Applying Model Transformation By-Example on Business Process Modeling Languages*

Michael Strommer[1], Marion Murzek[2]**, and Manuel Wimmer[1]

Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology, Austria
{strommer|wimmer}@big.tuwien.ac.at
Womens Postgraduate College of Internet Technologies
Institute for Software and Interactive Systems
Vienna University of Technology, Austria
murzek@wit.tuwien.ac.at

Abstract. Model transformations are playing a vital role in the field of model engineering. However, for non-trivial transformation issues most approaches require imperative definitions, which are cumbersome and error-prone to create. Therefore, Model Transformation By Example (MTBE) approaches have been proposed as user-friendly alternative that simplifies the definition of model transformations. Up to now, MTBE approaches have been applied to structural models, only. In this work we apply MTBE to the domain of business process modeling languages, i.e., Event-driven Process Chains and UML activity diagrams. Compared to structural languages, business process modeling languages cover static semantic constraints, which are not specified in the metamodel. As a consequence, reasoning on the abstract syntax level is not sufficient. The contribution of this paper is to extend existing MTBE approaches by (1) new alignment operators on the user level and by (2) new reasoning algorithms operating on the concrete syntax level. Our extensions to MTBE then allow for more expressiveness in the user mappings as well as improved transformation code generation.

Key words: Business Process Models, Model Transformation

1 Introduction

With the rise of model engineering, model transformations have been steadily put in the limelight in the past five years. Growing tool support indicates, that model transformations are not only attractive for researchers, but also for industrial parties accommodating to their customers needs. Model transformation

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.0
** This research has been partly funded by the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002.

scenarios include transformations between different refinement levels of models (*vertical transformations*) as well as transformations between different modeling languages which rely on the same abstraction level (*horizontal transformations*). Languages used for defining model transformations are ATL [7], QVT [13], Triple Graph Grammars, XSLT, and even general purpose languages such as Java [1]. Many model transformation languages are hybrid, meaning that besides a declarative style, an imperative style is provided for problems that cannot be solved by the provided declarative features. Nevertheless, the development of imperative fragments is often a tedious and error-prone task. In contrast, declarative solutions are compact descriptions but not always intuitive to find. Therefore, approaches such as Model Transformation By-Example (MTBE) came up. Similar to other By-Example methodologies such as Query By-Example (QBE) [18], probably the most prominent one, MTBE aims at generating most of the transformation rules automatically, given that the user provides manually defined mappings between example models covering the concepts of their individual languages. A short overview of our MTBE approach will be given in the next section.

Instead of focusing on the domain of *structural modeling languages*, what has been done in previous investigations [17], [16], in this paper we concentrate on *behavioral modeling languages*. More specifically, we apply MTBE on the domain of business process modeling (BPM), which, up to our best knowledge, has not yet been subject to the MTBE approach. The definition of requirements for MTBE in the context of business process modeling and how they can be met in terms of proper generation of transformation rules comprise the main contribution of this paper. Therefore, we present *main challenges encountered in business process (BP) model transformations*, and how these challenges can be tackled by extending already proposed MTBE *mapping operators* and *reasoning algorithms* in order to allow a more sophisticated model transformation code generation. Furthermore, the proposed extensions are explained by a running example in which two prominent BP modeling languages are used, namely the UML Activity Diagram and Event Driven Process Chains.

The remainder of this paper is structured as follows. First we provide a short introduction to MTBE in Section 2. Subsequently, we describe in Section 3 why we focus on BP models and present two common BP modeling languages which are later used in a running example. Model transformation problems encountered in BPM and how these can be solved is explained in Section 4. In Section 5 we discuss mapping and transformation issues in the area of BPM in a by-example manner and how MTBE might help to eradicate those problems. In Section 6 we discuss related approaches, and finally, in Section 7 we provide conclusion and future work.

2 Overview of MTBE for structural Modeling

In this section we give a brief outline of our MTBE approach we introduced in [17]. We have recognized two main issues in conjunction with the task of defining model transformations. The first one is about the gap between the way a person thinks about models and the way a computer represents those models internally. And the second issue is about the way concepts are represented in the metamodel (MM), i.e., whether one needs to have expert knowledge to identify those concepts or not. We call the phenomenon of hidden concepts in a metamodel *concept hiding* [8]. Having those issues in mind one can easily accept the fact, that the task of creating model transformation rules is not a user-friendly one. This is why we have come up with the idea of MTBE, that can be seen as a semi-automatic approach for the generation of model transformation rules. One of the main benefits of MTBE is the shift in abstraction. Mostly all of the proposed model transformation approaches operate on the abstract syntax (AS), although modelers might not be familiar with the abstract syntax. Therefore, we intend to make the transformation task more concrete and operate on a level the modelers or designers are familiar with, i.e, on the concrete syntax (CS). To make this semi-automatic methodology work, four conceptual steps have to be performed:

Definition of Model Mappings. The user has to manually define *one-to-one* correspondences between two sample models on the concrete syntax layer. These two models serve as examples and consequently, should cover as much language concepts as possible. This step is to be supported by a graphical mapping editor.

