A bidirectional-transformation-based framework for software visualization and visual editing

Xiao HE, ChangJun HU, ZhiYi MA and WeiZhong SHAO

Citation: SCIENCE CHINA Information Sciences **57**, 052109 (2014); doi: 10.1007/s11432-013-4919-1

View online: http://engine.scichina.com/doi/10.1007/s11432-013-4919-1

View Table of Contents: http://engine.scichina.com/publisher/scp/journal/SCIS/57/5

Published by the Science China Press

SCIENCE CHINA
Information Sciences

• **RESEARCH PAPER** •

May 2014, Vol. 57 052109:1–052109:23
doi: 10.1007/s11432-013-4919-1

A bidirectional-transformation-based framework for software visualization and visual editing

HE Xiao[1,2], HU ChangJun[1]*, MA ZhiYi[2] & SHAO WeiZhong[2]

[1]*School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing* 100083*, China;*
[2]*Key Laboratory of High Confidence Software Technologies of Ministry of Education* (*Peking University*), *Beijing* 100871*, China*

Received November 24, 2013; accepted February 12, 2014

Abstract Software visualization and visual editing are important and practical techniques to improve the development of complex software systems. A challenge when applying the two technologies is how to realize the correspondence, a bidirectional relationship, between the data and its visual representation correctly. Although many tools and frameworks have been developed to support the construction of visual tools, it is still complicated and error-prone to realize the bidirectional relationship. In this paper, we propose a model-driven and bidirectional-transformation-based framework for data visualization and visual editing. Our approach mainly focuses on 1) how to define and manage graphical symbols in the model form and 2) how to specify and implement the bidirectional relationship based on the technique of bidirectional model transformation. Then, a prototype tool and four case studies are presented to evaluate the feasibility of our work.

Keywords model-driven development, bidirectional transformation, software visualization, visual language, visual editing

Citation He X, Hu C J, Ma Z Y, et al. A bidirectional-transformation-based framework for software visualization and visual editing. Sci China Inf Sci, 2014, 57: 052109(23), doi: 10.1007/s11432-013-4919-1

1 Introduction

Software visualization and visual editing are two important and practical techniques in software engineering. They share some basic ideas, though they are often applied in different contexts. The process of software visualization ($\mathcal{P}_{\mathrm{SV}}$ for short) can be described as a mapping from the data and its visual representation. In this paper, we term the data to be visualized the *domain* and its visual representation the *diagram*. Then, $\mathcal{P}_{\mathrm{SV}}$ can be formalized as $\mathcal{P}_{\mathrm{SV}} : domain \rightarrow diagram$. Similarly, the process of visual editing ($\mathcal{P}_{\mathrm{VE}}$ for short) can be described as a mapping between the data to be edited visually and its graphical symbols denoting the data. We also term the data the *domain* and the graphical symbols the *diagram* to keep the terminology consistency. Hence, $\mathcal{P}_{\mathrm{VE}}$ can be formalized as $\mathcal{P}_{\mathrm{VE}} : domain \leftarrow diagram$.

In this paper, we term the mappings between the *domain* and the *diagram* the *domain-diagram relation* (or the *d-d relation* for short). The *d-d relation* decides the ways of *visualizing* the data with the graphical

*Corresponding author (email: huchangjun@ies.ustb.edu.cn) symbols ($\mathcal{P}_{\rm SV}$) and the ways of *editing* the data when the graphical symbols are changed ($\mathcal{P}_{\rm VE}$). In many cases, the *d-d relations* are bidirectional, each of which can be regarded as a pair of $\mathcal{P}_{\rm SV}$ and $\mathcal{P}_{\rm VE}$. For instance, before we edit a UML[1] model in a graphical modeling tool, it is essential to visualize the model in the tool first. Hence, without loss of generality, a *d-d relation* is defined as follows in this paper: $\textit{d-d relation} : \textit{domain} \leftrightarrow \textit{diagram}$.

It is a difficult task to realize the *d-d relation* during the development of visual tools (software visualization tools and visual editors). There are three main difficulties as follows:

Complex Mapping: The correspondence between the *domain* and the *diagram* may be very complex. For example, a correspondence may be a fragment-to-fragment mapping, but not an element-to-element mapping. How to support such complex mappings is a problem in visual tools.

Consistency: $\mathcal{P}_{\rm SV}$ and $\mathcal{P}_{\rm VE}$ must be consistent, otherwise the *domain* and the *diagram* will be out of synchronization. However, to make the two operations work consistently is complicated, especially when the mappings are complex.

Flexibility of Evolution: The *d-d relation* may be changed in a visual tool. For example, when a visual language evolved, its abstract and concrete syntax, and the syntax mappings may be modified. Then, corresponding visual editor must change the *d-d relation* to adapt itself to the new version of the language. However, how to evolve the *d-d relation* flexibly is usually difficult.

Although many approaches and frameworks, such as GEF[2], GMF[3], GME [1], MetaEdit+ [2], and AToM3 [3], have been proposed to facilitate visual tools, realizing the *d-d relation* is still complex and error-prone due to the limitations of existing approaches [4]. For one thing, current approaches usually employ some predefined mappings to characterize the *d-d relation* and often impose some structural restrictions, which decrease the flexibility and hinder the implementation of complex mappings. For another, in these frameworks, the *d-d relation* is normally implemented by two individual operations, i.e. 'visualize' ($\mathcal{P}_{\rm SV}$) and 'edit' ($\mathcal{P}_{\rm VE}$). This increases the difficulty of keeping them *consistent*.

This paper focuses on how to support complex *d-d relations flexibly* and *consistently* in a visual tool. We propose a bidirectional-model-transformation-based framework for this issue. First, in our framework, the *d-d relation* is defined as a declarative bidirectional model transformation (*BX* for short). In this way, we are able to define complex mappings between the data and the graphical symbols. Furthermore, our approach gives a better separation of the *domain* and the *diagram* because they are loosely connected by a set of transformation rules but not explicit references. This implies that the *d-d relation* can be changed *flexibly* by replacing specific *BX* rules.

Then, based on existing *BX* theory [5–8], the *BX* is interpreted and executed with the bidirectional transformation algorithm. In this way, the *d-d relation* can be implemented by a single *BX consistently* instead of two individual operations. A *BX* can be evaluated in two directions: if it is executed from the *domain* to the *diagram*, it realizes the operation 'visualize'; if it is performed from the diagram to the domain, it realizes the operation 'edit'. Hence, because only one *BX* is required to be developed instead of two operations. And, in our framework, it is easier to keep the two operations 'visualize' and 'edit' *consistent* by executing a single *BX*.

There are two main challenges when realizing our approach. The first is how to define and manage the *diagram* (e.g., the figures and the graphical symbols) in the form of model, since our approach is model-based. The second is how to specify and maintain the *d-d relation* with *BX*, i.e. what is the structure and the execution algorithm of the *BX* rule. The remainder of this paper will solve the two problems. It is structured as follows: Section 2 discusses the limitations of existing approaches and frameworks; Section 3 describes the overview of our framework; Sections 4 and 5 explain our approach in detail; Section 6 presents the tool support; Section 7 introduces four case studies to evaluate our framework; Section 8 addresses the related work; the last section presents the conclusion.

1) Object Management Group. Unified Modeling Language Superstructure Specification Version 2.4.1, formal/2011-08-06, 2011. http://www.omg.org/spec/UML/2.4.1.

2) Moore W, Dean D, Gerber A, et al. Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework. IBM International Technical Support Organization, 2004. http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf.

3) Graphical Modeling Project website: http://www.eclipse.org/modeling/gmp/.

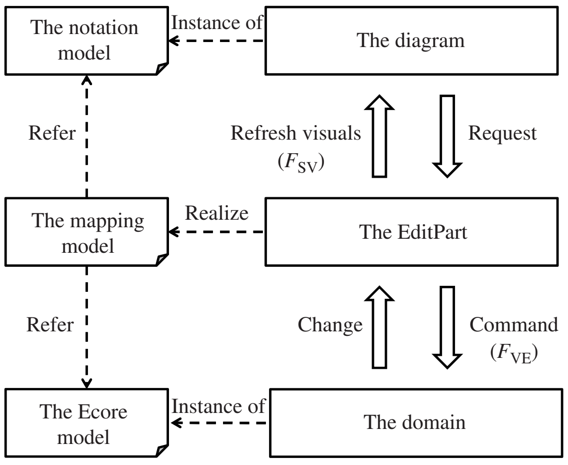


Figure 1 Framework of GMF.

2 Motivation

This section discusses the limitations of implementing *d-d relation* by analyzing the Graphical Modeling Framework (GMF). However, we have to emphasize that the limitations are not specific to GMF. They are shared by various frameworks and technologies of developing visual tools.

In GMF, the structure of the *domain* is specified as an Ecore model, where Ecore is an industrial implementation of the Meta Object Facility (MOF)[4]. The data of GMF is an instance of the Ecore model, which can be managed (created, deleted, and modified) by the Eclipse Modeling Framework (EMF) [9]. The structure of the *diagram* is defined as a notation model using the Graphical Definition Metamodel (GDM) of GMF, which can define the nodes, the links, and all kinds of figures that would be displayed in the *diagram*.

To connect the Ecore model and the notation model, GMF offers a Mapping Metamodel. The Mapping Model provides a set of predefined mappings that can be used to map the *domain* to the *diagram*. In brief, the Mapping Metamodel has four basic kinds of mappings: the canvas mapping, the node mapping, the link mapping, and the label mapping. The canvas mapping maps the root element to the canvas (i.e. the main drawing area) in the *diagram*. The node mapping maps an element in the data to a node symbol, while the link mapping maps an element in the data to a line symbol. At last, the label mapping maps a property of an element to a label, i.e. it specifies how to display a property in the textual form.

When developers have established the Ecore model, the notation model, and the mapping model, GMF will generate a visual editor according to these models: the generated editor can manage the data based on the Ecore model, draw the symbols based on the notation model, and maintain the connection between the data and the symbols based on the mapping model. The structure of GMF can be depicted as Figure 1.

GMF has promoted the development of visual tools. However, there are three limitations that hinder it from constructing complex tools:

1) *The Mapping Metamodel of GMF only supports one-to-one mappings.* As we have explained above, it is not difficult to find that all the predefined mappings are one-to-one mappings. Such kinds of one-to-one mappings are not sufficient in some complex cases. For example, an interface in UML can be denoted by a rectangle symbol, a lollipop symbol, or a socket symbol. However, this case cannot be supported by the one-to-one mappings.

2) *The Mapping Metamodel of GMF imposes some structural constraints.* For example, in GMF, a relationship, which is denoted by a line symbol, must be defined as an element that has two references pointing to the source and the target element respectively. If the relationship does not conform to this

4) Object Management Group. Meta Object Facility (MOF) Specification Version 2.4.1, formal/2011-08-07, 2011. http://www.omg.org/spec/MOF/2.4.1/PDF.

structural constraint, it cannot be mapped to a line symbol by a link mapping.

3) *As a GEF-based environment, GMF realizes the d-d relation separately.* GMF, an MVC-based framework, employs *EditParts* as controllers. An *EditPart* must implement an operation named 'refreshVisuals' to realize $\mathcal{P}_{\mathrm{SV}}$ to update the symbols. Meanwhile, the *EditPart* must respond to the *request* of changing the *diagram* and produce a *command* that realizes $\mathcal{P}_{\mathrm{VE}}$ to modify the data. Although, GMF could generate the default implementation of the *d-d relation*, it also requires a lot of development costs to evolve and maintain the *d-d relation* within *EditParts* consistently under the GMF framework.

The first two limitations reduce the flexibility and the descriptive power of GMF (they were also discussed in [4]). Because the arbitrary mappings between the *domain* and the *diagram* are not supported, complex visual languages (and their visual editors) cannot be specified and implemented within the GMF framework easily. Indeed, there are some ways of avoiding the two limitations. For example, to realize complex mappings, we can introduce a composition mechanism in the data model to group together the fragments of the data. However, this usually requires modifying the definition of the *domain*, which may cause the violation of a certain standard (e.g., the specification of UML). And in some cases, the structure of the *domain* is unable to be changed, when the data as defined and provided by others. Another trick is to use different profiles for different representations of a same data element. Nevertheless, such a profiling mechanism is not very popular in existing approaches and frameworks, especially in generative environments, such as GMF. Furthermore, we may require that the same kind of data elements can be represented by different symbols in a single diagram. This goal cannot be achieved by using the profiling mechanism. At last, the third limitation increases the cost of implementing the *d-d relation* and the difficulty of maintaining the consistency of the implementation.

