MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

A Compositional Approach to Bidirectional Model Transformation

Zhenjiang Hu, Dongxi Liu, Hong Mei,
Masato Takeichi, Yingfei Xiong and Haiyan Zhao

METR 2006–54 Octorber, 2006

A Compositional Approach to Bidirectional Model Transformation

Zhenjiang Hu[1], Dongxi Liu[1], Hong Mei[2], Masato Takeichi[1], Yingfei Xiong[1,2] and Haiyan Zhao[2]

[1]Department of Mathematical Informatics
Graduate School of Information Science and Technology
University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113-8656, Japan
{hu,liu,takeichi}@mist.i.u-tokyo.ac.jp
[2]Institute of Software
School of Electronics Engineering and Computer Science
Peking University, Beijing, 100871, China
{meih,xiongyf04,zhhy}@sei.pku.edu.cn

October 23, 2006

Abstract

Model-driven architecture is a discipline in software engineering that aims to develop, maintain and evolve software by performing model transformations. Many attempts have been made on introducing bidirectionality to model transformation to enable better consistency and traceability between different models. However, the existing approaches are ad-hoc without clear updating semantics, hardly support compositional style of transformation, and provide no means for automatic transformation strategy adaption. In this paper, we propose a new approach to bidirectional model transformations based on a well-defined bidirectional tree-transformation language BiXJ, by giving two novel techniques: representing graphs by trees together with a transformation expressing shareness, and realizing graph transformations by composition of tree transformations. Our approach has two distinguished features: being compositional and supporting automatic adaption of transformation strategy. The experimental results on a non-trivial case study have convinced the promise of the new approach.

1 Introduction

Model-driven architecture (MDA) [Fra03] is a discipline in software engineering that relies on models as first class entities and that aims to develop, maintain and evolve software by performing model transformations. Refinement, abstraction, refactoring, and integration of models are special cases of model transformations that can be found in many areas of software engineering [CH03].

A model transformation is *unidirectional* if it maps a source to a target model but not other way around. A transformation is *bidirectional* if it allows two direction mappings in the sense that both the source and target models can be modified after application of this transformation, and changes may be propagated in either direction.

Many attempts have been made on introducing bidirectionality into model transformation [AK02, BM02, CH03, KS03, GGadRH03, Gro04] to enable better consistency and traceability between different models. Two typical applications of bidirectionality are summarized as follows [BM02, CH03, Gro04]:

- *Synchronization Heterogeneous Views of Models*: Models in software engineering could be manipulated via several views (Figure 1(a)). A view is an abstraction of a model that focuses on a certain aspect like structure or behavior. For example, a feature model can have refinement view, constraint view and interaction view [ZMZ05]. These views are not independent; changes on one view may lead to changes on others. By establishing a bidirectional mapping between each view and the model, we can achieve consistency among all the views; a change on a view is reflected to the model and then propagated to other views.

- *Traceability Links in Software Development*: Bidirectional transformations can record links between their source and target models. These links are useful in analyzing how changing one model would affect other related models, maintaining consistency of models at different stages, and mapping the stepwise execution of an implementation back to its high-level model. As seen in Figure 1(b), if a problem is found in the final model, it is possible to trace back and find the model from which the problem origins.

Since models are graphs in general cases, model transformations are essentially graph transformations, and study on bidirectional model transformations requires study on bidirectional graph transformation. In spite of the usefulness of bidirectional model transformation, there are several important issues that have not been well addressed.

First, there lacks formal *well-formed conditions* on both bidirectional mappings and clear updating semantics on graph-structured view. Although many results [BS81, DB82, HMT04, FGM^{+}05] have been devoted to *updating semantics* for table-structured or tree-structured view, as far as we

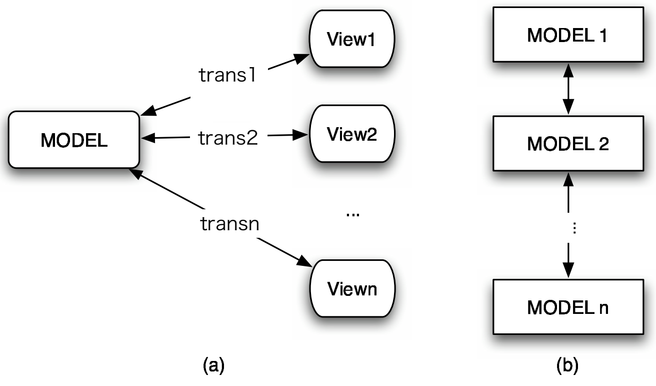

Figure 1: Two typical applications of bidirectional model transformation: (a) synchronization of multiple views; (b) traceability links.

are aware, there is no precise definition of updating semantics for graph-structured view.

Second, most existing MDA tools provide one-step model-to-code transformations, which they use for generating PSMs (Platform Specific Models) from PIMs (Platform Independent Models). But they are inadequate for flexible *composition* of model transformations, where intermediate models may be produced but not necessary to appear to the users. In fact, for large abstraction gaps between PIMs and PSMs, it would be easier to exploit some intermediate models rather than to go straight to the target PSM. Intermediate models are also useful for optimization and tuning. Like compositional approach to software design [KS98], compotional approach to bidirectional model transformation is of high modularity and supports creation of reusable and evolutionable software architectures.

Third, the existing approaches assume that a bidirectional transformation (or transformation strategy) is fixed whenever it is given and the consistency among models is kept only by performing this transformation if one model is changed. As seen in [HMT04], automatic adaption of *transformation strategy* would give high potential for automatic generation of software through model editing.

In this paper, we show that all these problems can be solved with the help of the existing *well-defined* bidirectional tree-transformation languages [Abi99, HMT04, MHT04, FGM$^+$05] that are designed for supporting view updating (i.e. reflecting view modifications back to the original database or XML document) [BS81, DB82, GPZ88, OT94, Abi99]. By well-defined we mean that the language has clear view updating semantics, good support for modular programming, and high description power.

