must be declared as parameters of a named synchronizer or declared in other outer constructs. In the `synchronize` procedure, the synchronizer merges the two updates on the variables and returns the merged result. For example, if users update `a` to 1 and do not modify `b`, the procedure will merge the update `!1` and `void`, and returns the merged result `!1` on both `a` and `b`. If the updates on the two variables conflict, the procedure will fail and report an error. In this way we can propagate the update on either variable to the other, i.e., deducing a propagation direction from updates. In addition, the stability, preservation and consistency properties are all satisfied.

The `initialize` procedure of `a=b` is a bit more complex. Because we do not require the input values to be consistent, we may face multiple options in synchronization. When the input values are not equal, we will have two choices: using the value of `a` to overwrite `b` or vice versa. We allow users to customize this behavior through the input updates. Because of the preservation property, the updated part cannot be modified. Then users can use the input updates to mark some unmodifiable part and force the synchronizer to modify other part. When the input updates on the two variables are both `void`, the synchronizer will choose the natural order of assignment, i.e., using the variable at the right of "=" to overwrite the variable at the left. In the case of `a=b`, `a` will be overwritten by `b`. This choice also applies to other expressions connected by "=", and users can make use of this feature to further control the behavior of synchronization. The `initialize` procedure of `a=b` proceeds as follows. It first merges the updates on `a` and `b`, and applies the merged update on `a`. Then it updates `b` to the new `a` and produces the result.

One interesting part is the expressions between line 5 and line 7. These expressions involves three variables, `d`, `k` and `v`, and tries to keep `v` equal to the value in the location to which `d` maps `k`. Because we have three variables, when `v` changes, we can propagate the update to either `d` or `k`, or both. We use "`!`" to indicate the variable to which the update is going to be propagated. The expression !d.k=v propagates the update on v to d, that is, it uses k as a constant value and proceeds like d."k"=v. The expression d.!k=v propagates the update on v to k, that is, it uses d as a constant value and tries to find a key whose corresponding value in d equals the updated v. When such a key cannot be found, it reports a failure. The expression !d.!k=v propagates the update on v to both d and k. Its behavior is similar to d.!k=v, but when a proper key cannot be found, it inserts a new key to the dictionary. In this way we can precisely control what to update and what not to.

For example, suppose the current values on d, k and v are [{1="a", 2="b"}, 1, "a"], and the input updates on d, k and v are [void, void, "c"]. The synchronize procedure of !d.k=v will return an update {1↦"c"} on d, the synchronize procedure of d.!k=v will fail, and the synchronize procedure of !d.!k=v will return an update {3↦"c"} on d and {!3} on k. The new key 3 is generated by a unique key generator to make sure it is different from all existing keys.

The last category of expressions is used to test conditions, as shown in line 8 and 9. In synchronization, such an expression reports a failure with no variable changed when the condition does not hold. These expressions can be used to check constraints on data, and are also useful in disjunctions of synchronizers.

Conjunction Line 11 of Table 3 shows conjunction of synchronizers, which tries to establish the consistency relations of all inner synchronizers. In addition, users can also define inner variables. In the synchronize procedure, the input updates on inner variables are considered as void. In the initialize procedure, the input updates on inner variables are considered as void and the input data on inner variables are considered as null.

The synchronize procedure of conjunction works as follows: whenever the update on a variable changes, the synchronize procedures of the inner synchronizers declared on the variable will be invoked. When no inner synchronizer can be invoked, the procedure returns. When there are more than one inner synchronizers that can be invoked, the one appearing earliest in the declaration will be invoked first. As an example, the following table shows an invocation of the synchronizer in line 11.

input updates:		
a	b	c
void	{"name"↦!"x"}	void
b changes, and we invoke b."name"=c:		
void	{"name"↦!"x"}	!"x"
c changes, and we invoke a."name"=c:		
{"name"↦!"x"}	{"name"↦!"x"}	!"x"

Next let us consider the initialize procedure. The easiest way to implement the procedure is to adopt the similar method of synchronize, invoking the initialize procedure of inner synchronizers when the values or updates on the related variables need to be synchronized. However, this method may cause unnecessary failures. Consider the synchronizer in line 11. Suppose the input values on `a` and `b` are `[{"name"↦"x"}, {}]`, and the input updates are `[void, {"name"↦!"y"}]`. The expected output updates should be `[{"name"↦!"y"}, {"name"↦!"y"}]` because updates have higher priority than values. However, when we invoke the `initialize` procedure of `c=a."name"`, we will get an update `!"x"` on `c`, and when we invoke `b."name"=c`, the procedure will fail because the updates on `c` and `b` conflict.

To solve this problem, we distinguish updates on variables into two levels. The high level is the updates propagated from input updates or the input updates themselves (such as `{"name"↦!"y"}` on `b`). The low level includes the updates propagated from values (such as `!"x"` on `c`) plus the updates in the high level. The updates in the high level can overwrite those in the low level. For each variable, we store one update for the high level and one update for the low level. After we distinguish the updates into two levels, the update `{"name"↦!"y"}` will be on a higher level than `!"x"` and will overwrite `!"x"` without failure.