3 Solution overview

The limitations discussed in Section 2 are not specific to GMF, but also exists in other development frameworks, visualization tools, and visual editors. In this section, we present an overview of our bidirectional-model-transformation-based framework for software visualization and visual editing. The essentials of our approach are 1) to make a fully separation of the *domain* and the *diagram* so that they can be defined and evolved independently, 2) to specify and manage the structures of the *domain* and the *diagram* (i.e. the abstract and the concrete syntax in VL) as (meta-)models, and 3) to realize the *d-d relation* flexibly and consistently with a *BX* between the (meta-)models because the *BX* can support arbitrary mappings and can be executed consistently in two directions.

The overview of our framework can be depicted as Figure 2. First, we have to define the structure of the *domain*, which can be specified as a data metamodel. Because our framework is bidirectional-model-transformation-based, the *domain* and the *diagram* must be structured in the model form. We simply employ Ecore, the most widely-used metametamodel on Eclipse platform, to establish the data metamodel (i.e. an Ecore model), and regard the data is an Ecore-based model (i.e. an instance of the Ecore model), because we do not focus on how to define the structure of the *domain* in this paper. However, if the data is not in the model form, we have to use a third-party library to wrap the data as a model. For example, if the data to be visualized is a Java source file, the library of JaMoPP [10] may be used to convert the source file to an Ecore-based Java model.

The second step in our framework is to establish a symbol definition model. The symbol definition model declares what kinds of graphical symbols there are, which will be used to represent the data visually. And for each graphical symbol, the symbol definition model also specifies its structure that will direct the render engine of the visual tool to draw it on the screen. For example, if we want to realize a Class diagram editor, we have to declare in the symbol definition model that the Class diagram contains the class symbol. Then, we also describe that the class symbol is a rectangle with three compartments. In this way, the visual tool could draw this symbol correctly.

The symbol definition model describes the structure of the graphical symbols. However, as a model-based approach, it must also be able to manage those graphical symbols as a model at runtime. Hence, the third step in our framework is to derive a Runtime Symbol Metamodel (RSM) from the symbol

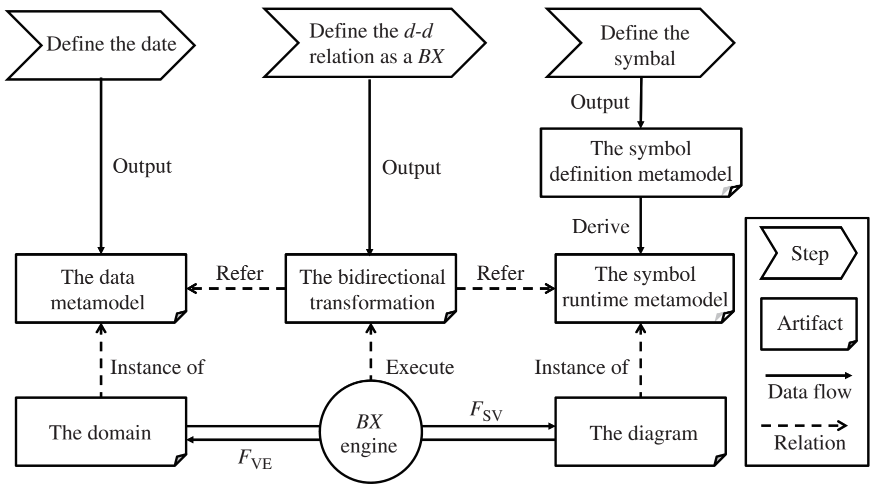


Figure 2 Overview of our solution.

definition model established in the second step automatically. A RSM focuses on the *runtime properties* of symbol instances created on the screen, such as the location and the size of a symbol instance. Those *runtime properties* count for controlling and managing the symbol instances. Note that RSM is different from its corresponding symbol definition model, for they concentrate on different aspects of the same symbols: the symbol definition model defines the structures of the symbols at design time (e.g. a class symbol is a rectangle), while RSM specifies the *runtime properties* that are valid at runtime (e.g. each instance of a class symbol has a location and a size, which are of no use at design time).

The fourth step is to specify the *d-d relation* between the data metamodel (i.e. the Ecore model) and RSM. In our framework, it is defined as a bidirectional model transformation, which consists of a set of *BX* rules specifying the mappings between the *domain* and the *diagram*. To support some complex mappings, we utilize an OCL[5)]-based language to define the *BX* declaratively.

At runtime, the *domain* is wrapped as a data model that is an instance of the data metamodel, while the *diagram* is structured as a runtime symbol model conforming to the runtime symbol metamodel. Then, a *BX* engine is employed in our framework to execute the *BX* so that the data model and the runtime symbol model can be connected and synchronized at runtime. When a user change the data/runtime symbol model, the changes can be propagated to the other side via the *BX* engine.

Let us review the two challenges identified in Section 1. The first is how to define and manage the *diagram* in the model form. In our approach, the *diagram* is defined as a symbol definition model and managed with the help of RSM derived from the definition model. Section 4 discusses how to establish a symbol definition model and how to derive RSM automatically. The second challenge is how to specify and maintain the *d-d relation* with *BX*. Our approach extends the general structure and the algorithm of *BX* to support the *d-d relation*. And Section 5 will explain the structure and the execution algorithm of the *BX* for the *d-d relation*. Note that the paper will not discuss how to define the data metamodel and how to wrap the data as a model because the paper mainly focuses on visualization. As mentioned above, we simply leave this issue to EMF and other third-party libraries.

We illustrate our approach through an example of constructing a *simple Class Diagram* editor. In this example, we use a simplified Class Diagram metamodel as the definition of the *domain*. As shown in Figure 3(a) (actually the metamodel is implemented with Ecore), we can find that there are four non-abstract elements in the metamodel, i.e. *SModel*, *SClass*, *SProperty*, and *SReference* (the character ‘S’ is prefixed to avoid the confusion of the terminology with UML). We try to develop a visual editor that supports the graphical symbols as shown in Figure 3(b). It is not difficult to find that the mapping between the inheritance symbol and ‘general’ association cannot be realized in GMF.

5) Object Management Group. Object Constraint Language (OCL) Specification Version 2.3.1, formal/2012-01-01, 2012. http://www.omg.org/spec/OCL/2.3.1/PDF.

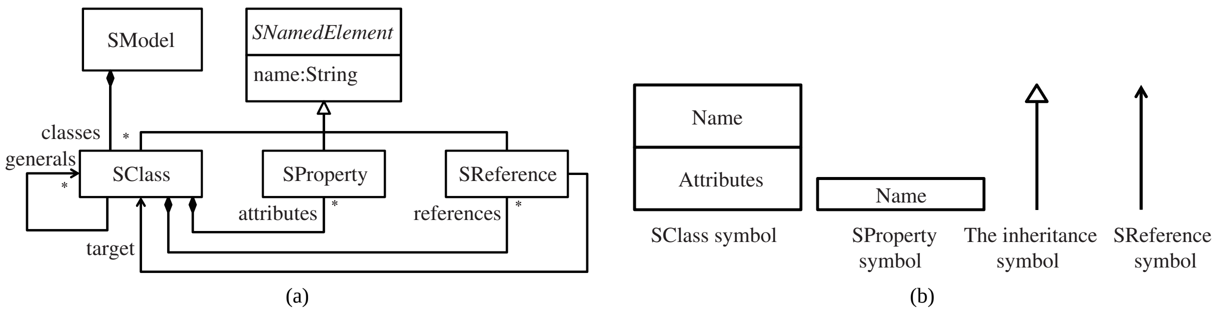


Figure 3 Simple Class Diagram. (a) Metamodel; (b) graphical symbols.

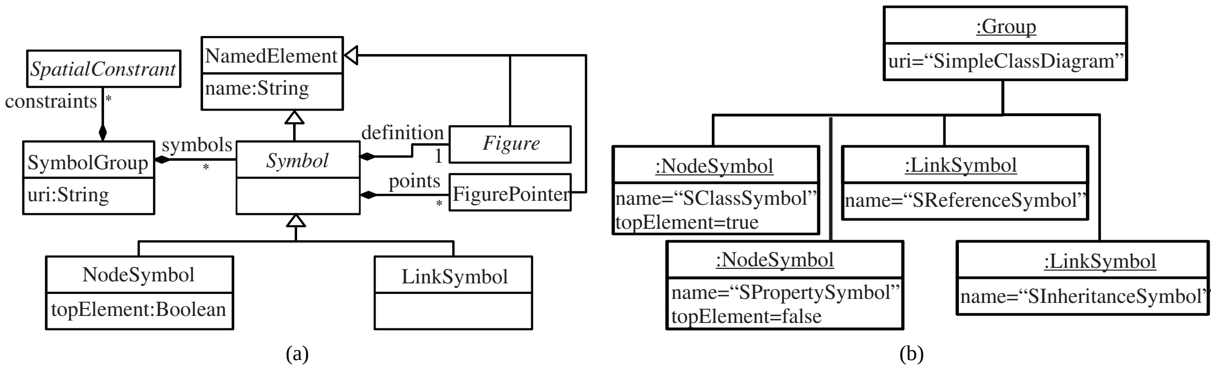


Figure 4 Symbol declaration of Symbol Definition Metamodel. (a) Metamodel; (b) example.

4 Defining graphical symbols

4.1 Establishing symbol definition model

We define the graphical symbols as a symbol definition model. A symbol definition model has three main functions: 1) to declare a set of graphical symbols, which can be used to represent the data visually; 2) to specify the structure of each graphical symbol declared; 3) to define the spatial constraints, which restrict the composition of these graphical symbols.

To establish a symbol definition model, we provide a Symbol Definition Metamodel (SDM), as shown in Figure 4(a). For clarity, we divide SDM into three parts. The first part of SDM, as shown in Figure 4(a), defines the concepts that are used to declare graphical symbols. *SymbolGroup* in Figure 4(a) is a container element where all the symbols and spatial constraints will be included. There are two concrete symbols, i.e. *NodeSymbol* (e.g. the *SClass* symbol in Figure 3(b)) and *LinkSymbol* (e.g. the inheritance symbol and the *SReference* symbol in Figure 3(b)). A *NodeSymbol* can be specified as a top-level symbol by setting the attribute *topElement true*. A top-level symbol can be drawn in the *diagram* directly (e.g. the *SClass* symbol), while a non-top-level symbol can only be drawn within other symbols (e.g. the *SProperty* symbol). An example of declaring the symbols in the *simple Class Diagram* is presented in Figure 4(b). It declares four graphical symbols: *SClassSymbol*, *SPropertySymbol*, *SReferenceSymbol*, and *SInheritanceSymbol*.

Each symbol declared in the symbol definition model must have a *figure definition*, denoted by the aggregation between *Symbol* and *Figure*. *Figure* is an abstract class representing a shape. The child classes of *Figure* are shown in Figure 5(a). However, due to the size limitation, Figure 5(a) only presents a portion of the metamodel. This part takes the implementation framework, i.e. Draw2D [6)], into account. It provides a set of basic figures and some layout policies to compose a graphical symbol. *Figures* can be combined together to form a complex one. The composition of *Figures* is defined as a *FigureReference-Entry* structure. A *FigureReference*, owned by a parent *Figure*, declares a child for the parent. An

6) Draw2d is a rendering framework integrated with GEF. http://www.eclipse.org/gef/draw2d.

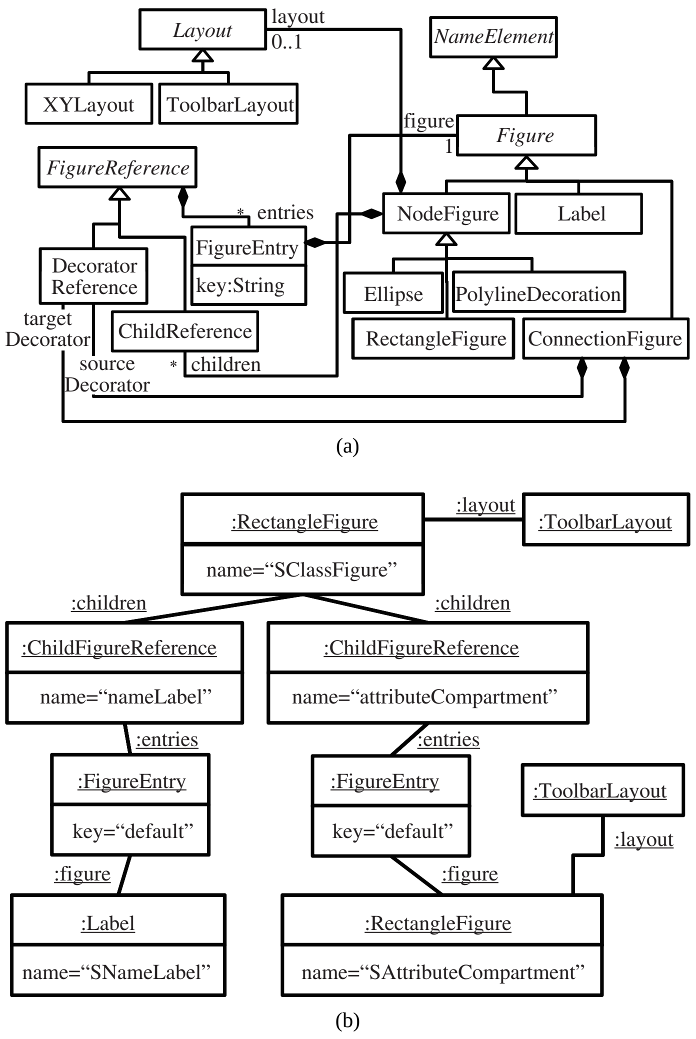


Figure 5 Figure definition of Symbol Definition Metamodel. (a) Metamodel; (b) definition of *SClassSymbol*.