We shall distinguish two different types of model transformations. One is application-independent transformation, and the other is application-specific transformation. In *application-independent transformations*, transformation strategies are derived from common experiences or domain knowledge and are fixed among all applications. Examples in this type include those transformations from UML diagrams to code, from entity-relation diagrams to database tables and synchronization among UML diagrams. In *application-specific transformations*, transformation strategies are derived by developers after analyzing the applications they are working on, usually varying from one application to another. For instance, the transformation from requirement model to software architecture model belongs to this type, in which transformation strategies are given by system analysts after analyzing the particular requirement.

Existing CASE (Computer Aided Software Engineering) tools have already provided supports for many kinds of application-independent transformations, but few of them provide supports for application-specific transformation. As we will see in Section 5, our approach can support these two types of transformations.

The main contributions of this paper can be summarized as follows.

- In theory, we propose a compositional approach to bidirectional model transformations based on a well-defined bidirectional tree-transformation language BiXJ (a Java version of language X [HMT04]), by giving two novel techniques (Section 4): representing graphs by trees together with constraints expressing shareness among tree nodes and realizing graph transformations by composition of tree transformations.

- In practice, we have implemented and tested our approach on the development of a simple document editor under the ABC approach [Mei04], a uniform framework supporting software development from requirement analysis to code generation. From the case study, we get the following experiences: due to bidirectionality of our approach, traceability is better supported during the whole lifecycle of software development; due to the compositional property, evolutional models can be created more easily by exploiting existing transformations for old models; due to BiXJ's support of *computing-as-editing* paradigm [HS95, HMT04], automatic adaption of transformation strategies is allowed; due to BiXJ's support of view dependency [HMT04], the inner dependency in a model can be easily maintained.

The organization of this paper is as follows. We start with a scenario to which our approach will be applied in Section 2. Section 3 introduces language BiXJ. Then, based on BiXJ, we construct a compositional framework for bidirectional model transformation in Section 4, where two techniques of how to realize bidirectional graph transformation from bidirectional tree transformation are introduced. To validate our approach, we provide a nontrivial case study in Section 5. Finally, we explain the related work in Section

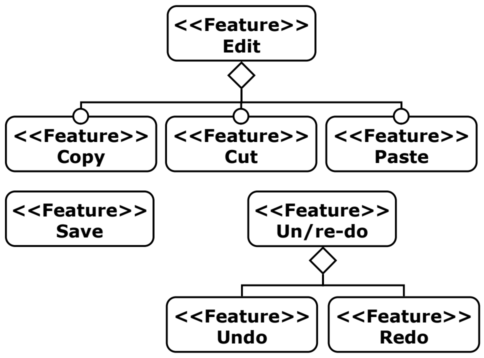

Figure 2: The Feature Model of Simple Editor

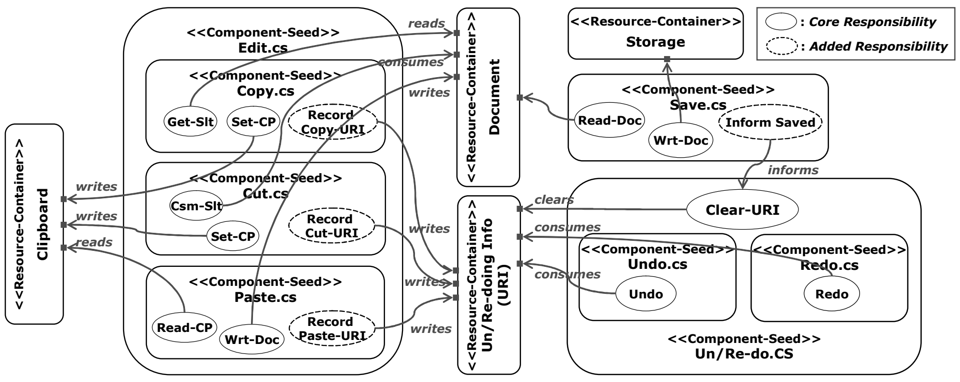
<<Component-Seed>> Edit.cs
<<Component-Seed>> Copy.cs
Get-Slt
Set-CP
Record Copy-URI
<<Component-Seed>> Cut.cs
Csm-Slt
Set-CP
Record Cut-URI
<<Component-Seed>> Paste.cs
Read-CP
Wrt-Doc
Record Paste-URI
<<Resource-Container>> Clipboard
<<Resource-Container>> Document
<<Resource-Container>> Un/Re-doing Info (URI)
<<Resource-Container>> Storage
<<Component-Seed>> Save.cs
Read-Doc
Wrt-Doc
Inform Saved
Clear-URI
<<Component-Seed>> Undo.cs
Undo
<<Component-Seed>> Redo.cs
Redo
<<Component-Seed>> Un/Re-do.CS
reads
consumes
writes
clears
informs
: Core Responsibility
: Added Responsibility

Figure 3: The Responsibility Diagram of Simple Editor

6, and conclude the paper in Section 7.

2 A Scenario of Our Approach

In this section, we give a scenario to which our approach will be applied. This scenario is selected from our previous publication [ZMZY05], which used a simple document editor to illustrate how to do the transformation from CIM (Computation Independent Model) to PIM in ABC approach [Mei04]. Software development in ABC approach is divided into several stages, including: feature-oriented requirement analysis, software architecture modelling, component composition, deployment and maintenance. In [ZMZY05], we have given the CIM as feature model, and the PIM as the responsibility model and the conceptual component model of the editor. Here we advance the development of the editor a little further by giving a detailed design in UML class diagram. The feature model, responsibility model and the UML class diagram are given respectively in Figure 2, 3 and 4.

