JTL: A Bidirectional and Change Propagating Transformation Language

Antonio Cicchetti[1], Davide Di Ruscio[2], Romina Eramo[2], and Alfonso Pierantonio[2]

[1] School of Innovation, Design and Engineering
Mälardalen University,
SE-721 23, Västerås, Sweden
antonio.cicchetti@mdh.se
[2] Dipartimento di Informatica
Università degli Studi dell'Aquila
Via Vetoio, Coppito I-67010, L'Aquila, Italy
{davide.diruscio,romina.eramo,alfonso.pierantonio}@univaq.it

Abstract. In Model Driven Engineering bidirectional transformations are considered a core ingredient for managing both the consistency and synchronization of two or more related models. However, while non-bijectivity in bidirectional transformations is considered relevant, current languages still lack of a common understanding of its semantic implications hampering their applicability in practice.

In this paper, the Janus Transformation Language (JTL) is presented, a bidirectional model transformation language specifically designed to support non-bijective transformations and change propagation. In particular, the language propagates changes occurring in a model to one or more related models according to the specified transformation regardless of the transformation direction. Additionally, whenever manual modifications let a model be non reachable anymore by a transformation, the *closest* model which approximate the ideal source one is inferred. The language semantics is also presented and its expressivity and applicability are validated against a reference benchmark. JTL is embedded in a framework available on the Eclipse platform which aims to facilitate the use of the approach, especially in the definition of model transformations.

1 Introduction

In Model-Driven Engineering [1] (MDE) model transformations are considered as the gluing mechanism between the different abstraction layers and viewpoints by which a system is described [2,3]. Their employment includes mapping models to other models to focus on particular features of the system, operate some analysis, simulate/validate a given application, not excluding the operation of keeping them synchronized or in a consistent state. Given the variety of scenarios in which they can be employed, each transformation problem can demand for different characteristics making the expectation of a single approach suitable for all contexts not realistic.

Bidirectionality and change propagation are relevant aspects in model transformations: often it is assumed that during development only the source model of a transformation undergoes modifications, however in practice it is necessary for developers to

B. Malloy, S. Staab, and M. van den Brand (Eds.): SLE 2010, LNCS 6563, pp. 183–202, 2011.

modify both the source and the target models of a transformation and propagate changes in both directions [4,5]. There are two main approaches for realizing bidirectional transformations: by programming forward and backward transformations in any convenient unidirectional language and manually ensuring they are consistent; or by using a bidirectional transformation language where every program describes both a forward and a backward transformation simultaneously. A major advantage of the latter approach is that the consistency of the transformations can be guaranteed by construction. Moreover, source and target roles are not fixed since the transformation direction entails them. Therefore, considerations made about the mapping executed in one direction are completely equivalent to the opposite one.

The relevance of bidirectionality in model transformations has been acknowledged already in 2005 by the Object Management Group (OMG) by including a bidirectional language in their Query View Transformation (QVT) [6]. Unfortunately, as pointed out by Perdita Stevens in [7] the language definition is affected by several weaknesses. Therefore, while MDE requirements demand enough expressiveness to write non-bijective transformations [8], the QVT standard does not clarify how to deal with corresponding issues, leaving their resolution to tool implementations. Moreover, a number of approaches and languages have been proposed due to the intrinsic complexity of bidirectionality. Each one of those languages is characterized by a set of specific properties pertaining to a particular applicative domain [9].

This paper presents the Janus Transformation Language (JTL), a declarative model transformation language specifically tailored to support bidirectionality and change propagation. In particular, the distinctive characteristics of JTL are

- *non-bijectivity*, non-bijective bidirectional transformations are capable of mapping a model into a set of models, as for instance when a single change in a target model might semantically correspond to a family of related changes in more than one source model. JTL provides support to non-bijectivity and its semantics assures that all the models are computed at once independently whether they represent the outcome of the backward or forward execution of the bidirectional transformation;
- *model approximation*, generally transformations are not total which means that target models can be manually modified in such a way they are not reachable anymore by any forward transformation, then traceability information are employed to back propagate the changes from the modified targets by inferring the *closest* model that approximates the ideal source one at best.