FigureEntry denotes a display alternative for the child. Besides, a *Layout* denotes a strategy of arranging the child *Figures* within the parent. Figure 5(b) shows the figure definition of *SClassSymbol*.

As shown in Figure 4(a), a *Symbol* may also have some *FigurePointers*. A *FigurePointer* can be regarded as an accessor to the child *Figure* composing the *Symbol*. Generally, if we want to control a *Figure* at runtime, we have to establish a *FigurePointer* to this *Figure*.

The last part of SDM can be used to specify the spatial constraints among graphical symbols. The metamodel of this part is presented in Figure 6(a). A spatial constraint restricts the composition of different symbols. There two concrete constraints defined in Figure 6(b). A *NestingConstraint* specifies that a *child NodeSymbol* can be placed into the *container* figure of the *parent NodeSymbol*. For example, as shown in Figure 6(b), a *NestingConstraint* is created to declare a *SPropertySymbol* can be nested within the *attributeCompartment* of a *SClassSymbol*.

4.2 Deriving runtime symbol metamodel

The symbol definition model focuses on the declaration and the structure definition of graphical symbols. When a user wants to create an instance of a symbol on the screen, the model will be used to direct the visual tool to draw the symbol instance. For example, if a *SClassSymbol* is created, the tool knows that a rectangle must be displayed for this instance based on the definition in Figure 5(b). Furthermore, a visual tool also has to record some runtime information for the symbol instance, such as the location and the size. Not contributing to the symbol definition, the runtime information is not defined in the definition

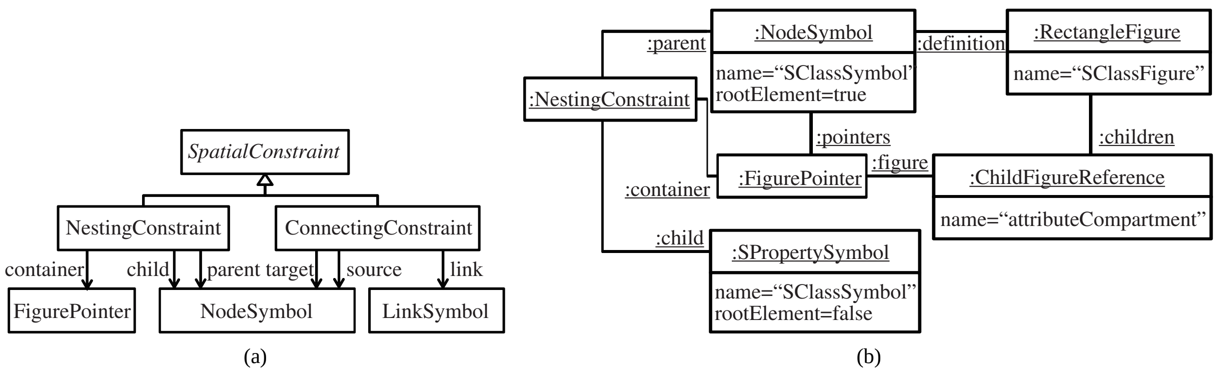


Figure 6 Spatial constraint of Symbol Definition Metamodel. (a) Metamodel; (b) example of spatial constraints.

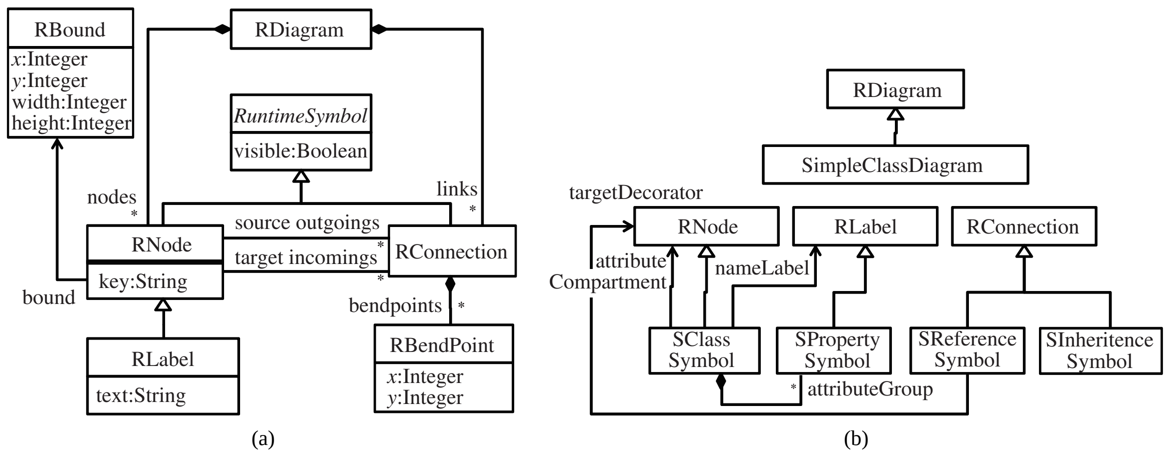


Figure 7 Runtime Symbol Metamodel. (a) Generic RSM; (b) example of definition-specific RSM.

model. To help a visual tool to manage symbol instances, we use a Runtime Symbol Metamodel (RSM) to specify and organize the runtime information, since our approach is model-driven.

RSM focuses on the runtime structures that count for the control and the manipulation of graphical symbols in a visual tool[7]. We divide the definition of RSM into two parts: the generic RSM and the symbol-definition-model-specific RSM (or the definition-specific RSM for short). The generic RSM defines a common structure for symbol *instance*, as shown in Figure 7(a). *RDiagram* in the metamodel is the root element used to organize the model structure. *RNode* specifies the common runtime properties for a node symbol or a node figure within a symbol. A *RNode* is related to a *RBound* that represents the location and the size of the node. *RLabel*, denoting the instance of label, is a subclass of *RNode* with an additional property *text* indicating the string displayed within the label. *RConnection* denotes an instance of a link symbol. It owns some *RBendPoints* that specify the path of the link.

However, a certain symbol set, defined by a symbol definition model, also has some runtime properties specific to the model. Figure 7(b) shows the definition-specific RSM for the *simple class diagram*. It is not difficult to find that the definition-specific RSM imports the generic RSM to complete its structure, e.g. *SClassSymbol** is generalized from *RNode*. Note that the elements *SClassSymbol**, *SPropertySymbol**, *SReferenceSymbol**, and *SInheritanceSymbol** are not moved from Figure 4(b), but are newly created elements that are used to characterize the instances of the graphical symbols with the same names in Figure 4(b)[8]. For example, *SClassSymbol** in Figure 7(b) defines the runtime properties for *SClassSymbol* in Figure 4(b). As mentioned above, a definition-specific RSM also contains some particular runtime properties. For instance, according to Figure 6(b), a *SPropertySymbol* can be placed into a *SClassSymbol*.

7) The basic idea of RSM partially derived from the concept of *Models@Runtime* [11].

8) '*' is a mark, but not a part of the element name. It denote that the element, which is defined in RSM, corresponds to the one with the same in the symbol definition model.

Hence, Figure 7(b) contains a reference between *SClassSymbol** and *SPropertySymbol**, which is used to indicate the *SPropertySymbols** contained by a *SClassSymbol** at runtime.

It is obvious that the definition-specific RSM is closely related to the symbol definition model (we can find some relationships via the element names). Actually, it can be transformed from the symbol definition model automatically. The transformation rules can be briefly stated as follows: 1) a *SymbolGroup* is transformed to a class inherited from *RDiagram*; 2) a *NodeSymbol* is transformed to a class inherited from *RNode* (or an *RLabel* if its figure definition is a *Label*) with the same name, while a *LinkSymbol* is transformed to a class inherited from *RConnection*; 3) a *FigurePointer* owned by a symbol is transformed to a reference owned by the class transformed from the symbol in step 2 to *RNode* (or an *RLabel* if it points to a *Label*), where the name of the reference is equal to the name of the target element of the *FigurePointer*; 4) a *NestingConstraint* is transformed to a reference, owned by the class transformed from the *parent* of the *NestingConstraint*, to the class transformed from the *child* of the *NestingConstraint*, where the name of the reference is equal to the name of the *NestingConstraint*. And it is not difficult to validate that Figure 7(b) is derived from Figure 4(b) and Figure 6(b).

5 Realizing *d-d relation* with bidirectional transformation

After wrapping the graphical symbols in a symbol model, this section discusses how to realize the *d-d relation* with *BX* flexibly and consistently.

5.1 Bidirectional model transformation and *d-d relation*

In general, a bidirectional transformation derives *two* directions of transformations consistently from *one* relation [5]. Formally, a bidirectional model transformation can be defined as follows:

$$\mathcal{M} \xleftrightarrow{R} \mathcal{N}, \tag{1}$$

where $\mathcal{M}$ and $\mathcal{N}$ are two metamodels, and R is a relation between $\mathcal{M}$ and $\mathcal{N}$, i.e. $R \subseteq \mathcal{M} \times \mathcal{N}$.

The forward and the backward transformation derived from the *BX* can be defined as $\overrightarrow{R}$ and $\overleftarrow{R}$ respectively [6, 7]:

$$\overrightarrow{R} : \mathcal{M} \times \mathcal{N} \to \mathcal{N}, \tag{2}$$

$$\overleftarrow{R} : \mathcal{M} \times \mathcal{N} \to \mathcal{M}. \tag{3}$$

$\overrightarrow{R}$ takes a pair of model (m', n) as input and produces a new model n', where $(m', n') \in R$, i.e. it modifies n to n' to enforce the relation R. And $\overleftarrow{R}$ propagates the changes in the opposite direction.

Apparently, when we apply *BX* to constructing visual tools, the relation R of *BX* is the *d-d relation*, and $\mathcal{M}$ and $\mathcal{N}$ become the data metamodel (DM, i.e. the *domain*) and the Runtime Symbol Metamodel (RSM, i.e. the *diagram*) respectively. Hence, the specialized *BX* can be defined as follows:

$$DM \xleftrightarrow{d\text{-}d\ relation} RSM. \tag{4}$$

while its forward and backward transformation are $\overrightarrow{d\text{-}d}$ and $\overleftarrow{d\text{-}d}$. In fact, $\overrightarrow{d\text{-}d}$ and $\overleftarrow{d\text{-}d}$ realize the operations $\mathcal{P}_{\mathrm{SV}}$ and $\mathcal{P}_{\mathrm{VE}}$ respectively. Note that the declaration of $\overrightarrow{d\text{-}d}$ does not conform to $\mathcal{P}_{\mathrm{SV}}$ exactly. $\overrightarrow{d\text{-}d}$ denotes a transformation for realization, while $\mathcal{P}_{\mathrm{SV}}$ is a mapping in concept. $\overrightarrow{d\text{-}d}$ realizes $\mathcal{P}_{\mathrm{SV}}$ but is not identical to $\mathcal{P}_{\mathrm{SV}}$. The case between $\overleftarrow{d\text{-}d}$ and $\mathcal{P}_{\mathrm{VE}}$ is similar to this one.

To ensure the correctness the *BX* between the *domain* and the *diagram*, we define three basic properties to constrain the behavior of the *BX*.

Property 1 (Stability). $\overrightarrow{d\text{-}d}$ and $\overleftarrow{d\text{-}d}$ must be stable, i.e. for any pair of $(d, r) \in$ *d-d relation*, $\overrightarrow{d\text{-}d}\,(d, r) = r$ and $\overleftarrow{d\text{-}d}\,(d, r) = d$.