To establish the traceability and automatic consistency maintenance be-

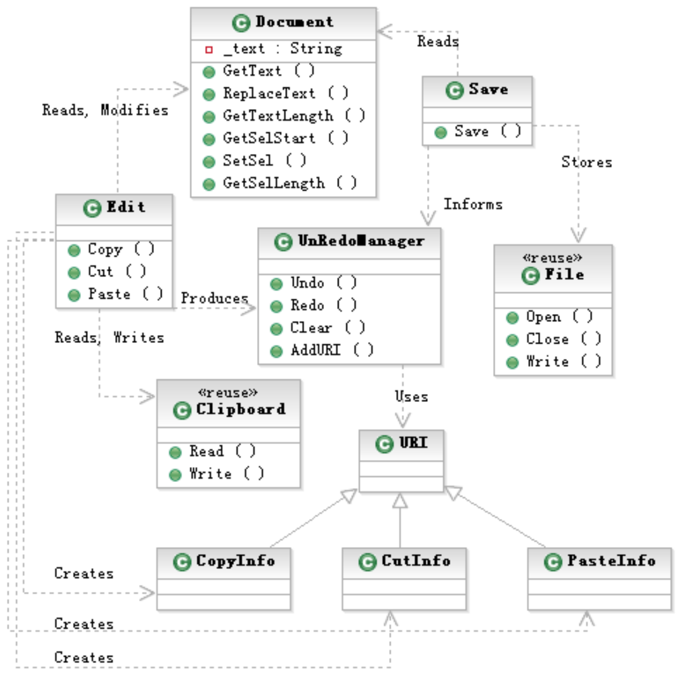

Figure 4: The Class Diagram of Simple Editor

tween these models, we want to use bidirectional transformation techniques to describe the transformations between these models. However, the existing techniques like [BM02, Gro04, FGM+05] cannot satisfy our development requirements in the following aspects:

Model Evolution In an application-specific transformation, the target model is likely to evolve in later stages. For example, in the class diagram shown in Figure 4, there are three methods to get all the information related to a selection in the "Document" class, namely, "GetSelStart", "GetSelLength" and "GetText". Later on, the developer may want to encapsulate the information in a class "Selection" and use one method "GetSelection" instead. The unsatisfied thing is that the existing techniques is not compositional. Hence, even if the target model evolves just a little, we have to write a new transformation to generate it, that is there is no easy way for the existing techniques to generate the evolutional model by exploiting the transformation for the old target model.

Adaption of Transformation Strategy In an application-specific transformation, there always exist lots of components in target models introduced in transformations, which generally reflect users' transformation strategy. For instance, from feature model to responsibility model, the resource container "Document" is introduced by developers in trans-

$$\begin{array}{rcl}
X & ::= & BX \mid XC \mid CM \\
BX & ::= & \texttt{<xconst>}[\textit{element}] \mid \texttt{<xnewroot>}[\textit{tag}] \\
 & & \mid \texttt{<xmove>}[\texttt{<from>}[\textit{path}_0]\ \ \texttt{<to>}[\textit{path}_1]] \\
XC & ::= & \texttt{<xseq>}[X_0 \ldots X_n] \mid \texttt{<xzip>}[X_0 \ldots X_n] \\
CM & ::= & \texttt{<xstore>}[v] \mid \texttt{<xload>}[v] \mid \texttt{<xfree>}[v] \\
\textit{path} & ::= & \texttt{n}_0 \mid \textit{path}\ \texttt{n}_1
\end{array}$$

Figure 5: Syntax of BiXJ

formation. If these elements are modified, maybe developers want to update their transformation strategy, too. However, the existing techniques cannot provide any help since transformations in them are assumed fixed.

Inner Dependency in Model There are a lot of duplications in software models. For example, in Java source code, the same class name may appear in its class declaration, class constructors and some variable declarations with that class as their types. These duplications lead to inner dependencies on each other in that they should always remain same during model transformations. However, the existing techniques couldn't support inner dependency in model.

3 BiXJ: A Bidirectional Transformation Language

Bidirectional model transformation in our work is implemented using BiXJ, which is a bidirectional transformation language over tree-structured XML data. BiXJ is a descendant of language X [HMT04] with clear view updating semantics.

3.1 Syntax of BiXJ

BiXJ syntax is defined in Figure 5, where each transformation construct is represented as an XML element. Here, only some constructs are presented for explaining our approach, and others can be referred to in [HMT04].

For brevity, the end tag of an XML element is omitted and its contents are enclosed by brackets. For example, $<$ `const` $>$*element*$<$ /`const` $>$ is represented as <`xconst`>[*element*]. Source and target XML data also use this style. Each element in an XML document can be reached from root by a unique sequence of integers, as indicated by *path*. Root element is assigned path 0 (it can be omitted in programming); an element has path *path n* if its parent element has path *path* and it is the (n+1)th child of its parent.

For transformation X, $[\![X]\!]_F(s)$ means transforming the source data s into some target data, i.e., forward transformation; $[\![X]\!]_B(s,t')$ means the backward transformation taking as input the original source s and the modified target t', and returning not only the updated source but also an updated transformation. With this feature, backward transformation in BiXJ allows automatic adaption of model transformation strategy.

3.2 Basic Transformations

Intuitively, a basic transformation only performs one step of operation over the input data. Generally, basic transformations are simple to implement.

xconst: Let $X = $ <`xconst`>[*element*].

$$\begin{array}{rcl} [\![X]\!]_F(s) & = & \mathit{element} \\ [\![X]\!]_B(s, \mathit{element}') & = & (s, X') \\ \texttt{where } X' & = & \texttt{<xconst>}[\mathit{element}'] \end{array}$$

If this transformation intends to construct new model components, such as refined or added model components, then after backward execution, it may also be updated to keep track of the users' new transformation strategy.

xrename: Let $X = $ <`xrename`>[*tag*$'$] and $s = $ <*tag*>[*conts*].

$$\begin{array}{rcl} [\![X]\!]_F(s) & = & \texttt{<}\mathit{tag}'\texttt{>}[\mathit{conts}] \\ [\![X]\!]_B(s, \texttt{<}\mathit{tag}''\texttt{>}[\mathit{conts}']) & = & (\texttt{<}\mathit{tag}\texttt{>}[\mathit{conts}'], X') \\ \texttt{where } X' & = & \texttt{<xrename>}[\mathit{tag}''] \end{array}$$

Note that if *tag*$'$ is updated in the target model, this modification is kept in transformation.

xmove: Let $X = $ <`xmove`>[<`from`>[$path_0$] <`to`>[$path_1$]].

$$\begin{array}{rcl} [\![X]\!]_F(s) & = & t \\ [\![X]\!]_B(s, t') & = & (s', X) \end{array}$$

where t is just s except that the element at position $path_0$ is moved to position $path_1$; similarly, s' is just t' except that the element at position $path_1$ is moved to position $path_0$.