Derivation of Metamodel Mappings. The model mappings serve as input to derive appropriate mappings between the metamodel elements. These metamodel mappings are either *full equivalence* or *conditional equivalence* mappings. With the term conditional mapping we mean a mapping that involves a constraint specified in Object Constraint Language (OCL). These OCL constraints actually stem from the notation, i.e., the conceptual mapping between a concrete syntax and an abstract syntax element. The notation is defined as

$$Triple := < as\_E, cs\_E, const(as\_E)? > \tag{1}$$

where as_E is some abstract syntax, e.g. Class object, and cs_E some concrete syntax element, e.g. a rectangle figure with a label. The metamodel mappings are derived via reasoning algorithms which take the models, the model mappings (cf. Step 1), as well as the metamodels as input.

Model Transformation Generation. The mappings between the metamodel elements are used to generate model transformation code. Note that we do not stick to one specific transformation language, although we decided to use ATL for demonstration purposes and early prototyping. The ATL transformations are produced from a code generation component, which takes the metamodel mappings of Step 2 as input.

User Refinements. The last step concerns testing of the generated transformation rules and their refinement. This is due to the fact, that some transformations are not derivable by MTBE so far and must be completed manually. Furthermore, we believe that MTBE represents a typical iterative approach, because the example models used for defining the model mappings (cf. Step

1) can be reused for test purposes to gain feedback for adapting the model mappings accordingly.

3 Models for Business Processes

Business process models are in use for quite a long time and continue to gain importance as support from the software engineering field is improving significantly. Particularly model engineering fosters research in the area of BPM. There exist several metamodels for existing languages in order to raise there acceptance and tool interoperability. Due to this growing interest in BPM and proper tool support, we believe MTBE can be advantageous for specifying model transformations between BP models. Usually BP models cover various perspectives as e.g. described in [3]. The following two BP modeling languages we choose to use in our case study presented in Section 5, however, cover only the behavioral perspectives of BPM.

3.1 UML 2.1 Activity Diagram

The UML 2.1 Activity Diagram (UML AD) [14] is a specification of the Object Management Group. The metamodel in the left part of Figure 1 depicts an excerpt of the UML AD language, namely the basic control flow elements which are used for modeling BP models, as well as the concrete syntax. The central

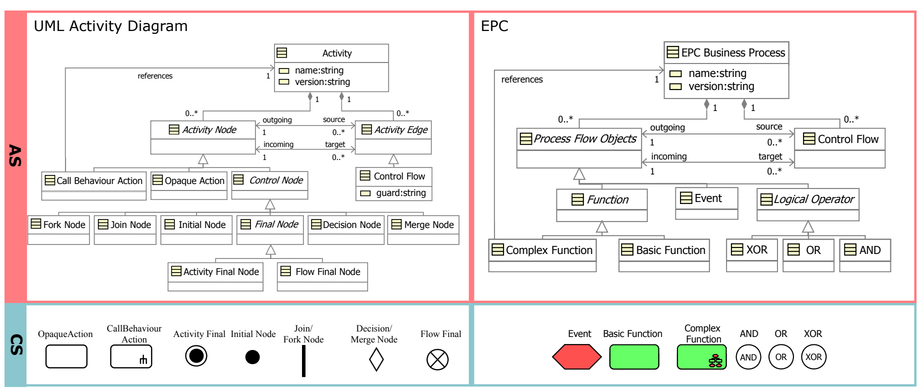

Fig. 1. Parts of the UML 2.1 AD and EPC meta models and their concrete syntax

element is the *Opaque Action*, which is used to model the activities within a process. The *Call Behavior Action* represents the concept of a sub process call. *Control Nodes* are used to structure the process. More specifically, a *Fork Node* and a *Join Node* express a concurrent flow as well as a *Decision Node* and a *Merge Node* to express an alternative flow. The *Initial Node* marks the begin of a process model. The UML AD differs between two final nodes, the *Flow Final Node* (*FFN*) and the *Activity Final Node* (*AFN*). The *FFN* is used to mark the final of a distinct flow, that means if it is reached the remaining tokens in the process will proceed. Whereas the *AFN* marks the end of the whole process which means if it is reached the remaining tokens in the process are killed immediately. The only kind of *Activity Edge* we consider in this work is the *Control Flow*, which is used to connect the *Activity Nodes* to form the flow of control of a process.

3.2 Event-driven Process Chains

Event-driven Process Chains (EPCs) [9] have been introduced by Keller et al in 1992 as a formalism to model processes. In this paper we focus on the main elements, which are used to model the control flow aspect of a BP model. The metamodel and concrete syntax of EPCs are illustrated in Figure 1 on the right.