However, if we design our synchronizers to directly support this kind of two-level synchronization, the semantics of synchronizers would be too complex for both users and language implementers. To keep the language simple, here we propose a novel technique to simulate two-level synchronization using one-level synchronization. Each time we need to invoke the `initialize` procedure of an inner synchronizer, we invoke the procedure twice to give different priorities to the two levels. In the first invoke, we use updates on the low level and store the result on the low level. In the second invoke, we first apply updates on the low level to the values, and pass the new values and updates on the high level to the procedure. The result is stored in the high level, and is also merged into the low level as a later update.

To see how this work, let us consider the previous example again. The initial values and updates on `a` and `c` are shown below.

	values	low level	high level
a	`{"name"↦!"x"}`	void	void
c	`null`	void	void

After the first invocation of `c=a."name"`, the values and updates will become:

a	`{"name"↦!"x"}`	`{"name"↦!"x"}`	void
c	`null`	`!"x"`	void

The second invocation will change no values nor updates because the input values are consistent and the high level updates are `void`. Then we proceed to invoke `b."name"=c`. Before invocation, the value and updates on `b` are:

b	`{"name"↦!"x"}`	`{"name"↦!"y"}`	`{"name"↦!"y"}`

The first invocation will fail because the updates on the low level conflict. However, in the second invocation, an update `!"y"` will be propagated to `c` because the high level update on `c` is void. This update will further be merged into the low level and result is shown below.

c	null	!"y"	!"y"
b	{"name"↦!"x"}	{"name"↦!"y"}	{"name"↦!"y"}

Then `c=a."name"` will be invoked again to further propagate the update to `a`. In this way we simulate multi-level by invoking one-level synchronizers multiple times.

Besides the basic expressions between line 2 and line 10, Beanbag provides more types of expressions as syntax sugars. These expressions are translated into the conjunction of basic expressions at the compiling time. For example, an expression `b."name"=a."name"` will be translated to the code in line 11 of Table 3, and an expression `a."k"==b."k"` will be translated to `{var c,d; c=a."k"; d=b."k"; c==d;}`.

Disjunction Line 12 of Table 3 shows disjunction of synchronizers. Disjunction will try to establish one of the consistency relations of the inner synchronizers. The `initialize` procedure of disjunction will invoke the `initialize` procedure of inner synchronizers in the order that they are declared, and will return the result of the first succeeded procedure as the result of the whole disjunction. The `synchronize` procedure of disjunction will first find the last succeeded inner synchronizer, and invoke its `synchronize` procedure. If the invocation fails, the `initialize` procedure of other inner synchronizers will be invoked, still in the order of declaration.

For example, the following code shows a synchronizer that ensures an object reference to be valid. The `objRef` reference should either exist in the `objs` dictionary or be equal to `null`.

```
nullableRef(objRef, objs){
 objs.!objRef≠null | objRef=null
}
```

For another example, in MOF [OMG02] there is a special reference called *containment reference*, which indicates one object is contained in another object with 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.

```
containmentRef⟨attr⟩(srcObj, tgtDict) {
 { var ref; ref=srcObj.attr; tgtDict.!ref≠null; }
 | srcObj=null
}
```

For Statement The language constructs we have seen so far are used to synchronize static data structures. The `for` statement is used to dynamically establish consistency relations over entries in different dictionaries. Line 13 of Table 3 shows a `for` statement. This statement has the same behavior as `dictA=dictB` over dictionaries. The `for` statement first matches the entries in the two dictionaries by key, and then applies the inner synchronizer `a=b` to the value parts of every two matched entries.

In the `synchronize` procedure of `for`, the system creates a new instance of the inner synchronizer and invokes its `initialize` procedure for newly inserted pairs, and calls the `synchronize` procedure of the existing inner synchronizer for modified pairs. In the `initialize` procedure of `for`, the system creates a new instance of the inner synchronizer and invokes its `initialize` procedure for every two entries.

One use of the `for` statement is to filter some entries in a dictionary of objects. For example, the following synchronizer ensures that `persistentObjs` contains and only contains the objects whose `persistent` attribute is `true` in `objs`.

```
filter(objs, persistentObjs) {
 for[obj, pobj] in [objs, persistentObjs] {
  {obj."persistent"=true;pobj=obj;}
  | {obj."persistent"=false;pobj=null;}
  | {obj=null;pobj=null;}
 }
}
```

Because we consider a key mapped to `null` is a key that does not exist in a dictionary, the synchronizer `pobj=null` can ensure no `pobj` exist in `persistentObjs`. Note the last synchronizer `{obj=null;pobj=null;}` is always needed if we want to handle deletions in `objs`.

The two dictionaries in the above example contains homogeneous objects. Sometimes we need to synchronize two dictionaries containing heterogeneous objects. For instance, in the EJB example we need to match a persistent EJB with an entity bean and synchronize them. We cannot match the objects by key because the objects are created independently and there is no corresponding relationship between their keys. To support this kind of synchronization, we provide another matching mode in the `for` statement: matching by the inner synchronizer. Two entries in the two dictionaries are matched if they are successfully synchronized by the `initialize` procedure of the inner synchronizer. Once two entries are matched, the `for` statement will remember their trace relationship. If any of the two entries are updated, the `synchronize` procedure of the inner synchronizer will be invoked to synchronize them. We use angle brackets around dictionary variables instead of square brackets to indicate matching by the inner synchronization.

So far we only use `for` to synchronize two dictionaries. The `for` statement can also be applied to more than two dictionaries or just one dictionary. In the latter case, it just applies the inner synchronizer to all entries in the dictionary.