Property 1 implies that if d and r has satisfied the *d-d relation*, $\overrightarrow{d\text{-}d}$ and $\overleftarrow{d\text{-}d}$ should not modified them.

Property 2 (Consistency)**.** $\overrightarrow{d\text{-}d}$ must be consistent, i.e. for a certain d', if there exists any r' that makes $(d', r') \in$ *d-d relation*, then $\overrightarrow{d\text{-}d}$ (d', r) must return r'', where $(d', r'') \in$ *d-d relation*. Similarly, $\overleftarrow{d\text{-}d}$ must be consistent.

Property 2 implies that $\overrightarrow{d\text{-}d}$ (or $\overleftarrow{d\text{-}d}$) always returns a certain r'' (or d'' for $\overleftarrow{d\text{-}d}$) that makes the *d-d relation* hold if d' (or r' for $\overleftarrow{d\text{-}d}$) can satisfy the *d-d relation* with a certain r' (or d'). Note that we do not require that r'' (or d'') be identical to r' (or d').

Property 3 (Minimum Diagram for $\overrightarrow{d\text{-}d}$)**.** If $r' = \overrightarrow{d\text{-}d}$ (d', r), there should not exist r'' that makes $(d', r'') \in$ *d-d relation* hold, where $r'' \subset r'$.

Property 3 implies that $\overrightarrow{d\text{-}d}$ must return the minimum *diagram*. It means that the output *diagram* should not contain any redundant symbol instances. This property is used to prevent incomplete operations in the *diagram* from any confusion and ambiguity. Because the *diagram* is usually expected to reflect the data truthfully, if the *diagram* contained any symbol instances that did not reflect any information in the *domain*, they should be removed from the *diagram*. In practice, this property is not always valid because some visual editors allow users to create some "pure shapes" that are not connected to the data within the *d-d relation*, such as comments, in the *diagram*. However, this problem can be solved by appending an extra list containing all possible "pure shapes" to the symbol definition model so that these shapes will not be regarded as useless and redundant shapes during the execution of $\overrightarrow{d\text{-}d}$.

5.2 Specifying *d-d relation* as *BX*

As discussed in Subsection 5.1, the *d-d relation* can be regarded as a *BX* between the *domain* and the *diagram*. The specification of a *BX* consists of a set of *BX* rules, each of which describes a part of the *d-d relation*. Although it is possible to define a *BX* rule imperatively, the rule is usually specified declaratively, i.e. we describe the condition on which the rule is satisfied.

In general, a *BX* rule can be defined as a triple:

$$\langle P, L, R\rangle, \tag{5}$$

where P represents the precondition of this rule, L is the left-hand-side of the rule, and R is the right-hand-side. The rule specifies a correspondence between L and R. In this paper, L is the *domain* pattern and R is the *diagram* pattern.

For declarative model transformation, P, L and R are patterns. A pattern can be regarded to be composed by a set of variables $\{v_1, v_2, \ldots, v_n\}$ and a predicate c among these variables, i.e. a pattern is $\langle\{v_1, v_2, \ldots, v_n\}, c\rangle$. An match of a pattern can be considered as a set μ for variable-assignments that makes the predicates c satisfied, i.e. $c(\mu[v_1], \mu[v_2], \ldots, \mu[v_n]) = \mathit{true}$, where $\mu[v]$ returns the value assigned to v in this match. If μ is a match of a pattern p, we write $p(\mu)$; otherwise, $\neg p(\mu)$.

However, in the *d-d relation*, the same data item in the *domain* may be represented by different symbols under different preconditions in the *diagram*. For example, as shown in Figure 8, an *Interface* in UML can be represented as a rectangle or a "lollipop". Furthermore, if an Interface is represented as a rectangle, its *ownedOperations* are also displayed within the second compartment of the rectangle. Such kinds of complex cases that cannot be described in the form of $\langle P, L, R\rangle$ because there may be more than one P and R within one rule. To cover those complex cases, we extend the general structure of a transformation rule as the following one:

$$\langle L, (P_1, R_1), (P_2, R_2), \ldots, (P_n, R_n)\rangle. \tag{6}$$

It can be simply explained as "L should be represented visually by R_i when P_i holds, where $\forall j$ if P_j holds, then $j \geqslant i$; and for each i, if P_i holds, then R_i represents L."

Figure 9(a) shows the structure of the *BX* rule used in the remainder of this paper. It has only one *domain* pattern (i.e. L). It is because one symbol should have only one interpretation in the *domain*; otherwise, there may be some ambiguities in the *diagram*.

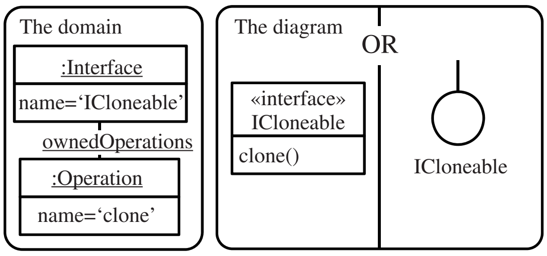


Figure 8 Representations of interface.

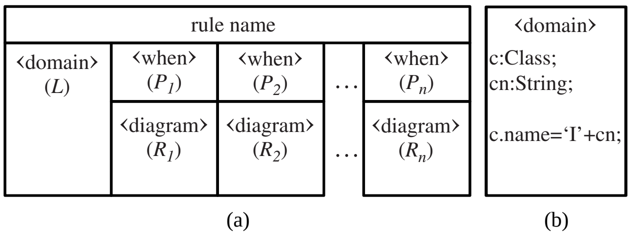


Figure 9 Rule structure. (a) Structure definition; (b) an example of pattern.

Figure 9(b) shows an example of pattern. Each pattern in a rule consists of a sequence of variable declarations and a predicate. A variable is declared in the form of "name : type ;"; and the predicate is a boolean expression involving the variables declared in the pattern. In practice, we employ an OCL-based language, MOL (the Model Operation Language to describe and to evaluate the predicate.

MOL extends OCL in two dimensions: the syntax and the semantics. In the syntax, there are two extensions. First, MOL allows using free variables[9] to simplify the expression and to pass values across patterns. For example, in the following expression, $c1, c2$ are iterators and cn is a free variable:

$$Class :: AllInstances()\text{-}{>}exists(c1 : Class, c2 : Class|c1.name = 'I' + cn\,and\,c2.name = 'C' + cn).$$

If we do not use this free variable, we may have to define a local variable using a *let* expression and use some complex string operations to specify this predicate. Second, MOL supports transformation rule invocation expressions. The following example is a call of the rule "TestRule", where $a := b + 1$ means an explicit parameter assignment from the actual parameter $b + 1$ to the formal parameter a: $TestRule(a := b + 1)$ This syntax enables us to pass value to any variable (or to omit it) within the rule.

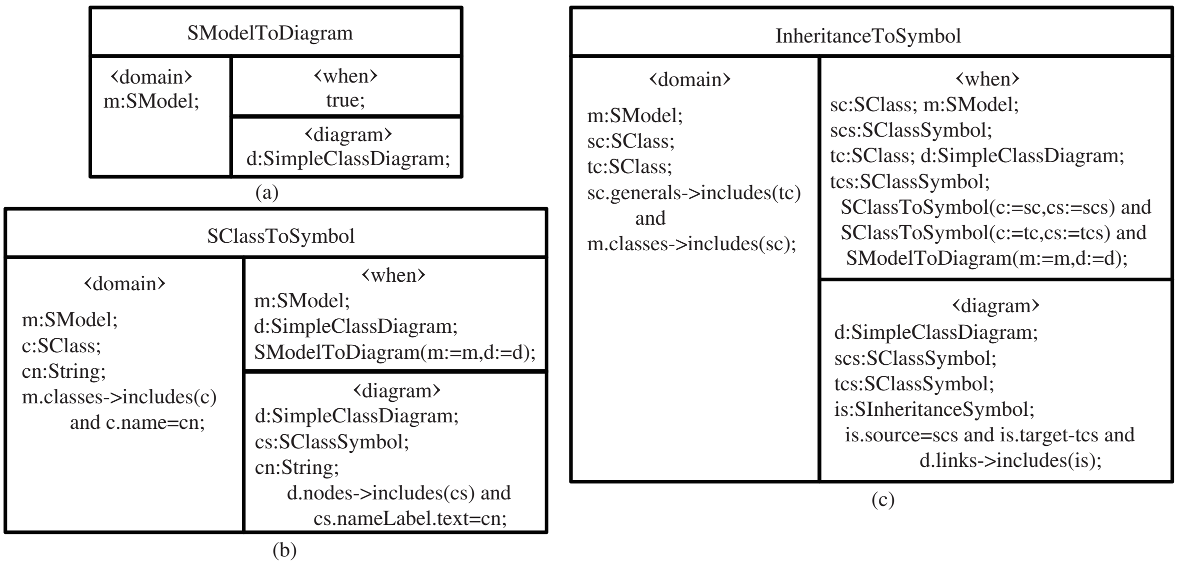
MOL also extends the standard semantics of OCL. It provides three computation modes as follows:

• **Checking.** The *Checking* semantics is almost the same as the standard OCL semantics, which computes the OCL/MOL expression without any modifications to the model. The difference is that the *Checking* semantics will try to estimate the values of the unbound free variables. For example, if the MOL expression $2 = 1 + n$ is evaluated in *Checking* mode (providing that n is an unbound free variable), it will return true and n will be bound to 1 after the evaluation. If there is not any unbound free variable, this semantics is equivalent to the standard OCL semantics.

• **Propagation.** The *Propagation* semantics will try to change the property values of model elements and to estimate the values of the unbound free variables to make the OCL/MOL predicate hold, when the predicate is false under the *Checking* semantics. This implies that the *Propagation* semantics will modify the model, i.e. it has side-effects. However, it could only change the property values, but cannot create new or delete existing model elements and relationships. And the same property can be changed only once during the evaluation. For example, if the predicate $c.name = cn\,and\,cn = 'X'$ is evaluated in the *Propagation* mode (providing that c has been bound to a Class in the model, $c.name$ has not been changed before, and cn has no value), it will be true, while cn will be assigned to $'X'$ and $c.name$ will be changed to $'X'$ too after the evaluation.

• **Enforcement.** The *Enforcement* semantics will change the model and estimate the values of the unbound free variables to enforce the predicate, if the predicate is false under the *Propagation* semantics. The *Enforcement* semantics can modify the model by creating and deleting model elements and relationships and by changing property values. However, all the modifications to the model should not have any conflictions, e.g. deleting a newly created element. For example, if the predicate $c.ownedAttributes\text{-}{>}exists(p : Property|p.name = 'X')$ is evaluated in the *Enforcement* mode (providing that c has been bound to a Class that does not own an attribute named 'X'), it will return true and a new Property named 'X' will be created and appended to *c.ownedAttributes* after the evaluation.

9) In OCL, a variable must be declared as a *context* variable, an iterator, or a formal parameter of an operation, or be initialized as a local variable by a *let* expression.



```
SModelToDiagram
<domain>      | <when>
m:SModel;     | true;
              | <diagram>
              | d:SimpleClassDiagram;
```
(a)

```
SClassToSymbol
<domain>                    | <when>
m:SModel;                   | m:SModel;
c:SClass;                   | d:SimpleClassDiagram;
cn:String;                  | SModelToDiagram(m:=m,d:=d);
m.classes->includes(c)      | <diagram>
     and c.name=cn;         | d:SimpleClassDiagram;
                            | cs:SClassSymbol;
                            | cn:String;
                            |    d.nodes->includes(cs) and
                            |       cs.nameLabel.text=cn;
```
(b)

```
InheritanceToSymbol
<domain>                    | <when>
m:SModel;                   | sc:SClass; m:SModel;
sc:SClass;                  | scs:SClassSymbol;
tc:SClass;                  | tc:SClass; d:SimpleClassDiagram;
sc.generals->includes(tc)   | tcs:SClassSymbol;
     and                    |  SClassToSymbol(c:=sc,cs:=scs) and
m.classes->includes(sc);    |  SClassToSymbol(c:=tc,cs:=tcs) and
                            |  SModelToDiagram(m:=m,d:=d);
                            | <diagram>
                            | d:SimpleClassDiagram;
                            | scs:SClassSymbol;
                            | tcs:SClassSymbol;
                            | is:SInheritanceSymbol;
                            |  is.source=scs and is.target-tcs and
                            |        d.links->includes(is);
```
(c)

Figure 10 Example rules. (a) SModelToDiagram rule; (b) SClassToSymbol rule; (c) InheritanceToSymbol rule.