The *Function* represents an activity. It creates and changes information objects within a certain time. The *Event* represents a BP state and is related to a point in time, it could be seen as passive element compared to the *Function* as an active element [11]. To model a sub process call the *Complex Function* is used. The *Logical Operators* elements are used to structure the proceed of the BP model.

When dealing with EPCs some special modeling restrictions must be considered which are not directly represented in the metamodel. EPCs do not provide a specific element to indicate the begin and the end of a BP model, instead the *Event* is used. *Event* elements are not allowed to be in front of an *OR* and *XOR* element. *Function* and *Event* elements must alternate in the proceed of the BP model and are connected via the *Control Flow*. This feature of the EPC language is in fact a static semantic constraint, which is not specified in the metamodel illustrated in Figure 1. Another restriction in EPCs is that parallel branches as well as alternative branches must be split and merged with the same kind of *Logial Operator*. Again we have to face a static semantic constraint in the context of *Logical Operators*, when it comes to specifying model transformations.

4 Mapping Operators for MTBE in the Light of BPM

During our investigation of BP models we discovered, that there are considerable differences compared to structural models concerning the requirements for MTBE. To transform structural models, one has to be familiar with the notation and hidden concepts in the metamodels, especially when dealing with UML diagrams. Resulting ambiguities on the metamodel layer have to be solved either by reasoning algorithms or user input, as we described in detail in our previous work. Now, with the task of transforming BP models we have to deal with quite different issues, in order to apply our MTBE approach. A lot of interesting aspects concerning the heterogeneity of BP models have been identified in [12]. One of the special requirements coming along with BP models has its root in

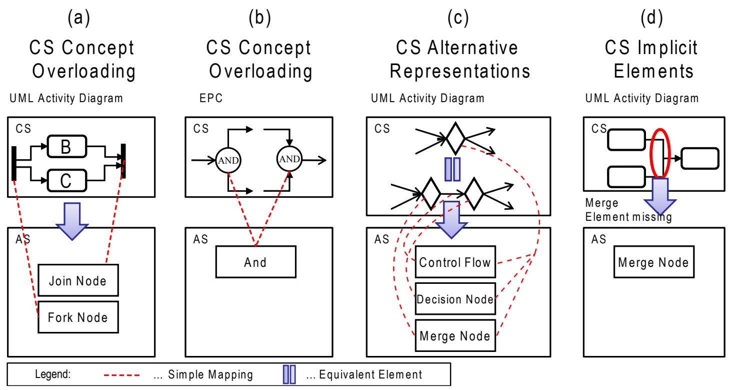

Fig. 2. Overview of BP models heterogeneities

the mapping from concrete to abstract syntax layer (notation) and the number of modeling elements involved on each layer. As we now allow for zero or more elements on each layer in the CS-AS mapping, the notation shown in equation 1 becomes to

$$Triple := < as\_E^*, cs\_E^*, const(as\_E)? > \tag{2}$$

In UML AD we have for example the notation:

$$< \{MergeNode, ControlFlow, DecisionNode\}, \{DecisionMergeFigure\}, \{\} >$$

as is illustrated in Figure 2 (c) for the CS modeling element on the very top. Note that the used modeling construct is here just an abbreviation on the CS layer and could be equivalently expressed by the following pattern of notation triples:

$$< \{DecisionNode\}, \{DecisionFigure\}, \{\} >$$
$$< \{ControlFlow\}, \{ConnectionFigure\}, \{\} >$$
$$< \{MergeNode\}, \{MergeFigure\}, \{\} >$$

We also observed several heterogeneities between modeling languages, which pose further requirements for MTBE. Figure 2 gives four examples for the peculiarities we found in the two BP modeling languages we introduced in Section 3. Examples *a* and *b* in Figure 2 depict the case of so called CS overloading in UML AD and EPC. In example *a* we encounter no problems because with the help of the notation we can distinguish between the two concepts join and fork despite the CS overloading. In example *b* CS overloading represents a real challenge for MTBE as two equal CS elements, but in fact featuring two different meanings, are mapped to the same AS element.

When we have to deal with alternative representations in the CS, see Figure 2 *c*, we can use the notation in MTBE to find them. The challenge arises not until we have to map two languages, where one consists of such variation points in the CS. Example *d* in Figure 2 shows the possibility in UML AD to merge parallel flows implicitly by omitting a merge/join node, i.e., we have no mapping from the AS to the CS.