Sometimes we also want to use the key part of an entry in synchronization, and it can be done by writing ⟨`keyVar, valueVar`⟩ instead of a single variable in the binding declaration. Because we do not want users to change the key, the variable bound to key will always have a *prim_update* to prevent further modification. Line 14 in Table 3 shows an example of that. An important use of key binding is to obtain the trace between two dictionaries. When entries in two dictionaries are matched by an inner synchronizer, we may want to know which two entries are matched. The following synchronizer shows how to do that.

```
for[⟨k1, v1⟩,⟨k2, v2⟩,trace] in ⟨DictA,DictB,traceDict⟩
{trace."left"=k1;
```

```
  trace."right"=k2;
  v1=v2;}
```

The synchronizer uses the idea of TGGs [KW07]: when two objects are matched, a `trace` object is created and referenced to the two objects. The trace objects are stored in a dictionary `traceDict` for later use.

The inner synchronizer may at times refer to some variables that are not declared in the `for` statement, in such cases we resynchronize all matched entries whenever the outer variables change. For example, the following synchronizer maintains references between two dictionaries using `nullableRef` which we have created:

```
nullableRefs⟨attr⟩(srcDict,tgtDict){
 for [srcObj] in [srcDict]{
  { var tgtRef;
    tgtRef = srcObj.attr;
    nullableRef(tgtRef, tgtDict);}
  | srcObj == null
}}
```

Similarly, the following synchronizer maintains containment references between two dictionaries using `containmentRef` which we have created:

```
containmentRefs⟨attr⟩(srcDict,tgtDict){
 for [srcObj] in [srcDict]
  {containmentRef⟨attr=attr⟩(srcObj, tgtDict)}
}
```

Now we have seen all language constructs in Beanbag. As an example, let us construct the `findBy` synchronizer we have seen in the EJB example.[2] The synchronizer ensures the object to which `d` maps `k` have an `attr` attribute equal to `v`, and when `v` changes, the synchronizer change `k` to locate another object whose `attr` attribute is equal to the changed `v`. To achieve this, we first map `d` to a dictionary `d0` containing only the `attr` attribute, and then use `!d0.!k = v` to find a key mapped to the updated `v` in `d0`.

```
findBy⟨attr⟩(d, v, k) {
 var d0;
 for [a, b] in [d, d0]
  {b = a.attr | {a=null;b = null;}}
 !d0.!k = v;
}
```

3.4 Formalization

So far our discussion on synchronizers is informal. To precisely characterize the behavior of synchronizers, we need more formal definitions. In this subsection we give the formal definitions of synchronizers and the three properties.

Before turning to synchronizers, we need to first define updates. An *update* u defined on some data set D is an idempotent function $u \in D \to D$, that is,

[2]Because this implementation is not efficient on memory usage, in our compiler we use a more efficient Java-based implementation.

$u \circ u = u$. The idempotent property allows us to apply an update twice and get the same result.

After we express updates as functions, we can represent the merging of updates as function compositions. However, we would expect the composite functions still to be updates. To ensure this, we introduce a concept *update set*. An update set U defined on a data set D is a set of updates closed on composition, that is, $\forall u_1, u_2 \in U : u_1 \circ u_2 \in U$. To be simple, we assume each data type D has a corresponding update set, denoted by U_D and $\mathsf{void} \in U_D$.

The conflict and preservation of updates can be tested using function compositions. Two updates u_1, u_2 *conflict* iff $u_1 \circ u_2 \neq u_2 \circ u_1$. An update u_1 is *preserved* in u_2 iff $u_1 \circ u_2 = u_2$.

Now we proceed to define synchronizers. A *synchronizer* s synchronizing n variables on data set D consists of four components:

- a consistency relation $s.R \subseteq D^n$,
- a state set $s.\Theta$,
- a partial function for synchronization $s.\textit{synchronize} \in U_D^n \times s.\Theta \to U_D^n \times s.\Theta$,
- and a partial function for initialization $s.\textit{initialize} \in U_D^n \times D^n \to U_D^n \times s.\Theta$

Given the definition of synchronizer, we can precisely define the properties to characterize the synchronization behavior. Suppose at an arbitrary point of time, the current state of the synchronizer is θ and the current values of the variables are $\langle v_1, \ldots, v_n \rangle$. The three properties are defined as follows.

Property 1 (Stability)
$s.\textit{synchronize}(\mathsf{void} \ldots \mathsf{void}, \theta) = \langle \mathsf{void} \ldots \mathsf{void}, \theta \rangle$

Property 2 (Preservation)
$s.\textit{synchronize}(u_1 \ldots u_n, \theta) = \langle u'_1 \ldots u'_n, \theta' \rangle \implies$
$\quad \forall i \in \{1 \ldots n\} : \; u_i \circ u'_i = u'_i$
$s.\textit{initialize}(u_1 \ldots u_n, d_1 \ldots d_n) = \langle u'_1 \ldots u'_n, \theta'' \rangle \implies$
$\quad \forall i \in \{1 \ldots n\} : \; u_i \circ u'_i = u'_i$

Property 3 (Consistency)
$s.\textit{synchronize}(u_1 \ldots u_n, \theta) = \langle u'_1 \ldots u'_n, \theta' \rangle \implies$
$\quad \langle u'_1(v_1), u'_2(v_2) \ldots u'_n(v_n) \rangle \in s.R$
$s.\textit{initialize}(u_1 \ldots u_n, d_1 \ldots d_n) = \langle u'_1 \ldots u'_n, \theta'' \rangle \implies$
$\quad \langle u'_1(d_1), u'_2(d_2) \ldots u'_n(d_n) \rangle \in s.R$

These properties form a starting point of reasoning the behavior of synchronizers and a specification for implementing the synchronizers. All our experiments show that Beanbag satisfies these three properties. However, to formally prove the three properties we need formal semantics of all synchronizers, which is beyond the scope of this paper.