MOL provides the fundamental support for the execution of a *BX* rule. In this way, a *BX* rule can be interpreted in different modes too. We will explain the execution semantics of a *BX* rule in detail in Subsection 5.3.

With the help of MOL, we are able to specify the *d-d relation* as a set of *BX* rules in the form shown in Figure 9(a). For instance, Figure 10 presents three example rules for the simple class diagram editor. Note that the mapping specified in Figure 10(c) cannot be defined in GMF because a reference (i.e. *sc.generals* in the figure) could not be mapped to a symbol using the Mapping Metamodel of GMF.

5.3 Maintaining *d-d relation* with *BX*

As discussed in Subsection 5.2, we can specify the *d-d relation* as a set of *BX* rules, as shown in Figure 10. And we can use those *BX* rules to maintain the *d-d relation* between the data model and the runtime symbol model. In brief, a *BX* rule, in the abstract form $\langle L, (P_1, R_1), (P_2, R_2), \ldots, (P_n, R_n)\rangle$, is satisfied under the variable-assignment set μ (or μ satisfies the rule) if the following condition holds: if $\exists i \in \{1..n\}$ that lets $P_i(\mu) = true$ and i is the smallest integer, i.e. $\neg P_j(\mu)$, $\forall j \in \{1..i-1\}$, then $L(\mu) \wedge R_i(\mu)$. And μ is called a *trace* of a *BX* rule, if it can satisfy the rule.

We term the combination of a data model and a runtime symbol model a *model space*. By checking the two conditions, we can find out all traces of a *BX* rule within the current model space. This can be realized as follows: 1) we can construct all possible variable-assignment sets $\mathcal{M}$ for the rule under the current model space; 2) for each $\mu \in \mathcal{M}$, if μ makes the rule satisfied, it is a trace of the rule. However, in some cases, we may have an incomplete variable-assignment set μ' in which some variables do not have values (such a kind of the incomplete variable-assignment sets is also named the seed in this paper), and will try to find out all traces generated from μ'. We can simply construct a set $\mathcal{M}' \equiv \{\mu | \mu' \subseteq \mu\}$, and select all the elements that satisfy the rule from $\mathcal{M}'$. It is also possible to use a *BX* rule to create the counterpart for the matched pattern. Assume that μ' is a seed. If $P_i(\mu') \wedge L(\mu') \wedge \neg R_i(\mu')$, then we can construct a new variable-assignment set μ that makes $R_i(\mu)$ by changing the model space, where $\mu' \subset \mu$. Similarly, we can create the counterpart when $P_i(\mu') \wedge R_i(\mu') \wedge \neg L(\mu')$. Besides, a trace, the instance of a *BX* rule, records a correspondence between the fragments of the data and the runtime symbol model. It means that we can propagate changes between the two models with the help of traces by resetting the property values of the model elements that occur in the trace.

As discussed above, a *BX* rule that specifies a part of the *d-d relation* can be used to achieve different tasks by executing it in different ways. Similar to MOL, the *BX* rule in this paper has three execute modes: *Checking*, *Propagation* and *Enforcement*. The three modes can be described briefly as follows:

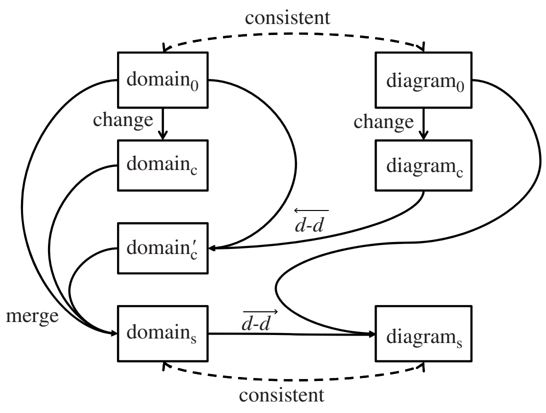


Figure 11 Framework of model synchronization.

• **Checking.** In the *Checking* mode, we try to check and find out all the rule instances within the current model space for a seed μ'(if there is no seed, μ' is regarded as $\emptyset$). First, for the seed μ' , construct the set $\mathcal{M} \equiv \{\mu|\mu' \subseteq \mu\}$. Then, evaluate the predicates contained in the rule using the *Checking* semantics of MOL. If the predicate in a pattern returns true for the variable-assignment set μ in the *Checking* mode of MOL, μ is a match of the pattern. At last, return all the traces of the rule, i.e. return $\{\mu|P_i(\mu) \wedge L(\mu) \wedge R_i(\mu),$ if i is the smallest integer that lets $P_i(\mu) = true\}$.

• **Propagation.** In the *Propagation* mode, we try to propagate the changes from one side to the other side. For a trace μ, assume that $P_i(\mu') \wedge R_i(\mu') \wedge L(\mu')$. If the data model (or the runtime symbol model) is changed, the changes can be propagated to the other side by evaluating the predicates in the L and R_i using the *Propagation* semantics of MOL, when P is still satisfied. Otherwise, delete all the model elements that only exist in the target side, when the propagation fails.

• **Enforcement.** In the *Enforcement* mode, we try to enforce a *BX* rule by creating new model elements in the model space. Let μ' be an incomplete variable-assignment set. Assume that $P_i(\mu') \wedge L(\mu')$ is true and the variables, which only exist in R_i, do not have values in μ'. If there is no variable-assignment set μ that is a super set of μ' and can satisfy the *BX* rule, change the runtime symbol model via creating new elements according to the pattern R_i and evaluating the predicate of R_i in the *Enforcement* mode of MOL. Similarly, the changes can also be propagated from the runtime symbol model to the data model.

After introducing the three execution modes, we can define the procedures of $\overrightarrow{d\text{-}d}$ and $\overleftarrow{d\text{-}d}$. Briefly, both $\overrightarrow{d\text{-}d}$ and $\overleftarrow{d\text{-}d}$ have the following four steps: 1) to propagate changes from one side to the other; 2) to check the rule to find new correspondences; 3) to enforce the rule to create the counterparts for the elements that do not correspond in the other side; 4) some postprocessing, e.g. delete unused elements in the runtime symbol model. The algorithms of $\overrightarrow{d\text{-}d}$ and $\overleftarrow{d\text{-}d}$ are listed as some pseudo codes in Algorithm 1 and Algorithm 2 respectively. The two procedures are very similar to each other, except for the execution directions and the postprocessing. Algorithm 1 propagates and enforces the rule from the *domain* to the *diagram*, while Algorithm 2 executes in the opposite direction. When unused elements in the diagram are found, Algorithm 1 deletes them from the model, while Algorithm 2 simply reports some warnings and rejects the output if required.

The two procedures described in Algorithm 1 and Algorithm 2 are implementations of $\mathcal{P}_{\mathrm{SV}}$ and $\mathcal{P}_{\mathrm{VE}}$. They can be used independently to realize visualization and visual editing separately if only one side (the data model or the runtime symbol model) is changed. However, if both sides are changed concurrently, we must combine the two algorithms together to achieve model synchronization. We will employ the model synchronization technique proposed in [8], which is based on the techniques of bidirectional transformation and model merging [12], to handle concurrent modifications. The model synchronization framework adapted from [8] is displayed in Figure 11. Assume that *domain*$_0$ and *diagram*$_0$ are consistent at the beginning. Then, they are changed concurrently to *domain*$_c$ and *diagram*$_c$. After that, we execute $\overleftarrow{d\text{-}d}(domain_0, diagram_c)$ according to Algorithm 2 to get *domain*$'_c$ reflecting the changes from *domain*$_c$ (note that we do not reject *domain*$'_c$ if there still are some unused symbols in *diagram*$_c$ after this step

Algorithm 1 The procedure of the forward transformation from the *domain* to the *diagram*

```
 1: for all rule $r$ in $d$-$d$ relation do
 2:     propagate the changes from the domain to the diagram for all traces of $r$
 3:     if the propagation failed for a trace $\mu$, delete the model elements bound to the variables that only occur in the
        diagram pattern
 4: end for
 5: for all rule $r = \langle L, (P_1, R_1), (P_2, R_2), \dots, (P_n, R_n)\rangle$ in $d$-$d$ relation do
 6:     check the rule $r$ and find all the correspondences
 7:     for each $\mu$ that let $L(\mu) = true$, if $P_i(\mu) \wedge \neg R_i(\mu)$, enforce the rule $r$ for $\mu$ so as to make $R_i(\mu) = true$, where $i$ is
        the smallest integer that makes $P_i(\mu) = true$
 8: end for
 9: for all element $e$ in the runtime symbol model do
10:     if $e$ is not referred by a trace found or created above, delete it from the model
11: end for
```

because they will be removed during $\overrightarrow{d\text{-}d}$). We merge the models $domain_0$, $domain_c$, and $domain'_c$ to combine the changes from $domain_c$ and $domain'_c$ to $domain_0$ (conflicted changes are also detected and resolved during merging). The result is $domain_s$ that contains all the changes from $domain_c$ and $domain'_c$. At last, we execute $\overrightarrow{d\text{-}d}(domain_s, diagram_s)$ to generate $diagram_s$ that is consistent with $domain_s$. The correctness of this synchronization approach has been discussed and validated in [8].

Algorithm 2 The procedure of the backward transformation from the *diagram* to the *domain*

```
 1: for all rule $r$ in $d$-$d$ relation do
 2:     propagate the changes from the diagram to the domain for all traces of $r$
 3:     if the propagation failed for a trace $\mu$, delete the model elements bound to the variables that only occur in the
        domain pattern
 4: end for
 5: for all rule $r = \langle L, (P_1, R_1), (P_2, R_2), \dots, (P_n, R_n)\rangle$ in $d$-$d$ relation do
 6:     check the rule $r$ and find all the correspondences
 7:     for each $\mu$, if $P_i(\mu) \wedge R_i(\mu) \wedge \neg L(\mu)$, enforce the rule $r$ for $\mu$ so as to make $L(\mu) = true$
 8: end for
 9: for all element $e$ in the runtime symbol model do
10:     if $e$ is not referred by a trace found or created above, reject the output model and report warnings
11: end for
```

We discuss the properties of the bidirectional transformation algorithms proposed above. We defined three properties in Subsection 5.1: *stability*, *consistency*, and *minimum diagram*.

First, Algorithm 1 and Algorithm 2 satisfy *stability*. If the data model and the runtime symbol model are consistent, the propagation step, the checking step, and the enforcing step will not modify the target model, and there will not be any unused symbols in the *diagram*. Hence, the two algorithms will return the original target model unchanged in such a case.

Second, the two algorithms satisfy *consistency*. It is because they will propagate the changes and enforce *BX* rules to the target model to make the two model consistent. If there are some useless symbols after the enforcement, $\overrightarrow{d\text{-}d}$ will delete them, while $\overleftarrow{d\text{-}d}$ will report warnings and reject the output. So if $\overrightarrow{d\text{-}d}$ or $\overleftarrow{d\text{-}d}$ can return a valid output, it must be consistent with the model on the opposite side.

Third, Algorithm 1 satisfies *minimum diagram*, because it is apparent that the algorithm deletes all unused symbols in the diagram in line 10 of Algorithm 1. And to be consistent with Algorithm 1, Algorithm 2 will reject the output if unused symbols are detected.

6 Tool support

In Section 4, we discussed how to define the graphical symbols as a symbol definition model and how to derive the runtime symbol metamodel automatically. In Section 5, we discussed how to specify the *d-d relation* as a set of *BX* rules and how to maintain the *d-d relation* with the technique of bidirectional model transformation in a visual tool. Based on our approach to software visualization and visual editing, we implemented a tool support named PKU Graphical Tool Platform (PKUGTP), an Eclipse-based tool,

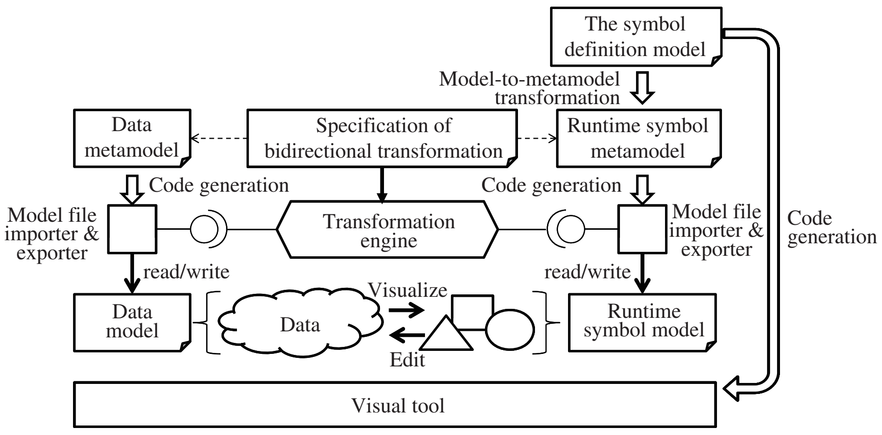


Figure 12 Tool support.

to facilitate the development of visual tools. The architecture of our tool is depicted in Figure 12. As shown in the figure, users of our tool have to provide the data metamodel, the symbol definition model , and the specification of the *BX* transformation. Then, our tool can derive the runtime symbol metamodel from the symbol definition model via a model-to-metamodel transformation. After that, our tool will invoke three code generation procedures to produce the codes of the visual tool and the model file importer/exportor for the data metamodel and the runtime symbol metamodel. In the generated visual tool, the data and the diagram are represented as a date model and a runtime symbol respectively. Finally, the transformation engine can interpret the specification of the *BX* rules and maintain the *d-d relation* in the visual tool by executing $\overrightarrow{d\text{-}d}$ and $\overleftarrow{d\text{-}d}$ according to the algorithms presented in Subsection 5.3.