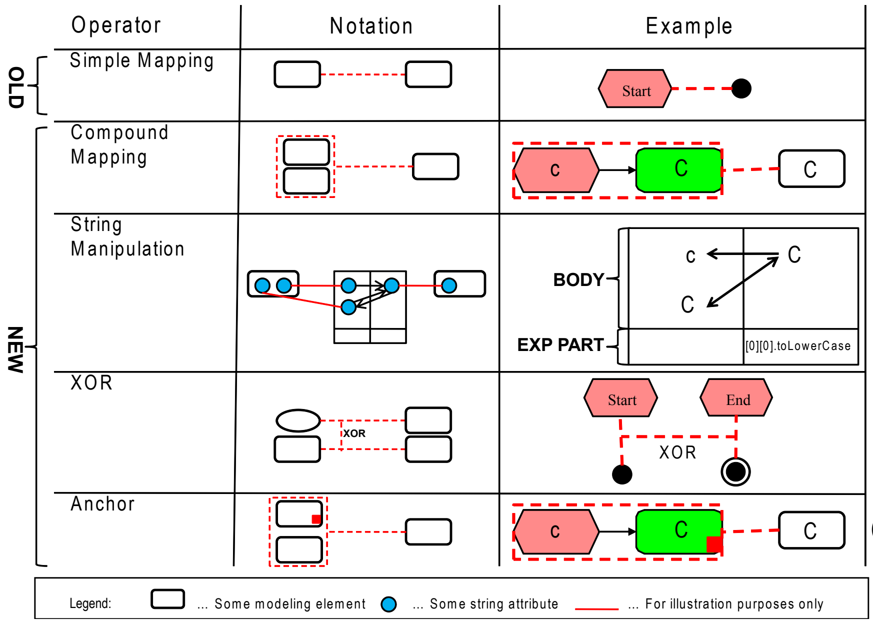

Fig. 3. Overview of MTBE Mapping Operators

In the following we present new mapping operators and transformation heuristics which resolve heterogeneities, as expressed in the examples *a*,*b*, and *c*, in Figure 2 are faced. Unfortunately, up to now we are not able to cope in MTBE with implicit elements as shown in example *d*. The problem here is twofold. First we have to address the question how to map these implicit elements on the concrete syntax layer. And second we have to adjust the code generation process accordingly.

So far our MTBE approach as presented in [17] has only dealt with *simple 1:1 mappings* on the concrete syntax, see Figure 3 *a*. In case of BP model transformations it is necessary to introduce new kinds of mapping operators. Based on the specialties and problems stated above we developed new operators. In the following we describe the semantics of these new operators to provide a notion of how they can be used. However, we still have to develop some formal specification for these operators.

The first new operator is the *compound mapping operator* (cf. Figure 3 *b*). This mapping operator allows for n:m mappings on the CS layer. Although we encountered only 1:n mappings so far, we want the user to have the feature of n:m mappings. With this mapping operator we intended to support the mapping of common work flow patterns, such as the one we show in the corresponding example for this operator.

Along with the compound mapping operator comes a *string manipulation operator*, that works in the context of compound mappings but is not restricted to them. A first notation approach is shown in Figure 3 *c* together with an example. Note, that this operator is used for *Attributes* specified in the metamodel, which are represented as labels in the model. This operator consists of two main components, i.e., a body and an expression part, each separated into left and right hand side. The two body parts consist of a list containing references to *Attributes*, that are going to be mapped. Furthermore each *Attribute* of the body manages a list containing the unidirectional mappings from itself to some other *Attributes*. During the transformation rule generation from one language to the other only one list of mappings is of interest. In the expression part one can use some simple string operations or regular expression. In the example given in the Figure above we apply a *toLowerCase* operation on the first mapping of the first *Attribute* on the right hand side.

For the *XOR operator* depicted in Figure 3 *d* there are two ways to use it, i.e., in an explicit way or in an implicit way. In Figure 3 we only illustrated the explicit use of this operator. In general an XOR mapping shall indicate that only one element should be created although one CS element in one language is mapped to more than one element in the other language. Omitting the XOR in the example in Figure 3 *d* would lead to the creation of an *Initial* and an *Activity Final Node* for every *Event* that is matched by the corresponding transformation rule. When using the XOR operator in an implicit way the whole issue is hidden from the user. Instead all XOR mappings are derived automatically in the metamodel as will be shown in Section 5. The drawback of this approach is that one looses the possibility of multiple object creations as mentioned before.

At last we introduce the *anchor operator*. The notation and an example are given in Figure 3 *e*. The anchor operator marks the element, which the transformation rule shall use as single source pattern element. It is thus always used in conjunction with the compound mapping operator, which usually leads to the creation of multiple source pattern elements in the rules. This operator proved very useful in the derivation of ATL rules for our heuristics.

5 Case Study - MTBE with UML AD and EPC

Our MTBE approach for the domain of Business Process modeling can be best explained in a by-example manner. Therefore, we use the two BP languages EPC and UML AD described in Section 3. For demonstration purposes we show what the generated code would look like in ATL. Although the example given in Figure 4 is rather simple, it still covers a lot of interesting aspects for MTBE.