3.3 Transformation Combinator

Transformation combinators are used to build complex BiXJ programs by organizing existing transformations together in some way. In this section, we explain combinators `xseq` and `xzip`.

xseq: Let $X = $ <`xseq`>$[X_0 \; ... \; X_n]$.

$$\begin{array}{rcl}
[\![X]\!]_F(s) & = & t \\
[\![X]\!]_B(s, t') & = & (s', \texttt{<xseq>}[X'_0 \; ... \; X'_n]) \\
\texttt{where } t_0 & = & [\![X_0]\!]_F(s) \\
... & & \\
t & = & [\![X_n]\!]_F(t_{n-1}) \\
(s'_{n-1}, X'_n) & = & [\![X_n]\!]_B(t_{n-1}, t') \\
... & & \\
(s', X'_0) & = & [\![X_0]\!]_B(s, s'_0)
\end{array}$$

In the intermediate states, s'_i is the updated source corresponding to t_i $(0 \leq i \leq n-1)$. The *compositional* feature of our approach is embodied in this construct. With this feature, users can divide a complex model transformation into a sequence of simpler transformations, X_0 to X_n. And then, after implementing and testing each simpler transformation, they can simply compose them by `xseq`.

xzip: Let $X = $ <`xzip`>$[X_0 \; ... \; X_n]$, $s = $ <*tag*>$[s_0 \; ... \; s_m]$ and $k = \texttt{max}(n, m)$.

$$\begin{array}{rcl}
[\![X]\!]_F(s) & = & \texttt{<}tag\texttt{>}[t_0 \; ... \; t_k] \\
[\![X]\!]_B(s, t') & = & (\texttt{<}tag'\texttt{>}[s'_0 \; ... \; s'_m], X') \\
\texttt{where } t_0 & = & [\![X_0]\!]_F(s_0) \\
... & & \\
t_k & = & [\![X_k]\!]_F(s_k) \\
\texttt{<}tag'\texttt{>}[t'_0 \; ... \; t'_k] & = & t' \\
(s'_0, X'_0) & = & [\![X_0]\!]_B(s_0, t'_0) \\
... & & \\
(s'_k, X'_k) & = & [\![X_k]\!]_B(s_k, t'_k) \\
X' & = & \texttt{<xzip>}[X'_0 \; ... \; X'_n]
\end{array}$$

The interesting thing about `xzip` is that n and m is not necessarily equal. If $m > n$, then each transformation $X_i(n+1 \leq i \leq m)$ is the identify transformation `xid`; if $m < n$, then each content $s_i(m+1 \leq i \leq n)$ is a null value. This feature of `xzip` is useful when constructing element contents for introducing new model components.

3.4 Execution Context Management

In BiXJ, there are three constructs `xstore`, `xload` and `xfree` for managing a global execution context $\mathcal{C}$, which maps a variable to an XML element and behaves like a stack. These constructs need to execute several operations after forward or backward transformations: $\texttt{push}(\mathcal{C}, [v \mapsto s])$ pushes a mapping to the top of $\mathcal{C}$, $\texttt{pop}(\mathcal{C}, v)$ removes the least recent mapping of variable v from $\mathcal{C}$, $\texttt{update}(\mathcal{C}, [v \mapsto t''])$ changes the least recent mapping of variable v such that it is mapped to value t'' and $\texttt{merge}(t', \mathcal{C}(v))$ merges all modifications on t' and $\mathcal{C}(v)$ into one updated data.

xstore: Let $X = \texttt{<xstore>}[v]$.

$$\begin{array}{rcl} [\![X]\!]_F(s) & = & s \ ; \ \texttt{push}(\mathcal{C}, [v \mapsto s]) \\ [\![X]\!]_B(s, t') & = & (\texttt{merge}(t', \mathcal{C}(v)), X) \ ; \ \texttt{pop}(\mathcal{C}, v) \end{array}$$

xload: Let $X = \texttt{<xload>}[v]$.

$$\begin{array}{rcl} [\![X]\!]_F(s) & = & \mathcal{C}(v) \\ [\![X]\!]_B(s, t') & = & (s, X) \ ; \ \texttt{update}(\mathcal{C}, [v \mapsto t'']) \\ \texttt{where } t'' & = & \texttt{merge}(t', \mathcal{C}(v)) \end{array}$$

xfree: Let $X = \texttt{<xfree>}[v]$.

$$\begin{array}{rcl} [\![X]\!]_F(s) & = & s \ ; \ s' := \texttt{pop}(\mathcal{C}, v) \\ [\![X]\!]_B(s, t') & = & (t', X) \ ; \ \texttt{push}(\mathcal{C}, [v \mapsto s']) \end{array}$$

3.5 Programming Examples

Some useful transformations need not to be defined from scratch, instead they can be defined using the existing transformations. Also as an example of BiXJ programming, we define transformation `xdup` using the transformations introduced before.

$$\begin{array}{rcl} \texttt{<xdup>}[n] & = & \texttt{<xseq>}[\textit{xlist}] \\ \texttt{where } \textit{xlist} & = & \texttt{<xstore>}[v] \ \textit{xlist}_0 \ \texttt{<xfree>}[v] \\ \textit{xlist}_0 & = & \texttt{<const>}[\texttt{<virtual>}[]] \ \textit{xlist}_1 \\ \textit{xlist}_1 & = & \texttt{<xzip>}[\texttt{xload}(v)_1 \ ... \ \texttt{xload}(v)_n] \end{array}$$

This transformation returns n copies of the source XML data wrapped with tag `virtual`. Note that these n copies are mutual dependent because they are loaded from the same value. Modifications on one copy will be propagated to other copies after a backward execution followed by a forward execution. This feature can be used to simulate bidirectional *tree to graph* transformation.