4 Implementation and Performance Evaluation

We have implemented a compiler for Beanbag. After compilation, the language will be translated to a Java program. There are two ways for users to use the Java program: 1) they can run it in command line and interact with the synchronizer using the syntax in Figure 3 and Figure 4, or 2) they can integrate this program into their Java project and interact with the synchronizer through Java method calls.

Our implementation uses incremental propagation to ensure a short synchronization time. We mainly apply the incremental techniques in two places. One is in conjunction, where we invoke a synchronizer only when a related variable is updated. The other one is in the `for` construct, where we synchronize entries only when they are updated by users.

To test how our incremental strategies work in practice, we experiment with the EJB program in Section 2. We also implement a similar program in QVT relations [Obj08] for comparison. This program captures the inter-relations between the two diagrams. QVT relations is an state-based incremental synchronization language and is also the standard of model transformation. The compiler of QVT relations that we use is medini QVT v1.4.0 [ikv], and our experiments are carried out on a laptop with 2 GHz Intel(R) Core(TM) Duo processor and 2 GB RAM.

The basic idea of our experiments is to carry small updates on a large set of data and see how efficient the synchronization can be. We first generate a large number of EJBs and modules, where every 10 EJBs belong to a module and the attributes are randomly assigned. Then we synchronize to get a consistent set of entity beans.

We carry three sets of experiments, each consisting of experiments on different number of EJBs. In the first set of experiment we randomly choose five entity beans and change their names. In the second set we delete five entity beans. In the third set we insert five entity beans[3].

In all experiments the two synchronization programs produce correct results and the time taken to synchronize is shown in Figure 5. Some sets of experiments take quite close time and their lines overlap in the figure. 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.

The modification and deletion in Beanbag takes very short time and remains almost constant when the data size increases. The insertion has a liner increase with the size of data . The reason is that we need to compare the name of the inserted entity bean with the names of all modules to find out which module the inserted entity bean belongs to. On the other hand, the time of medini QVT is much longer than Beanbag and is mainly related to the number of EJBs. This is probably because QVT relations works in a state-based way. When synchronizing,

[3]In fact, we have carried the fourth set of experiments: inserting five new EJBs. However, medini QVT ran extremely slow in these experiments. It took dozens of minutes to finish one synchronization. We believe this is caused by some implementation defects of medini QVT and do not include this set of experiments in paper.

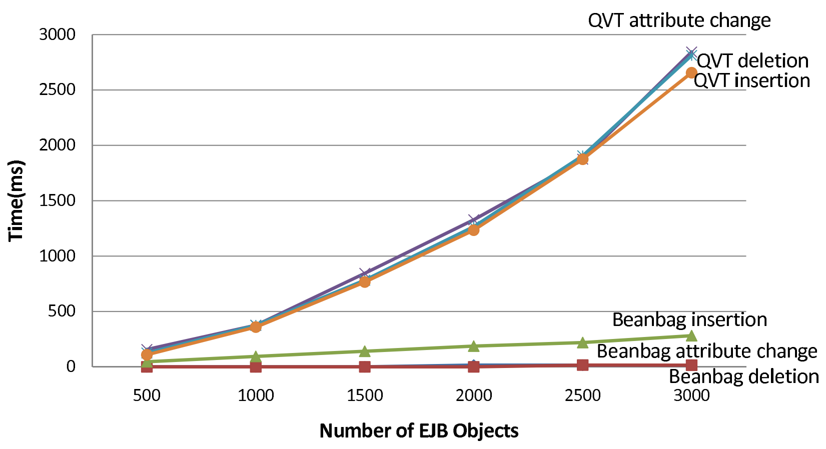

Figure 5: Synchronization time

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.

In summary, the experiments show that the incremental strategy in Beanbag can ensure efficient synchronization over large models. The synchronization time is much shorter than a state-based incremental synchronization tool and should be satisfactory for practical use.

5 Applications

In software engineering, there exist many applications that Beanbag can be applied to. For example, synchronizing multi-views in visual language editors [GHZL06], integration of heterogeneous tools [Tra05], synchronizing software architecture and runtime system [HMY06], and etc. We have successfully applied Beanbag to several case studies. In this section we describe two of them.

EJB Modeling Tool In the first case study we investigate the practicability of Beanbag by constructing a fully functional EJB modeling tool described in Section 1, and Figure 1 is actually a screen snapshot of the tool we have constructed. The main components of the tool are editing components generated by Eclipse Graphical Modeling Framework (GMF) [Ecl08] and a synchronization component generated from the program in Section 2 by Beanbag, and we only write a few hundred lines of Java code to glue them together.

GMF is a framework for generating graphical editors. Given a model definition, a view definition and their mappings, GMF generates a graphical view that reads from and writes to the model. GMF can generate multiple views for one model, but in a quite limited way: the views and the model cannot be structurally different, and multiple views cannot be edited at the same time. As the two views in the EJB modeling tool are structurally different (one hierarchical and one flat), the tool cannot be directly generated by GMF.