For the case study we assume that on the concrete syntax layer in EPC's *Events* and *Basic Functions* to always occur pairwise connected through a *Control Flow* edge. Furthermore, in UML AD modeling it could be possible to omit a *Join* node and therefore model joins implicitly. However, in our first MTBE approach for BPM we do not jet cope with implicit joins or merges.

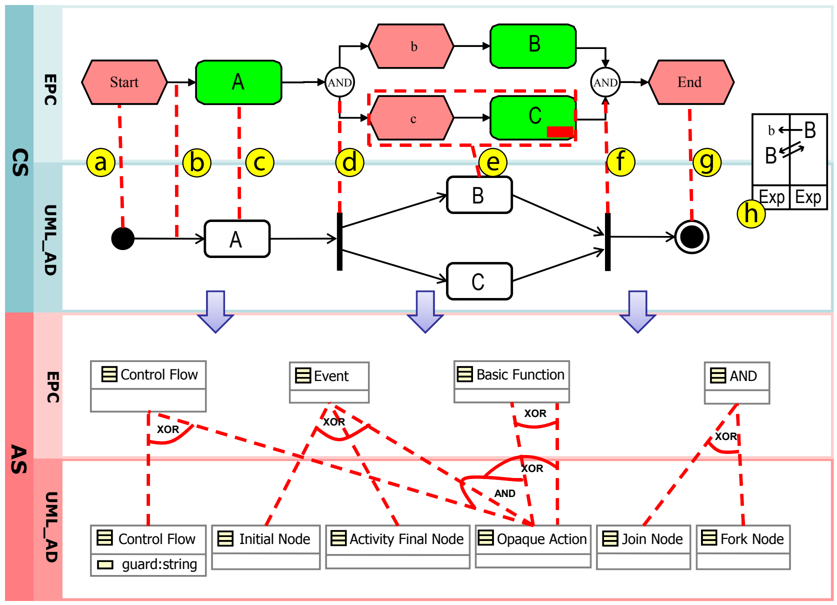

Fig. 4. Mapping EPC and UML Activity diagram - CS + AS perspectives

5.1 Model and Metamodel mappings

As a first step one has to define manual mappings between two languages, which the transformation model shall be derived from. In the example in Figure 4 we specified six mappings that capture all concepts being used in the two sample models. Mappings *a,b,c,d,f,* and *g* are of type simple mapping.

Mapping *e* is of type compound mapping with multiplicity 1:3. Consequently, whenever the pattern *Event*, *Control Flow*, *Basic Function* is matched this corresponds to a single *Opaque Action*. We also marked the *Basic Function C* in our compound mapping as anchor element, which has implications specific to transformation code generation. In our case the ATL code generator would use this *Basic Functions* metamodel element as single source pattern element instead of using multiple source pattern elements. During our implementation attempts we realized, that an anchor feature can be desirable in some transformation scenarios.

Mapping *h* in our example takes care of the labels used in *Events*, *Basic Functions* and *Opaque Actions*. To maintain usability this string manipulation operator is used in a separate modeling element and references the involved labels. To define string equivalences one can use only unidirectional mappings, which will be applied transforming from one set of labels to another. An optional expression allows us for example in mapping *h* to apply a *toLowerCase()* operation on the first mapping of the right hand side set of labels.

In EPC's there are no distinct metamodel elements nor distinct concrete syntax elements for start and end nodes, although these concepts are used in the modeling language implicitly. In UML AD we do have explicit concepts for start and end nodes both, in the model and the metamodel. If a transformation from EPC2UML_AD has to be performed the transformation model must know how to distinguish between start and end nodes even without having these concepts specified in EPC. We will elaborate on this issue in 5.3.

To keep our illustration in Figure 4 transparent and clear we omitted the mappings between CS and AS. Also these mappings are quite straightforward to define, as there are no constraints specified in the notation.

At last the mappings between the two metamodels can be derived from the user mappings and the notation. To highlight the existence of a compound mapping in the metamodel we marked the three involved mappings with an *and* operator. On the metamodel mapping level we now make use of our new XOR operator we introduced in Section 4. To keep the mapping task user-friendly the XOR between mappings can be reasoned automatically based on information in the metamodels. Whenever a meta class contains at least two outgoing mapping edges, an XOR relation can be set in an implicit way.

5.2 How to make Mappings executable

As the automatic generation of transformation rules is a difficult task, we do not claim to support fully automatic rule generation. Instead we believe in a semi-automatic approach. To face the new domain of business process models we implemented a methodology, which can be best compared to Architecture-Centric MDSD [15]. First of all we have implemented correct ATL transformation code, which acts as reference implementation. Thereby we have avoided imperative code sections and concentrate on coding in a declarative fashion.

In the next step we have developed the mapping operators described in Section 4. During this step we have turned our attention to the user-friendliness.