The language expressiveness and applicability have been validated by implementing the *Collapse/Expand State Diagrams* benchmark which have been defined in [10] to compare and assess different bidirectional approaches. The JTL semantics is defined in terms of the Answer Set Programming (ASP) [11], a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. Bidirectional transformations are translated via semantic anchoring [12] into search problems which are reduced to computing stable models, and the DLV solver [13] is used to perform search.

The structure of the paper is as follows: Section 2 sets the context of the paper through a motivating example that is used throughout the paper to demonstrate the

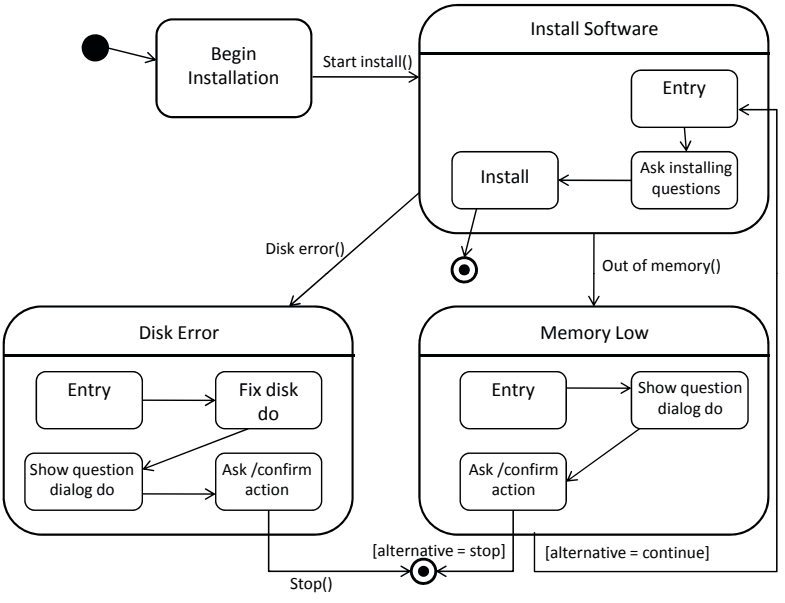
approach and Section 3 discusses requirements a bidirectional and change propagating language should support. Section 4 describes conceptual and implementation aspects of the proposed approach and Section 5 applies the approach to a case study. Section 6 relates the work presented in this paper with other approaches. Finally, Section 7 draws the conclusions and presents future work.

2 Motivating Scenario

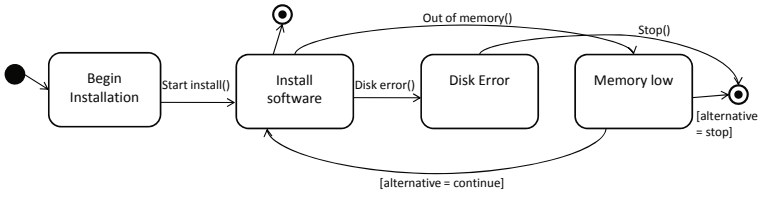
As aforesaid, bidirectionality in model transformations raises not obvious issues mainly related to non-bijectivity [7,14]. More precisely, let us consider the *Collapse/Expand State Diagrams* benchmark defined in the *GRACE International Meeting on Bidirectional Transformations* [10]: starting from a hierarchical state diagram (involving some one-level nesting) as the one reported in Figure 1.a, a flat view has to be provided as in Figure 1.b. Furthermore, any manual modifications on the (target) flat view should be back propagated and eventually reflected in the (source) hierarchical view. For instance, let us suppose the designer modifies the flat view by changing the name of the initial state from `Begin Installation` to `Start Install shield` (see Δ_1 change in Figure 2). Then, in order to persist such a refinement to new executions of the transformation, the hierarchical state machine has to be consistently updated by modifying its initial state as illustrated in Figure 3.