For example, there is a tree model comprising two features: `Edit` and `Copy`, where feature `Edit` is refined by feature `Copy`. In XML format, this tree model is represented as follows:

```
<model>[<refinement>[<feature>["Edit"]
                     <feature>["Copy"]]]
```

However, except for refinement relationship between features, the model should also reflect some constraints between features. And thus, the model becomes a graph since two feature nodes will be *shared* by two relationships. By duplicating the shared nodes, we can simulate the graph with the following tree:

```
<model>[<refinement>[<feature>["Edit"]
                     <feature>["Copy"]]
        <constraint>[<feature>["Edit"]
                     <feature>["Copy"]]]
```

From the above model to the below model, the transformation used is as follows:

```
<xseq>[<xzip>[<xdup>[2]]  <xrmtag>["virtual"]
       <xzip>[<xid>[]
              <xrename>["constraint"]]]
```

Therefore, for example, if `Edit` in one copy is changed to `edit`, then after a backward transformation and a forward transformation, all `Edit` become `edit` since they are actually one node in the graph. In the above code, `xrmtag` removes all "virtual" child nodes of the input XML data.

4 A Compositional Framework for Bidirectional Model Transformations

BiXJ cannot be directly used for model transformation because models are basically graphs rather than trees. Here, we propose two important techniques to implement bidirectional model transformation with BiXJ: representing a graph by a tree together with some constraint capturing node and relation sharing information and realizing model transformation by composition of tree transformations.

4.1 Representing Graph by Tree with Constraint

To use BiXJ in model transformation, we need to translate a graph into a tree, however this representation will lose some information in graph, such as node sharing. To keep these lost information, we can accompany this tree by some equality constraint on its nodes and relations. As a simple yet generic example, consider the graph (the upper part of Figure 6), which consists of three nodes and three edges, and has two kinds of sharing: node sharing (e.g., the node of class B is shared by two relations R1 and R2) and relation sharing (e.g., the relation R1 is shared by two edges). This graph can be represented by a labelled tree shown in the lower part of Figure 6 with the following constraint on tree nodes:

Class A:	l3 = l8 = l12
Class B:	l4 = l9 = l15
Class C:	l5 = l13 = l16
Relation R1:	l7 = l11

Now the problem is how to represent trees with equality constraints so that constraints can be easily maintained when some tree nodes are modified. Since BiXJ is able to simulate *tree to graph* transformation, our idea is to build a meta tree to represent all nodes and relations but without node or relation duplications, and code the constraint in BiXJ. As an example, in Figure 7, the tree in lower part with its constraint (same as the above one) is represented by a meta tree and a BiXJ transformation, shown as follows:

```
<xseq>[
  <xapply>[<path>[0 0] <xdup>[3]]
  <xapply>[<path>[0 1] <xdup>[3]]
  <xapply>[<path>[0 2] <xdup>[3]]
  <xapply>[<path>[1 0] <xdup>[2]]
  <xapply>[<path>[0] <xrmtag>["virtual"]]
  <xapply>[<path>[1] <xrmtag>["virtual"]]
  <xmove>[<from>[0 0] <to>[1 0 0]]
  <xmove>[<from>[0 0] <to>[1 1 0]]
  <xmove>[<from>[0 1] <to>[1 0 1]]
  <xmove>[<from>[0 1] <to>[1 2 0]]
  <xmove>[<from>[0 2] <to>[1 1 1]]
  <xmove>[<from>[0 2] <to>[1 2 1]]
]
```

In the above code, `<xapply>[<path>`[*path*] X] means applying transformation X to the element at position *path* in the input XML data. Suppose that we change the node l3 from A to D in Figure 7. What we want is that the nodes l8 and l12 should be changed to D because all these three nodes correspond to one node in the graph in Figure 6. This is achieved by the following two steps:

1. S_1: Propagating the change to the meta tree by the backward transformation of the above BiXJ code, which causes the node with content A changed to D;

2. S_2: Propagating the change of the meta tree back to the tree by the forward execution of the above BiXJ code, which causes the nodes labelled l8 and l12 to be D, the same as that of node l3.

That is, the constraint among tree nodes can be automatically preserved by $S_1; S_2$, a composition of two BiXJ transformations.

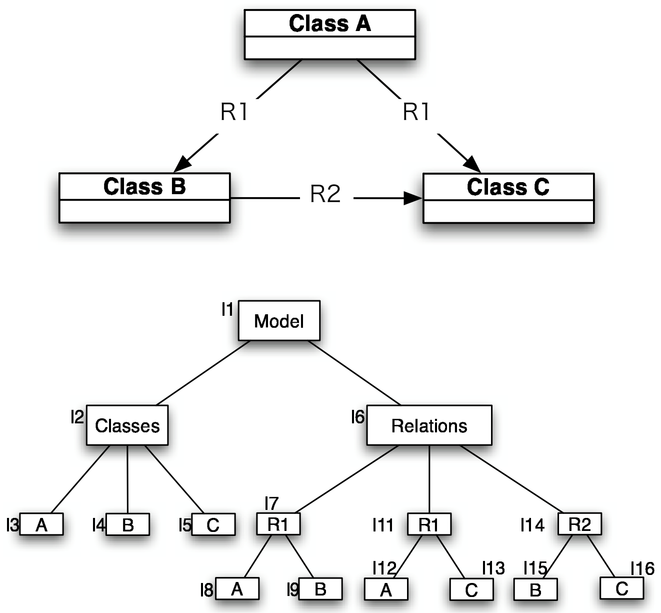

Figure 6: An example of a small graph and its corresponding tree.

In summary, for any graph, we can represent it in our approach by the following steps: 1)Building a meta tree with all graph nodes in one subtree and all graph relations in the other subtree; 2)Duplicating each shared node and relation n times if it is shared n times in graph; 3)Removing all auxiliary virtual tags; 4)Moving one copy of each shared node to the relations involving it. For model graphs exported by Rational Software Architect and feature modelling tool in ABC, which are represented in XML with id attributes for node references, we have implemented this procedure to generate the meta tree and the constraint BiXJ code automatically.