Next we have looked at the example models, the user mappings and the metamodels and tried to deduce the reference implementation. Code segments that could not be deduced automatically then lead to further refinement of the underlying heuristics. After refinement we tried again to deduce the reference implementation. This process can be seen as an iterative way to deduce heuristics on how to generate ATL transformation rules from a given set of models, metamodels and user mappings. The aim of this process is to optimize the relation between user-friendly mapping operators and the ability to generate executable transformation rules.

5.3 ATL Code generation

Due to space limitations we will not expand on every aspect of the ATL code generation for the example in Figure 4. Instead we focus on the most interesting and challenging parts, only. The three ATL code snippets presented in the following paragraphs transform from EPC models to UML ADs. However, the example mappings provided by the user, also allow for UML AD 2 EPC transformation code generation.

Event2InitialNode and Event2FlowFinal. We already mentioned that we somehow have to distinguish between *Events*, that can be either normal *Events*, *Start Events* or *End Events*, to properly generate elements in UML Activity models. In our previous work we tried to overcome mapping and thus generation problems by means of reasoning on the metamodel layer. For business process models it seams to be more appropriate to do reasoning on the model layer.

Listing 1.1. Event2InitialNode and Event2FlowFinal

```
rule StartEvent2InitialNode {
    from
            s : EPC!Event ( s.incoming->size() = 0 )
    to
            i : Activity!InitialNode (...)
}

rule EndEvent2FlowFinal {
    from
            e : EPC!Event ( e.outgoing->size() = 0 )
    to
            f : Activity!FlowFinalNode (...)
}
```

When the user maps two elements that are completely identical in the metamodel in one language, but correspond to two different elements in the other language, reasoning algorithms have to examine the graph structure in the example model. In our *Event* example the algorithm would have to determine, that *Start Events* do not have any incoming *Control Flows* and that *End Events* do not have any further outgoing *Control Flows*. Listing 1.1 shows the corresponding ATL rules with proper conditions in the source pattern. This addresses mappings a and g in Figure 4.

In ATL it is not possible to match an element more than once, i.e., to have more than one rule applied. Therefore, whenever a metamodel element occurs in at least two source patterns, we have to make sure that only one rule is matched. In the example above this would only be possible, if the user would model an *Event* without any *Control Flow* connected to it. Of course this would already violate some validity constraint. However, the OCL constraint to check for multiple matching would look like in ATL as follows:

$$
\begin{aligned}
&EPC!Event.allInstances()->select(e|e.incoming->size()=0)->asSet() \\
&->intersection(EPC!Event.allInstances()->select(e|e.outgoing->size()=0)) \\
&->size()=0
\end{aligned}
\tag{3}
$$

EventControlFlowFunction2OpaqueAction. Now we want to cope with the user mapping e of Figure 4. From the model itself and especially the metamodel we know, that in EPC there are three distinct concepts involved whereas in UML Activity diagrams only one concept is affected. For this reason we use a new feature coming along with ATL 2006, i.e., the matching of multiple source pattern elements, see Listing 1.2. Note that the returned set of elements from matched multiple source pattern elements corresponds to the cartesian product.

Listing 1.2. EventControlFlowFunction2OpaqueAction

```
rule EventControlFlowFunction2OpaqueAction {
    from
            ev : EPC!Event,
            c : EPC!ControlFlow,
            f : EPC!BasicFunction (
                c.target = f and c.source = ev and
                ev.incoming->size() <> 0 and
                ev.outgoing->size() <> 0
            )
    to
            o : Activity!OpaqueAction (
                name <- f.name,
                parent <- f.parent,
                incomming <- ev.incoming
            )
}
```

This is why we have to give a guard clause ($c.target = f\ and\ c.source = ev$) to select only those elements we are interested in. This is similar to a join in SQL. To generate this "join" condition automatically we have to assume that elements in a compound mapping are always connected through proper link elements. A reasoning algorithm can check for the existence of links and build conditions that must hold for the pattern to match.

There are two more conditions given in Listing 1.2 that must evaluate to true if this rule shall be executed. This condition originates from the XOR constraint we face in the metamodel between the mappings *Event_InitialNode*, *Event_ActivityFinalNode* and *Event_OpaqueAction*, which is actually part of a compound mapping indicated by an *and*. To avoid matching a rule twice we can just take the conditions we have deduced in the previous two ATL rules and insert their negation, i.e. $ev.incoming- > size() <> 0\ and\ ev.outgoing- > size() <> 0$. The idea of inserting the negation of already existing conditions in other rules can be seen as general heuristic.