7 Case study

We carried out some case studies to evaluate the feasibility of our approach and our tool support PKUGTP. We will introduce two cases, i.e. a Petri Net visual editor and a Java method Call Graph Tool, in detail in this section. And we compare our approach with other approaches qualitatively through those case studies.

7.1 Petri Net editor

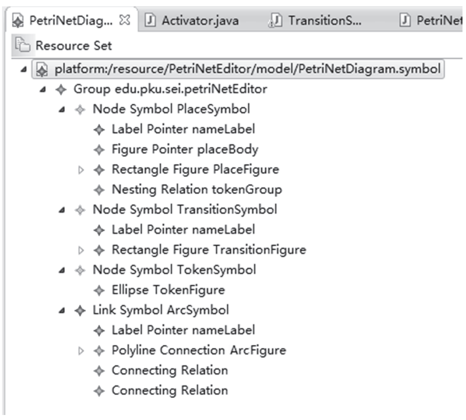
We will show how to use our approach to construct a visual modeling editor and how to use *BX* to specify and maintain the *d-d relation* in a modeling tool.

First, we defined the metamodel of Petri Net. It contains the following metaclasses: *PetriNet* (i.e. the root element of a Petri Net model), *Place*, *Token*, *Transition*, and *Arc*. Then, we established the symbol definition model for the Petri Net diagram as shown in Figure 13(a), which contains three node symbols (i.e. *PlaceSymbol*, *TransitionSymbol*, and *TokenSymbol*) and a link symbol (i.e. *ArcSymbol*).

After that, some code generation procedures were invoked to produce the visual editor automatically. The standard EMF code generator was executed first to generate the implementation of the Petri Net metamodel and the runtime symbol metamodel of Petri Net editor. The model importer and exporter were also generated by EMF at the same time. Then, PKUGTP began to generate the implementation of the figures, the edit parts, and the visual editor from the symbol definition model. At last, PKUGTP finished the tool generation and combined all the generated parts together.

To connect the *domain* and the *diagram*, we specified the *d-d relation* as six *BX* rules, i.e. *PetriNetToDiagram*, *PlaceToSymbol*, *TransitionToSymbol*, *PTArcToSymbol*, *TPArcToSymbol*, and *TokenToSymbol*. The specifications of *PlaceToSymbol* and *PTArcToSymbol* are depicted in Figure 14.

This case study shows that our framework can successfully be applied to visual modeling tools to maintain the bidirectional mappings between the data model and its visual representation.

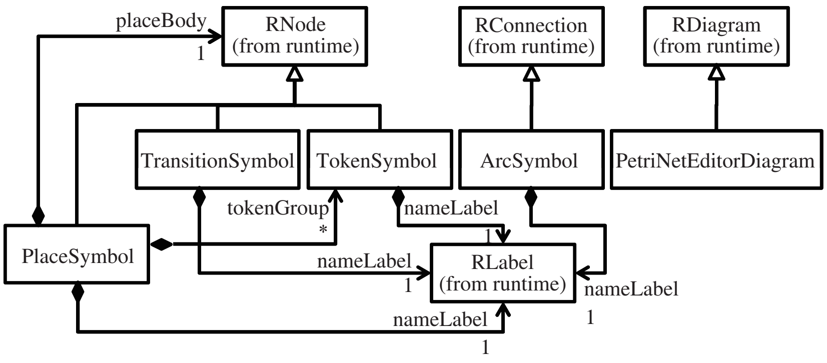


PetriNetDiag... ⊠ | Activator.java | TransitionS... | PetriNet

Resource Set

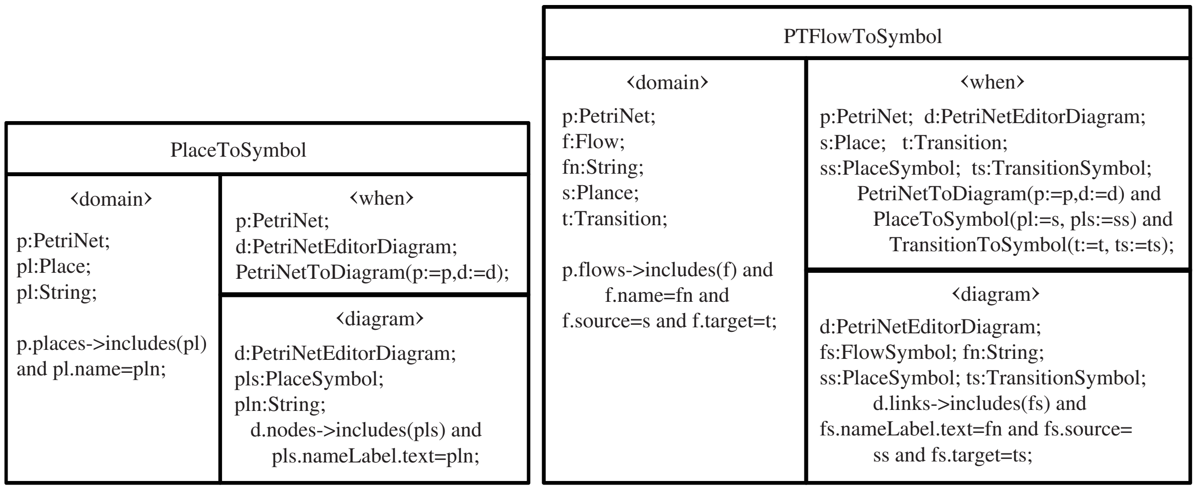
- platform:/resource/PetriNetEditor/model/PetriNetDiagram.symbol
  - Group edu.pku.sei.petriNetEditor
    - Node Symbol PlaceSymbol
      - Label Pointer nameLabel
      - Figure Pointer placeBody
      - Rectangle Figure PlaceFigure
      - Nesting Relation tokenGroup
    - Node Symbol TransitionSymbol
      - Label Pointer nameLabel
      - Rectangle Figure TransitionFigure
    - Node Symbol TokenSymbol
      - Ellipse TokenFigure
    - Link Symbol ArcSymbol
      - Label Pointer nameLabel
      - Polyline Connection ArcFigure
      - Connecting Relation
      - Connecting Relation

(a)



(b)

Figure 13 Data and symbol definition of Petri Net editor. (a) Symbol definition model; (b) runtime Symbol metamodel.



PlaceToSymbol

‹domain›
p:PetriNet;
pl:Place;
pl:String;

p.places->includes(pl)
and pl.name=pln;

‹when›
p:PetriNet;
d:PetriNetEditorDiagram;
PetriNetToDiagram(p:=p,d:=d);

‹diagram›
d:PetriNetEditorDiagram;
pls:PlaceSymbol;
pln:String;
d.nodes->includes(pls) and
pls.nameLabel.text=pln;

PTFlowToSymbol

‹domain›
p:PetriNet;
f:Flow;
fn:String;
s:Plance;
t:Transition;

p.flows->includes(f) and
f.name=fn and
f.source=s and f.target=t;

‹when›
p:PetriNet; d:PetriNetEditorDiagram;
s:Place; t:Transition;
ss:PlaceSymbol; ts:TransitionSymbol;
PetriNetToDiagram(p:=p,d:=d) and
PlaceToSymbol(pl:=s, pls:=ss) and
TransitionToSymbol(t:=t, ts:=ts);

‹diagram›
d:PetriNetEditorDiagram;
fs:FlowSymbol; fn:String;
ss:PlaceSymbol; ts:TransitionSymbol;
d.links->includes(fs) and
fs.nameLabel.text=fn and fs.source=
ss and fs.target=ts;

Figure 14 A part of *d-d relation* of Petri Net editor.

Besides, this case study can also demonstrate that it is simpler to realize and maintain the *d-d relation* in our approach than in the GEF-based tools. Let us take the mapping between *Place* and *PlaceSymbol* as an example. If we want to realize this mapping under GEF, we must implement a creation command, a deletion command, and a refreshing method, at least. However, in our approach, we do not have to realize those operations. We just specify the mapping as a *BX* rule, as shown in Figure 14. The rule prescribes that a *Place* in a *PetriNet* (i.e. the *domain*) is and must be represented by a *PlaceSymbol* in the *PetriNetEditorDiagram* (i.e. the *diagram*), and the name of the *Place* is also displayed in the *nameLabel* of the *PlaceSymbol*. Then, the rule can be interpreted according to the algorithms provided in Subsection 5.3 to maintain the correspondence between the *Places* and the *PlaceSymbols* automatically. Although we cannot strictly prove that specifying a *BX* rule is easier than implementing a set of operations, based on our observation, a *BX* rule usually has fewer lines of codes (LOC). Furthermore, with the help of the existing *BX* theory, we do not have to worry about the consistency problem existing in GEF-based tools. At least, if a rule is incorrect and cannot be executed bidirectionally, the *BX* engine can detect and report the problem, based on the properties defined in Subsection 5.1.

GMF-based tools are spared from the problems of development cost and the consistency because GMF generates the default implementation of the operations. However, if default implementation does not satisfy the requirement and needs customizing, we must modify the generated codes. This requires more efforts because we have to comprehend the generated codes. And we would confront the consistency problem again. Besides, GMF suffers from other problems, which will be discussed below.

7.2 Java method Call Graph Tool

In this section, we will employ our framework to construct a Java method call graph visualization tool. This tool can represent all method calls within a set of focused Java classes as a call graph visually. The focused classes are termed the *scope* in this subsection. Basically, in the tool, a method is denoted by a circle, while a method call is represented as a directed link from the caller to the callee.

Obviously, the domain of this tool is the Java code. Hence, to apply our approach, we have to transform Java codes as a Java model first. We utilized JaMoPP [10] to accomplish this task. JaMoPP can parse a .java file and return a Java model that reflects all information of the .java file. It also enables us to manipulate Java codes via modifying the Java model. We simply regard the Java model returned by JaMoPP as the data model.

We defined six symbols totally, i.e. *ClassSymbol*, *MethodSymbol*, *ExternalMethodSymbol*, *RegCallSymbol*, *RecCallSymbol*, and *ExtCallSymbol*. The link symbol *RegCallSymbol* represents a call from the caller method to the callee method within the scope; *RecCallSymbol* denotes a recursive call; and *ExtCallSymbol* represents a call from a method in the scope to an outside one. After that, we defined the *d-d relation* by seven transformation rules. The three core rules are *RegCallToSymbol*, *RecCallToSymbol*, and *ExternalCallToSymbol*, as shown in Figure 15.