Therefore we discard the usual way of generating two views for one model.

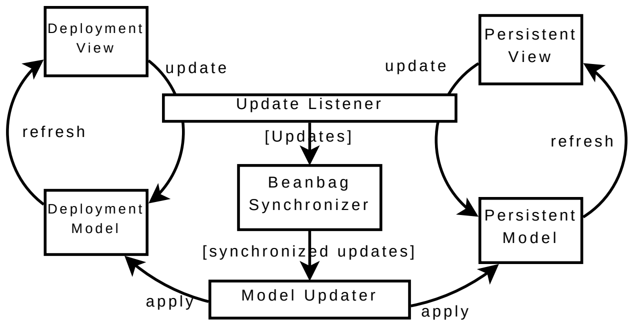

Figure 6: The architecture of the EJB tool

Instead, we generate two editors, each with an independent model. The two models can be structurally different and their consistency is maintained by a Beanbag synchronizer. On the interface side, the two editors are both integrated into Eclipse and act like one application.

The architecture of our implementation is shown in Figure 6. When users update a model through the view, we capture the updates by an update listener. When the two views need to be synchronized (when users explicitly request synchronization or, more automatically, whenever users update a model), we pass the updates to the synchronizer. After synchronization, a model updater updates the models according to the output.

One issue of implementing the update listener and the model updater is how to relate the uniquely generated keys (refer to Section 2) to objects in memory. To achieve it, we keep a bijective mapping between the keys and the in-memory addresses of objects. Because the generated keys are just integers, we can easily save the mapping with models using the serialization support of GMF, ensuring that the object addresses are always valid.

This small technique has great value in practice. To identify objects in state-based synchronization, users are often required to designate some key attributes [Obj08, BFP$^+$08] whose values are unique among all instances. However, based on our experience, many application data do not have a suitable candidate to be a key attribute [YKW$^+$08]. On the other hand, as operation-based synchronizers are tightly integrated into the system, we can directly use the in-memory address and get rid of the key attribute.

Class to RDBMS It should be interesting to see how Beanbag can tackle traditional bidirectional transformation problems. In the second case study, we apply Beanbag to the well-known "Class to RDBMS" transformation [BRST05]. As far as we know, there is yet no proposal claimed for bidirectionalization of the full transformation in the literature.

Figure 7 shows the meta models of the transformation. In the class meta model, a class consists of attributes, where the type of an attribute can either be primitive or another class. There are also associations and parent references between classes. In the RDBMS data model, a table consists of columns and foreign keys, where the

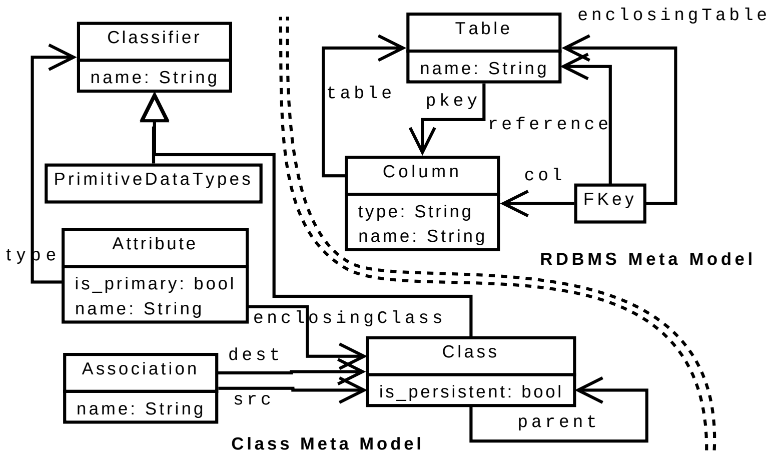

Figure 7: Meta models for Class2RDBMS

type of a column is represented by string.

“Class to RDBMS” is originally described as a unidirectional transformation converting a class model to a relational database (RDBMS) model. To also propagate the updates on RDBMS model back to the class model, we change a few transformation rules so that the correspondence between the two models is clearer. In our rules, each class corresponds to a table. Each attribute corresponds to a column if its type is primitive, and to a foreign key if its type is a class. A parent reference corresponds to a foreign key if it is not `null`. Each association also corresponds to a foreign key. Note this is only a high level outline of the rules. The actual rules are much more complicated involving attribute mapping and synchronization behavior specification.

We have successfully implemented the above rules in Beanbag and have tested it with 65 different updates. Beanbag works well in all tests. Our implementation can be found in Appendix C.

Our implementation captures not only the inter-relations between the two models but also the intra-relations within the models, and Beanbag ensures these relations will be composed together and work together well. For example, if a class on the class model is deleted, then not only its corresponding table but also the attributes of the class and their corresponding columns and foreign keys are all deleted. Furthermore, attributes whose type is the deleted class have their types set to `null` and the corresponding column types are set to `""`. Parent references that refer to the deleted class are set to `null` and the corresponding foreign keys are deleted.

6 Related Work

6.1 Bidirectional Transformation

The mainstream work of heterogeneous synchronization is bidirectional transformation. Typical work includes lens and similar composition-based approaches [FGM$^+$07, LHT07], QVT and similar pattern-based approaches [Obj08, KW07], and etc.