The flattening is a non-injective operation requiring specific support to back propagate modifications operated on the flattened state machine to the nested one. For instance, the flattened view reported in Figure 1 can be extended by adding the alternative `try again` from the state `Disk Error` to `Install software` (see Δ_2 changes in Figure 2). This gives place to an interesting situation: the new transition can be equally mapped to each one of the nested states within `Install Software` as well as to the container state itself. Consequently, more than one source model propagating the changes exists[1]. Intuitively, each time hierarchies are flattened there is a loss of information which causes ambiguities when trying to map back corresponding target revisions. Some of these problems can be alleviated by managing traceability information of the transformation executions which can be exploited later on to trace back the changes: like this each generated element can be linked with the corresponding source and contribute to the resolution of some of the ambiguities. Nonetheless, traceability is a necessary but not sufficient condition to support bidirectionality, since for instance elements discarded by the mapping may not appear in the traces, as well as new elements added on the target side. For instance, the generated flattened view in Figure 1.b can be additionally manipulated through the Δ_3 revisions which consist of adding some extra-functional information for the `Install Software` state and the transition between from `Memory low` and `Install Software` states. Because of the limited expressive power of the hierarchical state machine metamodel which does not support extra-functional annotations, the Δ_3 revisions do not have counterparts in the state machine in Figure 3.

[1] It is worth noting that the case study and examples have been kept deliberately simple since they suffice to show the relevant issues related to non-bijectivity



a) A sample Hierarchical State Machine (HSM).



b) The corresponding Non-Hierarchical State Machine (NHSM).

Fig. 1. Sample models for the *Collapse/Expand State Diagrams* benchmark

Current declarative bidirectional languages, such as QVT relations (QVT-R), are often ambivalent when discussing non-bijective transformations as already pointed out [7]; whilst other approaches, notably hybrid or graph-based transformation techniques, even if claiming the support of bidirectionality, are able to deal only with (partially) bijective mappings [4]. As a consequence, there is not a clear understanding of what non-bijectivity implies causing language implementors to adopt design decisions which differ from an implementation to another.

In order to better understand how the different languages deal with non-bijectivity, we have specified the hierarchical to non-hierarchical state machines transformation (HSM2NHSM) by means of the Medini[2] and MOFLON[3] systems. The former is an

[2] `http://projects.ikv.de/qvt/`
[3] `http://www.moflon.org`

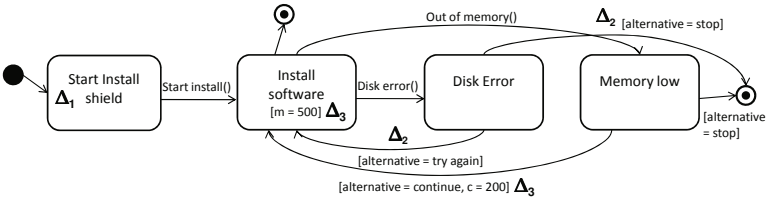


Fig. 2. A revision of the generated non-hierarchical state machine

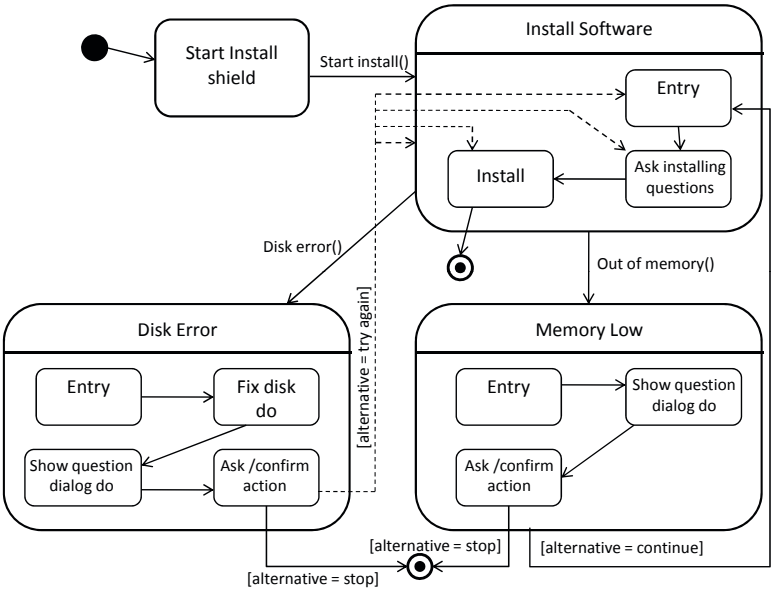


Fig. 3. The source hierarchical state machine synchronised with the target changes

implementation of the QVT-R transformation language, whereas the latter is a framework which bases on Triple Graph Grammars (TGGs) [15]: our experience with them is outlined in the following