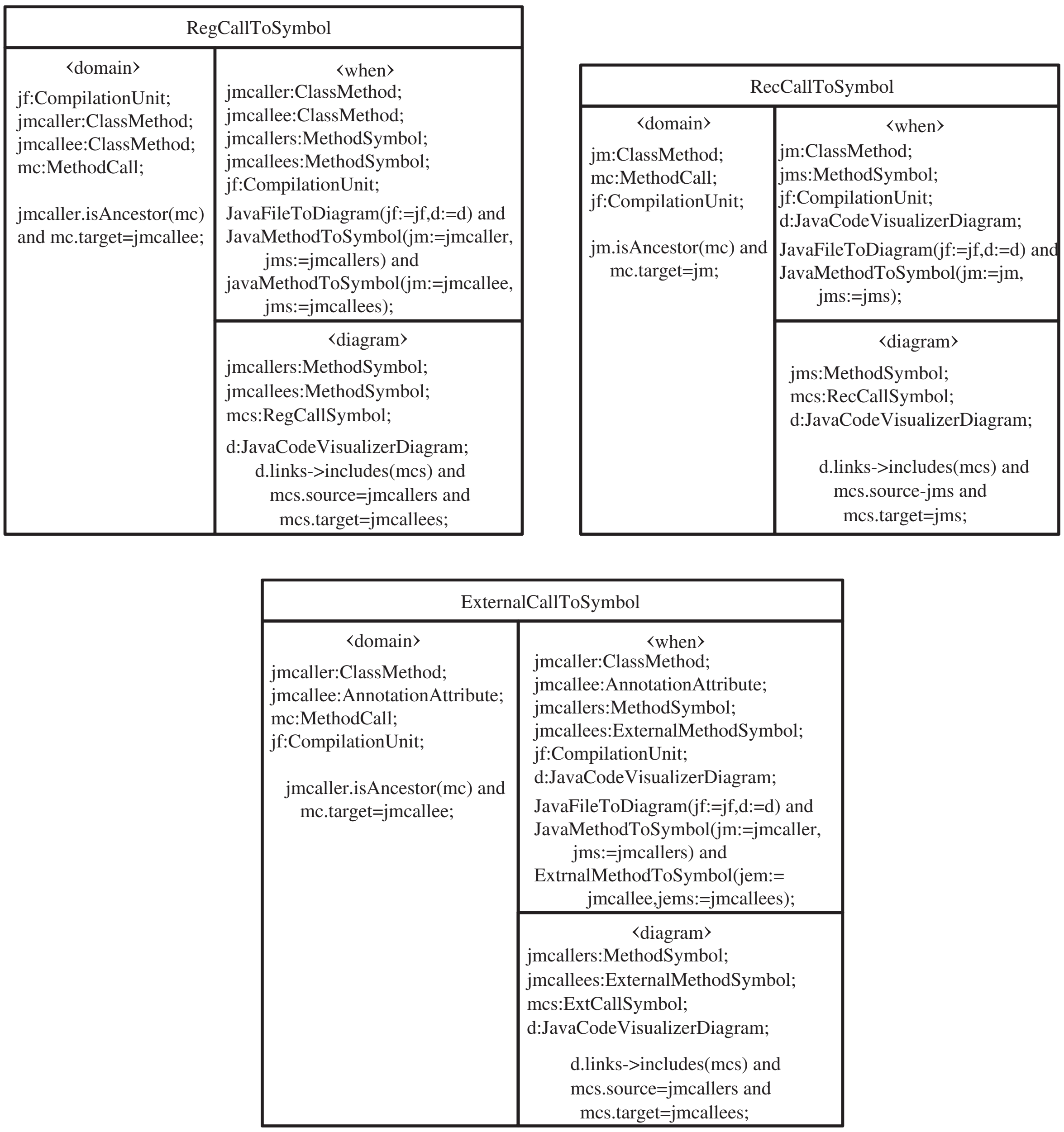
A method call may exist in almost any place within a method body. To find all method calls, we defined a new function *isAncestor* as follows: "self.isAncestor(arg)" returns true, if and only if "self" is equal to "arg" or "self" is the ancestor element of "arg" (i.e. there is a path of containment references from "self" to "arg"). For example, the predicate in the domain pattern of *RegCallSymbol* is "jmcaller.isAncestor(mc) and mc.target=jmcallee". It will return true if the method "jmcaller" contains a statement"mc" and the target of "mc" is the method "jmcallee".

This case study shows how to use our framework to construct a software visualization tool. We only use the forward transformation, i.e. $\overrightarrow{d\text{-}d}$, to realize the operation $\mathcal{P}_{\mathrm{SV}}$. If we apply the $\overleftarrow{d\text{-}d}$, the tool will also support visual editing. However, in this case study, the three rules presented in Figure 15 cannot be executed inversely. It is because the function "isAncestor", used to define the predicates, does not have an inverse function. Hence, the *BX* engine does not know how to enforce "self.isAncestor(arg)". This means that not all the modifications to the *diagram* can be propagated to the *domain* via the *BX* in this tool.

This case study demonstrates that our approach is able to handle complex mappings. As shown in Figure 15, we map *MethodCall* to *RegCallSymbol*, *RecCallSymbol*, and *ExtCallSymbol* under different conditions. This is a kind of one-to-many mappings. And, we just specify three rules to realize this one-to-many mapping.

7.3 Other case studies

Besides the two case studies presented above, we also carried out another three case studies as follows: a simple class diagram editor, a simple class diagram editor for Ecore, and a Java project dependency



```
RegCallToSymbol
‹domain›
jf:CompilationUnit;
jmcaller:ClassMethod;
jmcallee:ClassMethod;
mc:MethodCall;

jmcaller.isAncestor(mc)
and mc.target=jmcallee;

‹when›
jmcaller:ClassMethod;
jmcallee:ClassMethod;
jmcallers:MethodSymbol;
jmcallees:MethodSymbol;
jf:CompilationUnit;
JavaFileToDiagram(jf:=jf,d:=d) and
JavaMethodToSymbol(jm:=jmcaller,
        jms:=jmcallers) and
javaMethodToSymbol(jm:=jmcallee,
        jms:=jmcallees);

‹diagram›
jmcallers:MethodSymbol;
jmcallees:MethodSymbol;
mcs:RegCallSymbol;
d:JavaCodeVisualizerDiagram;
    d.links->includes(mcs) and
       mcs.source=jmcallers and
         mcs.target=jmcallees;
```

```
RecCallToSymbol
‹domain›
jm:ClassMethod;
mc:MethodCall;
jf:CompilationUnit;

jm.isAncestor(mc) and
    mc.target=jm;

‹when›
jm:ClassMethod;
jms:MethodSymbol;
jf:CompilationUnit;
d:JavaCodeVisualizerDiagram;
JavaFileToDiagram(jf:=jf,d:=d) and
JavaMethodToSymbol(jm:=jm,
        jms:=jms);

‹diagram›
jms:MethodSymbol;
mcs:RecCallSymbol;
d:JavaCodeVisualizerDiagram;

    d.links->includes(mcs) and
       mcs.source-jms and
         mcs.target=jms;
```

```
ExternalCallToSymbol
‹domain›
jmcaller:ClassMethod;
jmcallee:AnnotationAttribute;
mc:MethodCall;
jf:CompilationUnit;

   jmcaller.isAncestor(mc) and
      mc.target=jmcallee;

‹when›
jmcaller:ClassMethod;
jmcallee:AnnotationAttribute;
jmcallers:MethodSymbol;
jmcallees:ExternalMethodSymbol;
jf:CompilationUnit;
d:JavaCodeVisualizerDiagram;
JavaFileToDiagram(jf:=jf,d:=d) and
JavaMethodToSymbol(jm:=jmcaller,
        jms:=jmcallers) and
ExtrnalMethodToSymbol(jem:=
          jmcallee,jems:=jmcallees);

‹diagram›
jmcallers:MethodSymbol;
jmcallees:ExternalMethodSymbol;
mcs:ExtCallSymbol;
d:JavaCodeVisualizerDiagram;
    d.links->includes(mcs) and
    mcs.source=jmcallers and
      mcs.target=jmcallees;
```

Figure 15   Definition of *d-d relation* for Java method Call Graph Tool.

viewer. The simple class diagram editor has been used as a demonstration to explain our framework in Section 4 and Section 5. Hence, we will briefly introduce the other two case studies in this subsection.

Simple Class Diagram for Ecore. In the example of the simple class diagram editor, we defined some graphical symbols to represent simple class models visually. The metamodel of the simple class model is defined in Figure 3(a). However, we wanted to reuse the graphical symbols defined in this tool to support visual modeling with Ecore, the metametamodel that is used to establish all metamodels in our tool support. To achieve this goal, we specified a new set of *BX* rules that maps a portion of Ecore metametamodel to the simple class diagram, e.g. *EClass* is mapped to *SClassSymbol* and *EAttribute* is mapped to *SPropertySymbol*. We simply replaced the original *BX* rules with the new *BX* rules. Then, the editor became simple class diagram editor for Ecore without any extra coding.

Java Project Dependency Viewer. We implemented a Java project dependency viewer based on our framework. The goal of this viewer is to represent the dependency relationships among a set of Java projects (on Eclipse platform) as a dependency graph visually. First, we defined a metamodel of project dependency. Then, we implemented a data extractor that can read the project dependency information and output a project dependency model. After that, we defined the graphical symbols of the dependency graph and generated the viewer using PKUGTP. We also specified the *d-d relation* between the dependency model and the dependency graph. Then, the generated tool can visually represent dependencies among Java projects.

7.4 Discussion

In this subsection, we will discuss the flexibility, the consistency, and the execution efficiency of our framework as follows:

Flexibility and consistency. Our framework uses the technique of bidirectional transformation to realize the connection between the *domain* and the *diagram*, and the *d-d relation* is specified as a set of *BX* rules. The *BX* enhances the flexibility of our framework by providing a clear separation of the *domain* and the *diagram*. Our framework enables developers to evolve the *domain* and the *diagram* separately. For instance, as discussed in the simple class diagram for Ecore in Subsection 7.3, we changed the *domain* of the tool from the simple class model to the Ecore model. We only defined a new set of *BX* rules, then the editor became a visual metamodeling tool from a class diagram editor. However, if we want to change the *domain* (or the *diagram*) of a GMF/GEF-based editor, we have to modify the source codes of both the controller-part and the *diagram* (or the *domain*) due to their high coupling. Besides, as discussed in Subsection 5.3, the *d-d relation* can be realized by interpreting a single *BX* consistently because of the *consistency* of the *BX* algorithm. However, if developers implement the *d-d relation* in a GEF/GMF-based editor, they have to carefully avoid any conflicts between the operation "refreshVisuals" and request handlers (as well as the GEF commands produced by those handlers) of the edit part.

Execution efficiency. When performing the case studies, we noticed that the execution of the *BX* was sometimes less efficient. In such a case, users would sense an apparent time lag when the *BX* was executed in the background. We have to emphasize that not only our approach but also other model-transformation-based approaches may suffer the problem. It is because the transformation engine performs pattern matching, a time-consuming operation, frequently during execution. To evaluate the performance of our approach, we carried out two experiments on a 64-bit Windows 7 with an Intel Core i5-2520M and 8 GB DDR3-1333 RAM. The first experiment is based on the Petri Net editor introduced in Subsection 7.1. We constructed 100 different Petri Net models. Each 10 of them compose a group, i.e. there are 10 groups. Each group has the same size (the size is defined as the number of the elements in the model). The model size of the ith group is $10 \times i$. This experiment focused on two types of time consumption: the loading time and the updating time. The loading time is the time spent when the first time the visual tool executes the *BX* after the data is loaded from disk. The updating time is the time spent to propagate the changes after the *domain* or the *diagram* is modified. Figure 16(a) shows the average loading time (ALT) and the average updating time (AUT) of each group. ALT of group-1 is 6.4 ms, and ALT of group-10 is 1795.7 ms. From the result, we learn that ALT of our approach is acceptable, since a two-seconds loading time is very common. AUT ranges from 3.7 ms to 1939.5 ms. Obviously, AUT becomes greater than ALT when the model is greater because when propagating the changes the transformation engine checks the existing correspondences first and then executes the whole *BX* again to deal with the new correspondences. From the result, we can learn that if the model size is no more than 70, AUT is acceptable (around 500 ms); otherwise, the user may notice an apparent time lag each time the *domain* or the *diagram* is modified. Fortunately, many papers [13–16] have discussed and proposed some approaches to incremental model transformation. We will apply them to improve the execution efficiency of our tool support in the future work. The second experiment is based on the Java method Call Graph Tool introduced in Subsection 7.2. We collected 15 Java source code files from open source projects of Eclipse, such as Draw2D, GEF, and SWT. We evenly divided them into 3 groups. The files in group-1 contain about 500 lines of codes (LOC); the files in group-2 contain about 1 000 LOC; the files in group-3 contain about 2 000 LOC. Figure 16(b) shows the average execution time of each group. According to the figure, the average execution time of this experiment ranges from 52.3 s to 989.9 s, i.e.