It would be interesting to see whether existing bidirectional transformation languages can support intra-relations without redesign. First, existing bidirectional transformation languages are mainly of state-based semantics, and thus cannot support propagating updates inside a model directly. Although some work on TGGs [GW06] has used operation-based way to increase performance, the semantics is still state-based.

One idea in QVT is to use local applications to handle intra-relations (for example, MOF can handle the containment reference between objects), and let bidirectional transformation focus on inter-relations. This approach works on small examples, but becomes impractical when we consider more complex cases involving mutual effects between intra-relations and inter-relations. A local propagation may affect some inter-relations and require updates on the other model. When we invoke the bidirectional transformation to propagate updates on the other model, it may result in another local propagation in the other model and we may further need more invocations of bidirectional transformation. It is in general difficult to know how many round-trips are needed. Moreover, we have to write and maintain several programs: the local applications and the transformation program. Each time we change one program, we have to make sure all programs consistent. This actually runs back into the situation where heterogeneous synchronization research tries to avoid.

Another idea is to divide the data into smaller pieces which do not have inner constraint, and to write several bidirectional transformation programs to propagate updates among them. This approach works in some cases, but it requires extra work to divide and re-unite the data, which is sometimes not easy. Furthermore, it is also unclear how to invoke the bidirectional transformation programs in a proper order to propagate a specific update.

To sum up, as we could not find a satisfactory solution that can improve existing bidirectional transformation languages to handle intra-relations, we design Beanbag, a new language which handles intra-relations and inter-relations in a unified way.

Many aspects of Beanbag are inspired by bidirectional transformation research. For example, the composition of synchronizers is inspired by lens [FGM$^+$07], and the recording of states is inspired by QVT [Obj08] and some previous discussion [Tra08]. On the other hand, we have made a lot of improvements over bidirectional transformation to support intra-relations: our semantics is operation-based, rather than state-based; the conjunction in Beanbag allows free composition of inner relations, and has much more freedom than sequential composition in lens; we design the `initialize` procedure, which replaces the `create` function in lens and the object creation semantics in QVT to allow synchronization of no predefined direction.

6.2 Other related work

Some researchers focus on the consistency of multi-views in development environments, which is a typical application of Beanbag. Liu and et al. [LHG07] uses a spreadsheet-like mechanism to propagate updates among objects. However, the updates can be propagated only in one direction. Some other work [GHM98, FGH$^+$94] provides general frameworks for view consistency, where users manually write code for propagating updates in each direction. On the other hand, our approach only requires users to describe the consistency relation in the Beanbag language and they automatically get the ability of propagating updates in all necessary directions.

Another branch of related work is about the constraint satisfaction problem [Tsa93] and the transformation approach based on constraint solvers [CS03]. This type of work tries to find a set of values to satisfy a logic expression (in our context, the consistency relation over data). Compared to them, the language constructs in Beanbag are not as declarative as logical expressions, but each has a clear execution semantics, so that we can directly propagate the updates without exploring a state space, ensuring the efficiency of synchronization.

7 Conclusion and Future Work

In this paper we have proposed Beanbag, a language for operation-based synchronization with intra-relations, and have applied it to several applications. Beanbag capture intra-relations and inter-relations in a unified way, and keeps data consistent through propagating updates.

Several issues still need attention before Beanbag can be widely used. Here we discuss two issues. The first one is memory consumption. The current implementation buffers a lot of data to achieve incremental synchronization, which cause an overhead on memory consumption. Nevertheless, much of the consumption can probably be reduced by object sharing. We leave this engineering task for future work.

The second issue is conflict reporting. The current synchronizers only report the existence of conflicts. A more preferable way is to report the updated location causing the conflicts so that users know where to solve the conflicts. This can be possibly achieved by extending an update with a source location, which records the original location from which the update is transformed. We plan to further investigate this issue and design an algorithm for keeping the source locations of updates through propagation.

Acknowledgment

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.

References

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

[Bea] The Beanbag website. `http://code.google.com/p/synclib/`.

[BFP+08] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Proc. 35th POPL*, 2008.

[BRST05] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model transformations in practice workshop. In *Satellite Events at MoDELS*, pages 120–127, 2005.

[CS03] Compuware and Sun. XMOF queries, views and transformations on models using MOF, OCL and patterns. `http://www.omg.org/docs/ad/03-08-07`, 2003.

[Ecl08] Eclipse Consortium. The Eclipse Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/, 2008.

[FGH+94] A. 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.

[GHZL06] John C. Grundy, John G. Hosking, Nianping Zhu, and Na Liu. Generating domain-specific visual language editors from high-level tool specifications. In *Proc. 21st ASE*, pages 25–36, 2006.

[GW06] Holger Giese and Robert Wagner. Incremental model synchronization with triple graph grammars. In *Proc. 9th MoDELS*, pages 543–557, 2006.

[HMY06] Gang Huang, Hong Mei, and Fu-Qing Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Eng.*, 13(2):257–281, 2006.

[ikv] ikv++ technologies. medini QVT homepage. `http://projects.ikv.de/qvt`.

[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.

[LHG07] Na Liu, John Hosking, and John Grundy. Maramatatau: Extending a domain specific visual language meta tool with a declarative constraint mechanism. In *Proc. VL/HCC*, 2007.

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

[Obj08] Object Management Group. MOF query / views / transformations specification 1.0. `http://www.omg.org/docs/formal/08-04-03.pdf`, 2008.

[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.