Medini. When trying to map the generated target model back to the source without any modification, a new source model is generated which differs from the original one[4]. In particular, incoming (outgoing) transitions to (from) nested states are flattened to the corresponding parent: when going back such mapping makes the involved nested states to disappear (as `Entry` and `Install` in the `Install Software` composite in Figure 1). Moreover, the same mapping induces the creation of extra composite states for existing simple states, like `Begin Installation` and the initial and final states of the hierarchical state machine. Starting from this status, we made the modifications

[4] In this paper the details about the experiments done with Medini and TGGs can not be described in detail due to space restrictions. The interested reader can access the full implementation of both the attempts at http://www.mrtc.mdh.se/∼acicchetti/HSM2NHSM.php

on the target model as prescribed by Figure 2 and re-applied the transformation in the source direction, i.e. backward. In this case, the `Start Install shield` state is correctly mapped back by renaming the existing `Begin Installation` in the source. In the same way, the modified transition from `Disk Error` to the final state is consistently updated. However, the newly added transition outgoing from `Disk Error` to `Install software` is mapped by default to the composite state, which might not be the preferred option for the user. Finally, the manipulation of the attributes related to memory requirements and cost are not mapped back to any source element but are preserved when new executions of the transformation in the target direction are triggered.

MOFLON. The TGGs implementation offered by MOFLON is capable of generating Java programs starting from diagrammatic specifications of graph transformations. The generated code realizes two separate unidirectional transformations which as in other bidirectional languages should be consistent by construction. However, while the forward transformation implementation can be considered complete with respect to the transformation specification, the backward program restricts the change propagation to attribute updates and element deletions. In other words, the backward propagation is restricted to the contexts where the transformation can exploit trace information.

In the next sections, we firstly motivate a set of requirements a bidirectional transformation language should meet to fully achieve its potential; then, we introduce the JTL language, its support to non-bijective bidirectional transformations, and its ASP-based semantics.

3 Requirements for Bidirectionality and Change Propagation

This section refines the definition of bidirectional model transformations as proposed in [7] by explicitly considering non-bijective cases. Even if some of the existing bidirectional approaches enable the definition of non-bijective mappings [7,5], their validity is guaranteed only on bijective sub-portions of the problem. As a consequence, the forward transformation can be supposed to be an injective function, and the backward transformation its corresponding inverse; unfortunately, such requirement excludes most of the cases [16]. In general, a bidirectional transformation *R* between two classes of models, say *M* and *N*, and *M* more expressive than *N*, is characterized by two unidirectional transformations

$$\overrightarrow{R} : M \times N \rightarrow N$$
$$\overleftarrow{R} : M \times N \rightarrow M^*$$

where $\overrightarrow{R}$ takes a pair of models *(m, n)* and works out how to modify *n* so as to enforce the relation $\overrightarrow{R}$. In a similar way, $\overleftarrow{R}$ propagates changes in the opposite direction: $\overleftarrow{R}$ is a non-bijective function able to map the target model in a set of corresponding source models conforming to M[5]. Furthermore, since transformations are not total in

[5] For the sake of readability, we consider a non-bijective backward transformation assuming that only M contains elements not represented in N. However, the reasoning is completely analogous for the forward transformation and can be done by exchanging the roles of M and N.

general, bidirectionality has to be provided even in the case the generated model has been manually modified in such a way it is not reachable anymore by the considered transformation. Traceability information is employed to back propagate the changes from the modified targets by inferring the *closest*[6] model that approximates the ideal source one at best. More formally the backward transformation $\overleftarrow{R}$ is a function such that:

(i) if *R(m,n)* is a non-bijective consistency relation, $\overleftarrow{R}$ generates all the resulting models according to R;
(ii) if *R(m,n)* is a non-total consistency relation, $\overleftarrow{R}$ is able to generate a result model which approximates the ideal one.

This definition alone does not constrain much on the behavior of the reverse transformation and additional requirements are necessary in order to ensure that the propagation of changes behaves as expected.