4.2 Realizing Bidirectional Graph Transformation

After coding constraint in bidirectional transformation, we are able to deal with bidirectional transformation between two graphs. Figure 8 depicts how bidirectional graph transformation TG can be realized by composition of tree transformations. During transformation, the source graph G1 is represented by tree t1, and the target graph G2 is represented by tree t2. Tree t1 and its constraint are captured in meta tree $t1'$ and transformation T1, which are what we described in last section. For tree t2, shown in the lower part of Figure 9, there is no newly added relation shared, but a newly added node C2 shared. Hence, T12 only needs to reflect the following constraint by duplicating node C2 (after creating it), and then moving one copy to the

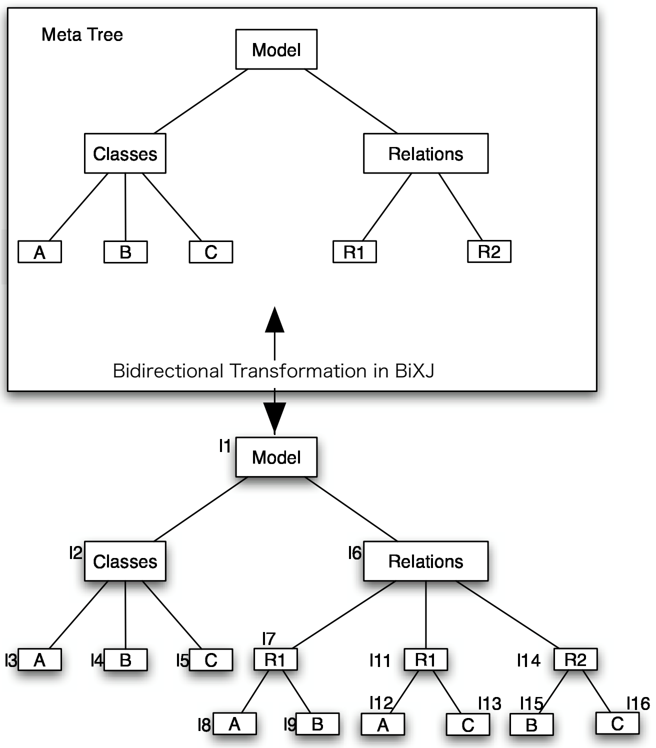

Figure 7: Graph = Tree + Transformation.

relation R2 and R3, respectively.

$$\text{Class C2:} \quad \mathsf{I17} = \mathsf{I20} = \mathsf{I24}$$

For the forward transformation of TG, we compose the forward transformations of T1 and T12, that is $[\![$<xseq>[T1 T12]$]\!]_F$(t1′). On the other hand, for the backward transformation, we need to perform backward transformation of T12 and T1, and then a forward transformation of T1 to get an updated tree t1 with its constraint satisfied, that is $[\![$<xseq>[T1 T12]$]\!]_B$(t1′, t2′) followed by $[\![$T1$]\!]_F$(t1″), where t2′ represents the modified target graph and t1″ is the updated meta tree.

Though we illustrate our approach by an example, this method of achieving graph transformation by composing tree transformations is also applicable to any other graph transformation.

5 A Case Study

To study its applicability to software engineering, we have applied our approach to the development of a simple document editor as described in

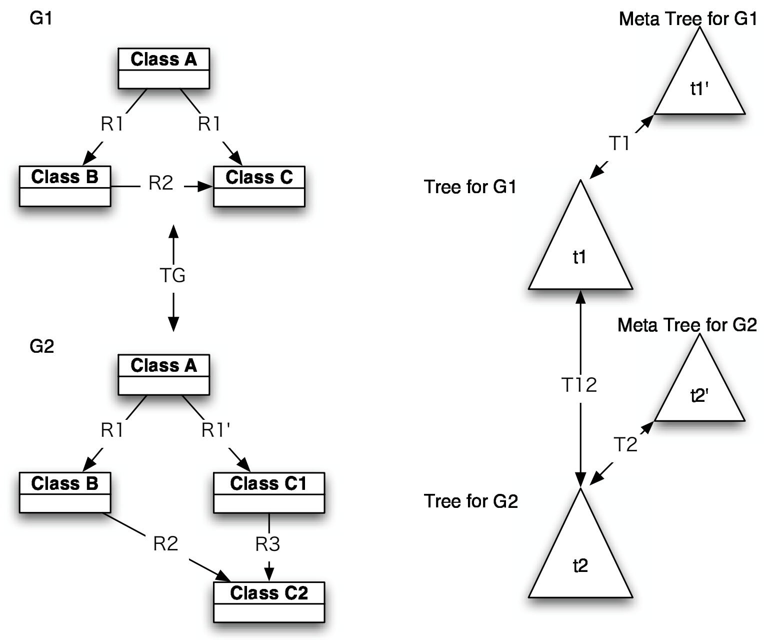

Figure 8: Bidirectional graph-to-graph transformation via bidirectional tree-to-tree transformations.

Section 2 in ABC approach [Mei04]. In this section, we will present some implementation details and the result of this case study.

It should be noted that the editor used here is not a workable software, because it does not contain basic operations like text editing and file opening. However, since our goal is to evaluate the applicability of our approach rather than to develop a workable editor, we think this feature set is complete enough to fulfill our goal.

5.1 Implementation Details

Before writing transformations between models, we first build the models. In our case, the features and responsibilities are modelled using the *feature modelling tool* in ABC approach [Mei04] developed by Institute of Software, Peking University. The software architecture were modelled in UML diagrams using *Rational Software Architect* of IBM. Besides the transformations between the given models, we also need to write the transformation generating Java source code from UML diagrams.