[Tra05] Laurence Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005.

[Tra08] Laurence Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–126, March 2008.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[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.

Appendix

A Library Relations in Beanbag

```
findBy⟨attr⟩(d, v, k) {
        var d0;
        for [a, b] in [d, d0]
            {b = a.attr | {a=null;b = null;}}
        !d0.!k = v;
}

findValueBy⟨attr⟩(d, attr, k, v) {
  findBy⟨attr=attr⟩(d, attr, k);
  v = !d.k;
}

findByNoChangeD⟨attr⟩(d, v, k) {
        var d0;
        for [a, b] in [d, d0]
            {b = a.attr | {a=null;b = null;}}
        d0.!k = v;
}

containmentRef⟨attr⟩(srcObj, tgtDict){
  {
    var ref;
    ref = srcObj.attr;
    !tgtDict.ref ⟨⟩ null;
  }
  | srcObj = null
}

containmentRefMaintainer⟨attr⟩(srcDict, tgtDict) {
  for [srcObj] in [srcDict]
    containmentRef⟨attr=attr⟩(srcObj, tgtDict)
}

nullableRef(objRef, objs) {objs.!objRef ⟨⟩ null | objRef = null}

nullableRefMaintainer⟨attr⟩(obj, tgtDict) {
  {
    var tgtRef;
    tgtRef = obj.attr;
    nullableRef(tgtRef, tgtDict);
  }
  | obj == null
}
```

B The Beanbag Program for the EJB Modeling Tool

```
include "lib.sync"

persistent(ejb, entitybean, modules) {
       var moduleRef, moduleName, module;
       ejb."Persistent" = true;
       entitybean."EJBName" = ejb."Name";
     moduleName=entitybean."ModuleName";
       moduleRef=ejb."Module";
     findBy⟨attr="Name"⟩(modules,moduleName,moduleRef);
     module."Description"=entitybean."ModuleDescription";
     module=!modules.moduleRef;
}

nonPersistent(ejb, entitybean) {
       ejb."Persistent" = false;
       entitybean = null;
}

main(ejbs, modules, entitybeans) {
       containmentRefMaintainer⟨attr="Module"⟩(ejbs, modules);
       for [ejb, entitybean] in ⟨ejbs, entitybeans⟩ {
            persistent(ejb, entitybean, modules) |
            nonPersistent(ejb, entitybean) |
            {ejb = null; entitybean = null;}
    }
}
```