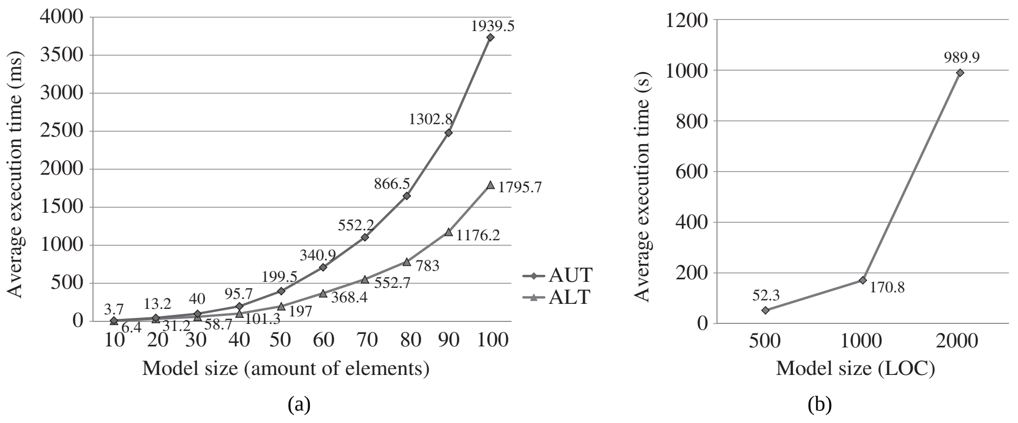


Figure 16 Experiment results of performance. (a) Experiment on Petri Net editor; (b) experiment on Call Graph Tool.

about from 1 min to 17 min. The time cost of this tool rises significantly when LOC increases. This is because the Java model created from a Java source file contains thousands of elements, e.g., the model created from a file containing 486 lines owns 4 679 elements in our experiment. To reduce the time cost, we can apply some preprocessing approaches, e.g., removing the elements irrelevant to method call when creating the Java model. This can improve the efficiency of the *BX*.

Complexity. Let us consider whether the *BX* is so complex and difficult to develop that the total cost could exceed the cost of coding and using and GMF. First, we believe that using *BX* requires fewer development costs than coding. There are two main reasons: 1) the *BX* employed in our approach is declarative, which is more concise than imperative source codes; and 2) the two operations $\mathcal{P}_{\mathrm{SV}}$ and $\mathcal{P}_{\mathrm{VE}}$ can be realized by a single *BX* as we have discussed in Subsection 7.1. Second, using GMF may be simpler than using *BX* because in GMF, we do not have to write any codes. However, compared to our approach, GMF loses the flexibility and the ability of supporting complex mappings as discussed in Subsections 7.2 and 7.3. As known to all, we always need to balance simplicity against flexibility and functional powerfulness in practice. Besides, we can integrate some advantages of GMF into improve our approach. For example, we can also provide some predefined mappings for users, which can be translated into *BX* rules automatically, to simplify the work of specifying *BX*.

8 Related work

A lot of meta-CASE and metamodeling approaches (as well as tools and frameworks) have addressed the problem of specifying and maintaining the *d-d relation* of a visual language:

GMF is a generative framework for domain-specific visual languages. It provides a separation of the abstract syntax (the *domain*) and the concrete syntax (the *diagram*. And, a Mapping Metamodel, which contains a set of predefined one-to-one mappings, is used in GMF to connect the abstract and the concrete syntax. As discussed in Section 2, predefined mappings cannot provide sufficient flexibility and may decrease the power of language definition. Similar to GMF, GME (the Graphical Modeling Environment) [1] also uses some predefined one-to-one mappings to link the abstract and the concrete syntax of a visual language. MetaEdit+ [2] is a commercial metamodeling tool for DSL languages and has been successfully applied to many application domains. However, it also employs a set of predefined mappings to specify how to represent a *concept* in a DSL with a graphical symbol. Compared to GMF and MetaEdit+, our approach does not employ any predefined mappings. The *d-d relation* is specified as a set of *BX* rules that support arbitrary mappings between the *domain* and the *diagram*. Besides, in principle , our approach can support the data metamodel defined with any metametamodel (e.g. MOF, Ecore [9], and KM3 [17]) as long as the transformation engine can interpret the metamodel (in fact, Ref. [18] has proposed a solution of this issue), while MetaEdit+ could only support the metametamodel named OPRR (Object-Property-Role-Relationship).

Some meta-CASE and metamodeling tools utilized model transformation to construct visual tools. However, they usually use model transformation 1) to support syntax-directed editing or 2) to encode the handling behaviors for editing events. In syntax-directed editor, users can only perform the editing operations provided by the editor. For example, DiaMeta (a.k.a. DiaGen) [19,20] uses model transformation to support syntax-directed editors. Each model transformation rule specifies an editing operation. If users want to perform a ceratin operation, they have to invoke the corresponding transformation rule. Model transformation is also used to encode the handling behaviors for editing events. This idea is derived the event-driven system. For instance, ATOM3 [3] uses Triple Graph Grammar (TGG) [21] to specify how to handle an editing event [22], e.g. create or delete a symbol. When a user manipulates the *diagram* in the editor, change events will be detected and sent to an extended TGG engine that can select and execute proper TGG rules to modify the data model. Compared to our approach, we do not use model transformation to specify syntax-directed editing operations or the editing event handling operations directly. We only have to define the correspondences between the *domain* and the *diagram* as a set of *BX* rules, then the *BX* engine will help us to propagate the changes (in two directions) automatically.

ViatraDSM [4] is a domain-specific language engineering framework. It combines a mapping metamodel extended form GMF Mapping Metamodel with live-transformation-based model synchronization to specify and maintain the connection between the abstract and the concrete syntax. The predefined mappings contained in ViatraDSM mapping metamodel are used to define the simple mappings, while complex mappings are handled by live transformations [14]. A live transformation can incrementally react to various changes of models. Similar to an event handling rule in ATOM3, a live transformation rule contains a *trigger* that specifies a kind of change and an operation that tackles the change. ViatraDSM uses live transformations to realize the synchronization between the *domain* and the *diagram*, while our framework uses *BX*. If the *d-d relation* is complex, we have to define more live transformation rules than *BX* rules to propagate the changes, because each live transformation rule can only propagate the changes from one side to the other. This implies that using live transformation may require more development costs than using *BX* to realize *d-d relation*. And the live transformation also suffered the consistency problem.

Fondement et al. [23] proposed a scheme-based approach to concrete syntax definition. In their approach, graphical symbols are defined as a Scalable Vector Graphics model. Then, the symbols and the abstract syntax are connected with a set of display manager classes. A display manager class "declares some attributes and operations what helps to lift up the abstraction level on which the syntax definition is given" [23]. A display manager class may be attached some OCL constraints, which are used as the criteria for a syntactically correct representation. The main difference from our approach is that OCL expressions occurring in Fondement's approach are only used to *check* the syntactical correctness of the visual representation. Developers still have to implement the *d-d relation* via coding. However, in our framework, OCL/MOL expressions will be evaluated directly to realize the *d-d relation*, i.e. to check, to propagate, and to enforce the *d-d relation*.

Besides, many authors discussed bidirectional model transformation. Some of them focused on the conceptual and the theoretical aspect of *BX* [5–7], while others concentrated on the algorithm and the implementation issues [8,13]. Basically, this paper did not extend the concepts and theories of *BX* and model synchronization. And we also employed the state-of-the-art model synchronization approach to maintain the *d-d relation* at runtime. We only extended the specification and the execution of the *BX* rule to support the complex mappings in *d-d relation*.

Song et al. [24] proposed a bidirectional-transformation-based approach to supporting runtime software architecture. They employed *BX* the realize the synchronization between the runtime system model and the runtime architecture model. To some degrees, the *domain* of a visual editor can be considered as the runtime system, while the *diagram* can be regarded as the runtime architecture. Hence, their job and our approach share some common ground. However, their work mainly focused on providing the runtime software architecture support to the existing systems in order to facilitate dynamic configuration, evolution, self-adaption of a running system, while our work focuses on the problems in software visualization and visual editing.

9 Conclusion and future work

In this paper, we propose a bidirectional-model-transformation-based framework to software visualization and visual editing. The framework provides a full separation between the *domain* and the *diagram*, and helps developers to realize the *d-d relation flexibly* and *consistently*, by connecting the *domain* and the *diagram* with *BX*. The main contributions of this paper can be summarized as follows: 1) we propose a model-based approach to the definition and manipulation the *diagram* of a visual tool; 2) we propose a *BX*-based approach to specifying and maintaining the *d-d relation*; 3) we implemented a tool support PKUGTP and reported four case studies to evaluate the feasibility of our approach.

In the future, we plan to extend the *BX* technique, e.g. the compensation mechanism as mentioned in Subsection 7.2, to support more complex mappings in *d-d relation.* Besides, we will also improve the execution efficiency and the usability of our tool support. Finally, our framework will be applied in more practical scenarios.

Acknowledgements

This work was supported by National Basic Research Program of China (973 Program) (Gant No. 2013CB329606), National High-tech R&D Program of China (863 Program) (Gant No. 2012AA011202), National Key Technology Research and Development Program of the Ministry of Science and Technology of China (Grant No. 2011BAK08B04), National Natural Science Foundation of China (Grant Nos. 61300009, 61272159, 60773137, 60573112), Beijing Natural Science Foundation (Grant No. 4122036), China Postdoctoral Science Foundation Funded Project (Grant No. 2013M540050) and Fundamental Research Funds for the Central Universities (Grant No. FRF-TP-13-014A). Thanks to Prof. XIONG YingFei and Mr. YE ZhiYi from PKU, Prof. SUN ChangAi from USTB, and Mrs. OU XianTing from Chinada.

References

1 Davis J. GME: the generic modeling environment. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM, 2003. 82–83

2 Kelly S, Lyytinen K, Rossi M. MetaEdit+: a fully configurable multi-user and multi-tool CASE and CAME environment. In: Proceedings of 8th International Conference on Advanced Information Systems Engineering, Heraklion, 1996. 1–21

3 De Lara J, Vangheluwe H. AToM3: a tool for multi-formalism and meta-modelling. In: Proceedings of 5th International Conference on Fundamental Approaches to Software Engineering, Grenoble, 2002. 174–188

4 Ráth I, Ökrös A, Varró D. Synchronization of abstract and concrete syntax in domain-specific modeling languages. Softw Syst Model, 2010, 9: 453–471

5 Czarnecki K, Foster J N, Hu Z J, et al. Bidirectional transformations: a cross-discipline perspective. In: Proceedings of Theory and Practice of Model Transformations, Zurich, 2009. 260–283

6 Diskin Z, Xiong Y F, Czarnecki K, et al. From state- to delta-based bidirectional model transformations: the symmetric case. In: Proceedings of Model Driven Engineering Languages and Systems, Wellington, 2011. 304–318

7 Stevens P. Bidirectional model transformations in QVT: semantic issues and open questions. In: Proceedings of Model Driven Engineering Languages and Systems, Nashville, 2007. 1–15

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

9 Steinberg D, Budinsky F, Merks E, et al. EMF: Eclipse Modeling Framework. 2nd ed. Indianapolis: Addison-Wesley Professional, 2008

10 Heidenreich F, Johannes J, Seifert M, et al. JaMoPP: the Java Model Parser and Printer. TU Dresden Technical Report TUD-FI09-10. 2009

11 Blair G, Bencomo N, France R. Models@run.time. Computer, 2009, 42: 22–27

12 Alanen M, Porres I. Difference and union of models. In: Proceeding of the Unified Modeling Language, San Francisco, 2003. 2–17

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

14 Hearnden D, Lawley M, Raymond K. Incremental model transformation for the evolution of model-driven systems. In: Proceeding of Model Driven Engineering Languages and Systems, Genova, 2006. 321–335

15 Johann S, Egyed A. Instant and incremental transformation of models. In: Proceedings of 19th International Conference on Automated Software Engineering, Linz, 2004. 362–365

16 Beaudoux O, Blouin A, Barais O, et al. Active operations on collections. In: Proceeding of Model Driven Engineering Languages and Systems, Oslo, 2010. 91–105

17 Jouault F, Bézivin J. KM3: a DSL for metamodel specification. In: Proceeding of Formal Methods for Open Object-Based Distributed Systems, Bologna, 2006. 171–185

18 Wilke C, Thiele M, Wende C. Extending variability for OCL interpretation. In: Proceeding of Model Driven Engineering Languages and Systems, Oslo, 2010. 261–375

19 Köth O, Minas M. Generating diagram editors providing freehand editing as well as syntax-directed editing. In: Proceedings of International Workshop on Graph Transformation, Berlin, 2000. 32–39

20 Minas M. Generating visual editors based on fujaba/moflon and diameta. University Paderborn, Technical Report tr-ri-06-275, 2006

21 Schürr A. Specification of graph translators with triple graph grammars. Lect Note Comput Sci, 1995, 903: 151–163

22 Guerra E, de Lara J. Event-driven grammars: relating abstract and concrete levels of visual languages. Softw Syst Model, 2007, 6: 317–347

23 Fondement F, Baar T. Making metamodels aware of concrete syntax. In: Proceedings of Model Driven Architecture Foundations and Applications, Nuremberg, 2005. 190–204

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