Since BiXJ only works on XML, we have to get the XML representation of these models. Fortunately, the feature modelling tool uses XML as its storage file format and Rational Software Architect has the ability to export and import XML file conforming to the XML Metadata Interchange (XMI)

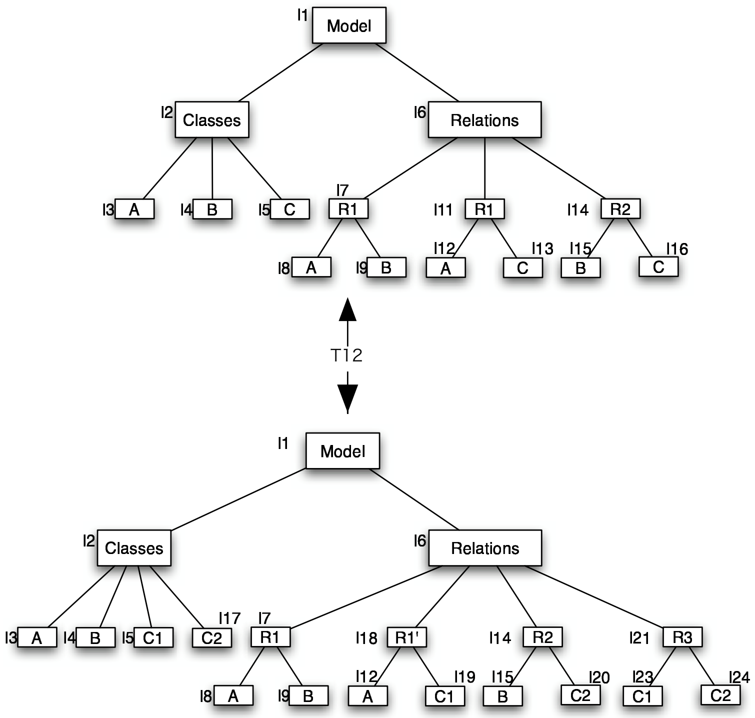

Figure 9: Tree-to-graph transformation.

Specification [Gro05]. Thus we can get the XML representation from these tools and these XML files can be synchronized with their corresponding models.

The problem comes with code. The Java source is stored in plain text format and has no XML representation. To generate Java source code from UML diagrams, we have designed an XML format for Java source code and written a simple program to transform from XML code to Java code. Therefore, the UML diagrams can be transformed to XML code and then transformed to Java code.

Another problem is related with the model itself since most of these models have graph structures. To transform these models with BiXJ, we have to use the techniques in Section 4 to translate graphs into meta trees together with some constraints. For graphs represented in XML with id attributes for node references, we have implemented an automatic tool based on steps presented in Section 4 to do this translation.

Finally, we get the transformation architecture of our case in Figure 10. First, we generate the meta tree (Meta XML) and its constraint (T_0) from the XML file of feature model F-XML$_G$ by our graph-to-tree translation tool. Applying T_0 to Meta XML we get F-XML$_T$ which is isomorphic

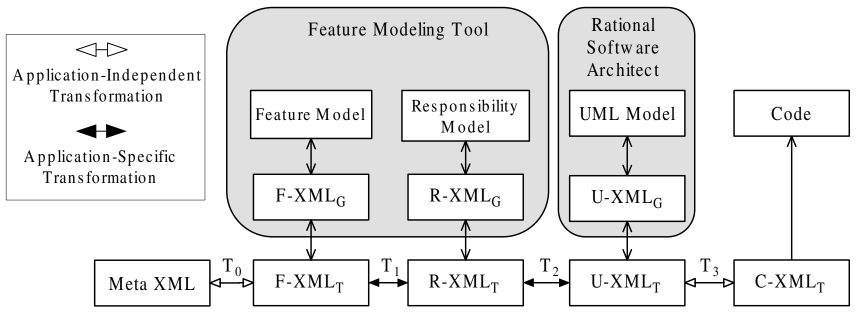

Figure 10: The Architecture of Implementation

with F-XML$_G$. Then, the F-XML$_T$ is transformed by BiXJ transformation T$_1$)into R-XML$_T$ which is isomorphic with the responsibility model(R-XML$_G$). After that, the R-XML$_T$ is transformed by T$_2$ into U-XML$_T$ which is isomorphic with the UML Design Model(U-XML$_G$). The transformations T$_1$ and T$_2$ are application-specific transformations, which is not supported by any CASE tool at present. The last transformation T$_3$, from U-XML$_T$ to XML code representation, is an application-independent transformation, which has been supported by CASE tools, but probably not in a bidirectional way, just like tools in Rational Software Architect.

5.2 The Results

The result is much inspiring. First, our approach is successful to be applied during the whole lifecycle of software development; Second, language BiXJ together with our techniques of representing graph by trees provide powerful means to do bidirectional model transformation in practical applications. Third, the unsatisfied aspects in existing techniques described in Section 2 is not in our approach, illustrated as follows:

Model Evolution With the compositional feature of BiXJ, the evolution of the target model can be described by writing a transformation from the old target model to the new target model, and then composing it with the old transformation using construct `<xseq>`. For example, A is transformed to B by BiXJ code T$_1$ and B evolves to B$'$ by BiXJ code T$_2$. Then we can get a transformation from A to B$'$ by combining T$_1$ and T$_2$ in the format of <`xseq`>[T$_1$ T$_2$].

Adaption of Transformation Strategy BiXJ has the ability to update transformations during backward transforation, which means that the modification to the target model is not only reflected to the source model but also to the transformation strategy. Thus the modification to elements, either newly added or originated from source model, will be kept during backward transformations.

Inner Dependency in Model Inner dependency can be easily realized using "`<xdup>`" construct in BiXJ, which automatically maintains the consistency of duplications during forward and backward transformations.

In addition, applying bidirectional transformation to our case brings lots of benefits to maintenance. First, the traceability between models can be established. For example, if we want to know from which feature the method "Copy" in code is originated, we can rename this method to, say, "__marked__Copy". Then we apply backward transformation on models, and check which feature in the feature model contains the character "__marked__". Second, the consistency between models can be automatically maintained. For example, if we rename the feature "Save" in the feature model to "FileStore" and apply the forward transformation, the Component-Seed "Save", which is derived directly from feature "Save", will also be renamed to "FileStore".