C The Beanbag Program for Class2Relation

```
include "lib.sync"

assocs2attrs(assocs, attrs, orig_attrs) {
  for [assoc, attr, orig_attr] in ⟨assocs, attrs, orig_attrs⟩ {
    {orig_attr = null; assoc = null; attr = null} |
    {assoc = null; attr = orig_attr; orig_attr ⟨⟩ null;} |
    {
     orig_attr = null;
     attr."enclosingClass" = assoc."src";
     attr."type" = assoc."dest";
     attr."name" = assoc."name";
     attr."is_primary" = false;
    }
  }
}

classes2tables(classes, tables, traces) {
  for [⟨classid, class⟩, ⟨tableid, table⟩, trace]
  in ⟨classes, tables, traces⟩ {
    {
      table."name" = class."name";
      trace."class" = classid;
      trace."table" = tableid;
    } |
    { class = null; table = null; trace = null; }
  }
}
```

```
nullableFindBy⟨attr⟩(d, v, k) {
  findByNoChangeD⟨attr=attr⟩(d, v, k) |
  { v = null; k = null }
}

inTrace⟨attr1, attr2⟩(v1, v2, trace) {
  var key;
  nullableFindBy⟨attr=attr1⟩(trace, v1, key);
  nullableFindBy⟨attr=attr2⟩(trace, v2, key);
}

isPrimitiveType(attr, classifiers) {
  var attrTypeRef, type;
  attrTypeRef = attr."type";
  type = classifiers.!attrTypeRef;
  type."__type" == "primitive";

}

isClassType(attr, classifiers) {
  var attrTypeRef, type;
  attrTypeRef = attr."type";
  type = classifiers.!attrTypeRef;
  type."__type" == "class";
}

attrType2columnType(attrTypeRef, columnType, classifiers) {
  {
    var type, typeName, ref;
    {
      findValueBy⟨attr="name"⟩
        (classifiers, columnType, ref, type)
      | {ref=null;columnType=null;}
    }
    ref = attrTypeRef;
    type."__type" = "primitive";
  } |
  { attrTypeRef = null; columnType = null; }
}

filterClassAttrs(attrs, pattrs, classifiers) {
  for [attr, pattr] in [attrs, pattrs] {
    {pattr = attr; isPrimitiveType(attr, classifiers)} |
    {pattr = null; isClassType(attr, classifiers)} |
    {attr = null; pattr = null;}
  }
}

attrs2columns(attrs, columns, a2cTraces, c2tTraces, classifiers) {
  var primitiveAttrs;
  filterClassAttrs(attrs, primitiveAttrs, classifiers);
  for [⟨attrid, attr⟩, ⟨columnid, column⟩, ⟨traceid, trace⟩]
  in ⟨primitiveAttrs, columns, a2cTraces⟩ {
    {attr = null; column = null; trace = null;} |
    {
      var columnType, attrType, classRef, tableRef;
      columnType = column."type";
      attrType = attr."type";
      attrType2columnType(attrType, columnType, classifiers);

      column."name" = attr."name";
      classRef = attr."enclosingClass";
      tableRef = column."table";
      inTrace⟨attr1="class", attr2="table"⟩
        (classRef, tableRef, c2tTraces);

      trace."attr" = attrid;
      trace."column" = columnid;
    }
  }
}

columnRef2table(columnRef, table, columns, tables) {
  var column, tableRef;
  column = columns.!columnRef;
  tableRef = column."table";
  table = !tables.tableRef;
}

attrs2prims(attrs, a2cTraces, columns, tables, classifiers) {
  for [⟨attrid, attr⟩] in ⟨attrs⟩ {
    {
      var table, columnid;
      attr."is_primary" = false;
      isPrimitiveType(attr, classifiers);
      inTrace⟨attr1="attr", attr2="column"⟩
        (attrid, columnid, a2cTraces);
      columnRef2table(columnid, table, columns, tables);
      table."pkey" ⟨⟩ columnid;
    } |
    {
      var table, columnid;
      attr."is_primary" = true;
      isPrimitiveType(attr, classifiers);
      inTrace⟨attr1="attr", attr2="column"⟩
        (attrid, columnid, a2cTraces);
      columnRef2table(columnid, table, columns, tables);
      table."pkey" = columnid;
    } |
    {
      attr."is_primary" = false;
      isClassType(attr, classifiers);
    } |
    attr = null
  }
}

noNullPKey(tables) {
  for [table] in [tables] {
    table."pkey" ⟨⟩ null | table = null
  }
}

filterPrimitiveAttrs(attrs, pattrs, classifiers) {
  for [attr, pattr] in [attrs, pattrs] {
    {pattr = attr; isClassType(attr, classifiers);} |
    {pattr = null; isPrimitiveType(attr, classifiers);} |
    {attr = null; pattr = null;}
  }
}

attrs2fkeys(attrs, fkeys, c2tTraces, tables, classifiers) {
  var classAttrs;
  filterPrimitiveAttrs(attrs, classAttrs, classifiers);
  for [⟨attrid, attr⟩, ⟨fkeyid, fkey⟩] in ⟨classAttrs, fkeys⟩ {
    {attr = null; fkey = null;} |
    {
      var typeTableRef, attrType, classRef,
        tableRef, name, table;
      typeTableRef = fkey."reference";
      attrType = attr."type";
      inTrace⟨attr1="class", attr2="table"⟩
        (attrType, typeTableRef, c2tTraces);

      name = attr."name";
      fkey."name" = name;
      name ⟨⟩ "__super";

      classRef = attr."enclosingClass";
      tableRef = fkey."table";
      inTrace⟨attr1="class", attr2="table"⟩
        (classRef, tableRef, c2tTraces);

      fkey."col" = table."pkey";
      table = !tables.typeTableRef;
    } |
    {
      attr = null;
      fkey."name" == "__super";
    }
  }
}

filterNonSuperFKeys(fkeys, superFKeys) {
  for [fkey, super] in [fkeys, superFKeys] {
    {fkey = super; fkey."name" == "__super";} |
    {super = null; fkey."name" ⟨⟩ "__super"} |
    {fkey = null; super = null;}
  }
}

supers2fkeys(classes, fkeys, tables, c2tTrace) {
  var supers;
  filterNonSuperFKeys(fkeys, supers);
  for [⟨classid, class⟩, ⟨fkeyid, fkey⟩] in ⟨classes, supers⟩ {
    {
      var generalClassRef, generalTableRef, tableRef, table;
      generalClassRef = class."parent";
      generalClassRef ⟨⟩ null;
      generalTableRef = fkey."reference";
      inTrace⟨attr1="class", attr2="table"⟩
        (generalClassRef, generalTableRef, c2tTrace);
      fkey."name" = "__super";
      tableRef = fkey."table";
      inTrace⟨attr1="class", attr2="table"⟩
        (classid, tableRef, c2tTrace);
      table."pkey" = fkey."col";
      table = !tables.generalTableRef;
    } |
    {
      class."parent" = null;
      fkey = null;
    } |
    {
      class = null;
      fkey = null;
    }
  }
}

typeMapper⟨type⟩(generalObjs, specializedObjs) {
  for [general, specialized] in [generalObjs, specializedObjs] {
    {general."__type" = type; specialized = general;} |
    {general."__type" ⟨⟩ type; specialized = null;} |
    {general = null; specialized = null;}
  }
}

main(classifiers, attrs, assocs, tables, columns, fkeys) {
  var allattrs, classTableTrace, attrColumnTrace, classes;
  typeMapper⟨type="class"⟩(classifiers, classes);
  nullableRefMaintainer⟨attr="parent"⟩(classes, classes);
  assocs2attrs(assocs, allattrs, attrs);
  containmentRefMaintainer⟨attr="enclosingClass"⟩
    (allattrs, classifiers);
  nullableRefMaintainer⟨attr="type"⟩(allattrs, classifiers);
  classes2tables(classes, tables, classTableTrace);
  attrs2columns(allattrs, columns, attrColumnTrace,
    classTableTrace, classifiers);
  attrs2prims(allattrs, attrColumnTrace, columns,
    tables, classifiers);
  noNullPKey(tables);
  attrs2fkeys(allattrs, fkeys, classTableTrace,
    tables, classifiers);
  supers2fkeys(classes, fkeys, tables, classTableTrace);
}
```