And2Fork and And2Join. In Figure 2 *b* we referred to the problem of concept overloading in the CS, which we face in the transformation from an EPC to a UML AD model. We know, that the simple mappings d and f are actually in an XOR relationship, which is determined from the deduced mappings between the metamodel elements. This transformation difficulty was also the reason why we introduced the XOR operator. The user mappings together with the derived XOR constraint are not yet sufficient to provide for a heuristic capable of generating valid transformation code. What we need in this special case of concept overloading is an algorithm performing "local reasoning" on a specific node and compare the results with the ones from another one. The differences in the properties found between these nodes are then used to distinguish between them. In our example we determined for the class *And* mapped to class *Fork Node* there has to be only one incoming *Control Flow* and at least two outgoing *Control Flows* on the CS layer if the rule And2Fork shall be applied. For the class *And* mapped to class *Join*

Node the opposite has to be true if the rule And2Join is supposed to match. Both rules are given in Listing 1.3

Listing 1.3. And2Fork and And2Join

```
rule And2Fork {
    from
            an : EPC!AND (
                an.incoming->size() = 1 and
                an.outgoing->size() > 1
            )
    to
            fn : Activity!ForkNode (...)
}

rule And2Join {
    from
            an : EPC!AND (
                an.incoming->size() > 1 and
                an.outgoing->size() = 1
            )
    to
            jn : Activity!JoinNode (...)
}
```

5.4 Critical Discussion

The code for the examples above is generated in a heuristic way and we believe that in many cases and languages there is no great effort for code refinement necessary. However, there are limitations of MTBE we want to briefly discuss. In our language definition of EPC we assumed, that every *Basic Function* is directly preceded by an *Event*. But it may be possible in our example to place the *Events b* and *c* as one *Event* in front of the *And* split. This is another static semantic constraint one can only capture in a natural language description of EPC. As an example we refer to Figure 5 *a*.

Due this alternative way of positioning concrete syntax elements our rule defined in Listing 1.2 would no longer match any of the elements in such small models. To solve this problem MTBE could again be applied on this new EPC example model and map the concepts of interest again. For the example given in Figure 5 *a* we would have to map the *Basic Function* located between the two *And* elements to an *Opaque Action* in our EPC 2 UML AD mapping scenario. Because of the XOR constraints later derived in the metamodels a heuristic could be applied to prevent multiple rule matching and select the rules properly. The mapping of the *Event a* remains however an open issue.

In Section 4 we presented the heterogeneity of alternative representations in CS. In UML AD we could model a join followed by a fork the way shown in Figure 5 *b1*. This representation is just an abbreviation, which we want to map in our example to EPC, where this form of abbreviation is not possible. From the users point of view it is sufficient to simply draw the compound mapping *a*. For both modeling constructs we have again drawn the corresponding metamodel elements and also their mapping to the CS (notation). As one can easily see it is not possible to determine from these notations and the compound mapping *a* how the elements in the metamodel shall be mapped from one language to the other.

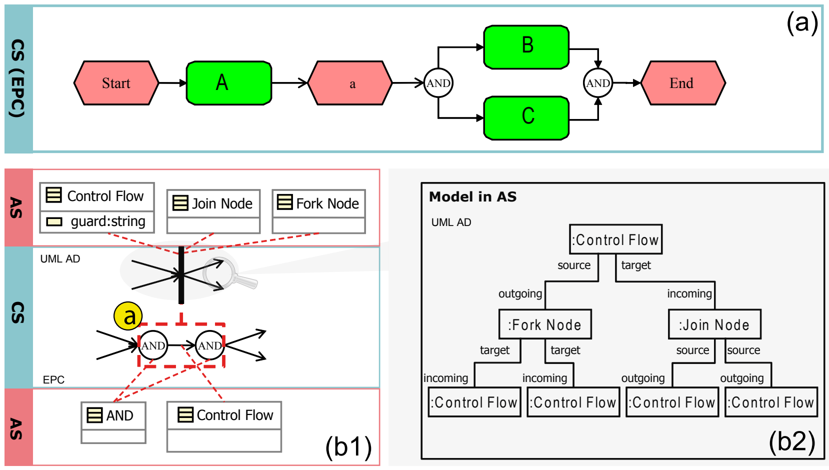

Fig. 5. Challenges for MTBE in Business Process Models

Again we can apply local reasoning algorithms operating on the model expressed in AS to find out what elements possibly go together. The UML AD example model given in Figure 5 *b1* is also illustrated in AS (see Figure 5 *b2*), modeled in UML object diagram concrete syntax. We can now reason on this representation of the model and try to find out how the single metamodel elements have to be mapped to the elements in EPC. For example we learn from this graph that *Fork* and *Join Nodes* have a single *outgoing* and *incoming* edge, respectively. The heuristic is similar to the one that copes with mappings d and f in the first example, cf. Figure 4.

6 Related Work

To our best knowledge, there exists no approach for finding semantic correspondences between business process models so far. However, there exists general approaches that allow the definition of semantic correspondences between two (meta)models, which have been applied in the area of structural models. The first approach is a *model-based* approach from Varró, while the second approach from Fabro et al is *metamodel-based* which allows automatically finding correspondences directly between metamodels.