6 Related Work

There has been a lot of work devoting to bidirectional model transformation [AK02, BM02, CH03, KS03, GGadRH03, Gro04]. The most natural approach is to use relation to describe constraints between the source and the target models [GLR$^+$02, Gro04]. Relational specifications (e.g. [AK02], relations in [Gro04], and mapping rules in [KASS03]) can be interpreted bidirectionally. However, relational specification is usual non-executable. To have executable transformations, directions must be fixed, as discussed in [Gro04]. Non-determinism and backtracking of the relational approach make it difficult to be used with the existing systems. In contrast, we adopt the functional approach which is both executable and deterministic, and can be easily used with the existing system, as demonstrated.

Since models are basically graphs, some attempt has been made to add bidirectionality to graph transformation [Sch94]. The point is that a graph can be separated into three corresponding subgraphs. Two of these subgraphs evolve simultaneously while the third keeps track on correspondences between the other graphs. So a graph is evolved by applying graph grammar rules. Other approaches based on graph transformation can be found in [BM02]. These approaches rely on a set of rules and the strategy of rule application, and they assume that there is just one pair of models in order to simplify selection and application of rules. The model produced by application of a rule cannot be used as an input of another rule. This is similar to XSLT, in which the data produced by XSLT cannot be processed again. This means that they are not compositional. In contrast, our approach represents graphs by a tree and a transformation for its constraint, and our approach supports composition of transformations.

Query/View/Transformation (QVT) approach [Gro04] is worth more words, because much effort has been devoted to making it a standard where the feature of bidirectionality is clearly stated. However, there still lack clear semantics for view-updating, and QVT does not support adaption of transformation strategies. Comparatively, we built our system based on an existing bidirectional language, and focus on finding the technique of using them for model transformations.

Bidirectional updating, though an old problem [BS81, DB82, OT94], has recently attracted much interests, each took a slightly different approach according to their target application [MHT04, HMT04, FGM$^+$05]. The language BiXJ is a java version of bidirectional language of X [HMT04]. Different from other bidirectional languages such as [FGM$^+$05], X has an important construct allowing duplication of information while keeping the duplicated information consistency. This plays an important role in formalizing graph-to-graph transformation in this paper.

7 Conclusion

In this paper, we propose a new approach to bidirectional model transformations based on a well-defined bidirectional language BiXJ, where composition plays an important role not only in realizing the framework (i.e., a graph transformation is described by a composition of two tree transformations), but also in using the framework for evolutional and traceable software development. The experimental results on a non-trivial case study have convinced the promise of the new approach.

We highlight one future work. As seen before, it is not straightforward for programmers to program with BiXJ, so it would be more practical to design a high-level language (with graph patterns and rules, e.g. XQuery) that can be compiled to BiXJ. Our ongoing work has shown that XQuery Core can be translated to BiXJ.

Acknowledgements

This work is partially sponsored by the National Natural Science Foundation of China under Grant No. 60528006; the Natural Science Foundation of Beijing under Grant No. 4052018; the National Basic Research Program (973) of China under Grant No. 2002CB312003; and *Comprehensive Development of e-Society Foundation Software* Program of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[Abi99] Serge Abiteboul. On views and XML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Database Systems*, pages 1–9. ACM Press, 1999.

[AK02] D. H. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language 5th International Conference*, number 2460 in LNCS, pages 243–258, Dresden, Germany, oct 2002. Springer.

[BM02] Peter Braun and Frank Marschall. Transforming object models with BOTL. *Electronic Notes on Theoretical Computer Science*, 72(3), 2002.

[BS81] F. Bancilhon and N. Spyratos. Updating semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA 03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[DB82] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM TODS*, 7(3):381–416, 1982.

[FGM+05] 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. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, pages 233–246, 2005.

[Fra03] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Comput ing.* John Wiley & Sons, 2003.

[GGadRH03] Tracy Gardner, Catherine Griffin, and Jana Koehler an d Rainer Hauser. A review of OMG MOF 2.0 Query / Views / Transformations submissions and recommendations towards the final standard. In *MetaModelling for MDA Workshop*, England, 2003. Springer.

[GLR+02] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of mda. In *Proc. Graph*

Transformation - First International Conference, LNCS 2505, pages 90–105, 2002.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

[Gro04] QVT-Merge Group. Revised submission for MOF 2.0 Query/Views/Transformation RFP (ad/2002-04-10), 2004.

[Gro05] Object Management Group. *MOF 2.0/XMI Mapping Specification, v2.1.* http://www.omg.org/cgi-bin/doc?formal/2005-09-01, 2005.

[HMT04] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation*, pages 178–189. ACM Press, 2004.

[HS95] Masami Hagiya and Tomoki Shiratori. Programming by example in computing-as-editing paradigm. In *International Conference on View Languages*, pages 275–283, 1995.

[KASS03] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003.

[KS98] Rudolf K. Keller and Reinhard Schauer. A compositional approach to software design. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 5*, page 386, Washington, DC, USA, 1998. IEEE Computer Society.

[KS03] S. Kent and R. Smith. The bidirectional mapping problem. *Electronic Notes on Theoretical Computer Science*, 82(7), 2003.

[Mei04] Hong Mei. ABC: Supporting software architectures in the whole lifecycle. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 342–143, 2004.

[MHT04] S.C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *Second ASIAN Symposium on Programming Languages and Systems(APLAS 2004)*, pages 2–18, Taipei, Taiwan, 2004. Springer, LNCS 3302.

[OT94] Atsushi Ohori and Keishi Tajima. A polymorphic calculus for views and object sharing. In *ACM PODS'94*, pages 255–266, 1994.

[Sch94] Andy Schurr. Specification of graph translators with triple graph gramm ars. In G. Tinhofer, editor, *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG'94), LNCS 903*, pages 151–163, 1994.

[ZMZ05] Wei Zhang, Hong Mei, and Haiyan Zhao. A feature-oriented approach to modeling requirements dependencies. *Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 273–284, 2005.

[ZMZY05] Wei Zhang, Hong Mei, Haiyan Zhao, and Jie Yang. Transformation from CIM to PIM: A feature-oriented component-based approach. *Lecture Notes in Computer Science*, 3713:248–263, 2005.