Model-based: Parallel to our MTBE approach [17] Varró proposed in [16] a similar approach. The overall aim of Varró's approach is comparable to ours, but the concrete realizations differ from each other. Our approach allows the definition of semantic correspondences on the concrete syntax, from which ATL rules can then be derived. In contrast, Varró's approach uses the abstract syntax to define the mappings between source and target models, only. The definition of the mapping is done with reference nodes leading to a mapping graph. To transform one model into the other, graph transformation formalisms [5],[6] are used. Furthermore, there is no publication on applying Varro's approach on business process models which is the general aim of this paper.

Metamodel-based: Orthogonal to MTBE there exists the approach of using matching transformations combined with weaving models [4] in order to generate an ATL transformation model. Matching transformations are defined such that they use common matching algorithms or modifications, e.g., similarity flooding [10], and then create a weaving model from the calculated similarity values. Afterwards these weaving models are taken as input for another transformation, called higher order transformation (HOT), to produce the desired ATL model describing the transformation rules between two metamodels. Because there will be always some mappings that can not be matched fully automatically, this approach is also to be considered semi-automatic. The model transformation generation process described in [4] currently focuses on using mappings between metamodels and is therefore based on the abstract syntax, while our approach aims at generating model transformation code from M1 mappings. Hence, we have shifted the definition of the mappings from the abstract syntax to the concrete syntax and from the metamodel layer to the model layer.

7 Conclusion and Future Work

In this work we have proposed an MTBE approach for the area of business process modeling languages. Therefore, we extended already existing MTBE techniques in two directions. First, special mapping operators are introduced giving the user more expressivity for defining the model mappings. Second, we introduced reasoning algorithms for each new mapping operator to extend the model transformation code generation.

Concerning future work, we particularly strive for first, the refinement of the proposed MTBE approach and second, its application to the domain of web modeling languages. Concerning the last mentioned direction, we want to apply our current MTBE approach to hypertext models, such as WebML [2], which represent navigation and data flow between hypertext nodes via links and link parameters. We hope that the area of web modeling languages offers new example mapping problems and allows the evaluation of our current insights of MTBE in more detail. Concerning the refinement of the proposed MTBE approach, we want to experiment with different versions of mapping models which possess different levels of granularity, size and modeling patterns. In particular we want to evaluate how much a model can be altered compared to the original model which is mapped by-example, and can still be transformed properly. Therefore, we want to map two models A and B, generate the transformations, and then alter the models two A' and B' and test the generated transformations with the new versions of the models.

References

1. D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier. Sitra: Simple transformations in java. In O. Nierstrasz, J. Whittle, D. Harel,

and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 351–364. Springer, 2006.

2. S. Ceri, P. Fraternali, and M. Matera. Conceptual modeling of data-intensive web applications. *IEEE Internet Computing*, 6(4):20–30, 2002.
3. B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Commun. ACM*, 35(9):75–90, 1992.
4. M. Del Fabro and P. Valduriez. Semi-automatic model integration using matching transformations and weaving models. In *The 22th Annual ACM SAC, MT 2007 - Model Transformation Track*, Seoul, Korea, October 2007.
5. H. Ehring, G. Engels, H.-J. Kreowsky, and G. Rozenberg. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999.
6. T. Fischer, J. Niere, L. Torunski, and A. Zuendorf. Story diagrams: A new graph transformation language based on uml and java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98)*, volume LNCS 1764 (2000), Paderborn, November 1998.
7. F. Jouault and I. Kurtev. Transforming Models with ATL. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.
8. G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer. Lifting metamodels to ontologies - a step to the semantic integration of modeling languages. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems, Genova, Italy, 2006.
9. G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)". Technical report, Institut für Wirtschaftsinformatik Universität Saarbrücken.
10. S. Melnik. *Generic Model Management: Concepts And Algorithms (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.
11. J. Mendling and M. Nüttgens. EPC Modelling based on Implicit Arc Types. In *Proceedings of the 2nd International Conference on Information Systems Technology and its Applications (ISTA)*, 2003.
12. M. Murzek and G. Kramler. Business Process Model Transformation Issues. In *Proceedings of the 9th International Conference on Enterprise Information Systems*, 2007.
13. OMG. *QVT-Merge Group: Revised submission for MOF 2.1 Query/View/Transformation*, version 2.0 formal/05-07-04 edition, 2005.
14. OMG. *UML 2.1 Superstructure Specification*. Object Management Group, http://www.omg.org/docs/ptc/06-04-02.pdf, April 2006.
15. T. Stahl and M. Völter. *Modellgetriebene Softwareentwicklung*. Dpunkt Verlag, March 2005.
16. D. Varró. Model transformation by example. In *9th International Conference on Model-Driven Engineering Languages and Systems (MODELS06)*, Genova, Italy, October 2006.
17. M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards model transformation generation by-example. *In proc. of HICSS-40 Hawaii International Conference on System Sciences, Hawaii, USA.*, 2007.
18. M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.