Reachability. In case a generated model has been manually modified (n'), the backward transformation $\overleftarrow{R}$ generates models (m^*) which are exact, meaning that the original target may be reached by each of them via the transformation without additional side effects. Formally:

$$\overleftarrow{R}(m, n') = m^* \in M^*$$

$$\overrightarrow{R}(m', n') = n' \in N \text{ for each } m' \in m^*$$

Choice preservation. Let n' be the target model generated from an arbitrary model m' in m^* as above: when the user selects m' as the appropriate source pertaining to n' the backward transformation has to generate exactly m' from n' disregarding the other possible alternatives $t \in m^*$ such that $t \neq m'$. In other words, a valid round-trip process has to be guaranteed even when multiple sources are available [14]:

$$\overleftarrow{R}(m', \overrightarrow{R}(m', n')) = m' \text{ for each } m' \in m^*$$

Clearly, the above requirement in order to be met demands for adequate traceability information management.

In the rest of the paper, the proposed language is introduced and shown to satisfy the above requirements. The details of the language and its supporting development environment are presented in Section 4, whereas in Section 5 the usage of the language is demonstrated by means of the benchmark case.

4 The Janus Transformation Language

The Janus Transformation Language (JTL) is a declarative model transformation language specifically tailored to support bidirectionality and change propagation. The

[6] This concept is clarified in Sect. 4, where the transformation engine and its derivation mechanism are discussed.

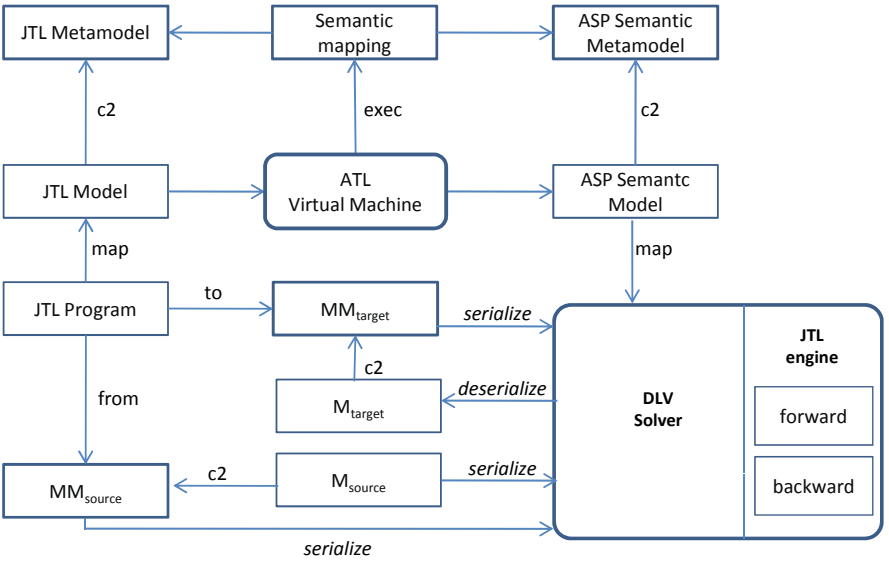


Fig. 4. Architecture overview of the JTL environment

implementation of the language relies on the Answer Set Programming (ASP) [11]. This is a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. Being more precise model transformations specified in JTL are transformed into ASP programs (search problems), then an ASP solver is executed to find all the possible stable models that are sets of atoms which are consistent with the rules of the considered program and supported by a deductive process.

The overall architecture of the environment supporting the execution of JTL transformations is reported in Figure 4. The *JTL engine* is written in the ASP language and makes use of the *DLV solver* [13] to execute transformations in both forward and backward directions. The engine executes JTL transformations which have been written in a QVT-like syntax, and then automatically transformed into ASP programs. Such a semantic anchoring has been implemented in terms of an ATL [17] transformation defined on the JTL and ASP metamodels. Also the source and target metamodels of the considered transformation (MM_{source}, MM_{target}) are automatically encoded in ASP and managed by the engine during the execution of the considered transformation and to generate the output models.

The overall architecture has been implemented as a set of plug-ins of the Eclipse framework and mainly exploits the Eclipse Modelling Framework (EMF) [18] and the ATLAS Model Management Architecture (AMMA) [19]. Moreover, the DLV solver has been wrapped and integrated in the overall environment.

In the rest of the section all the components of the architecture previously outlined are presented in detail. In particular, Section 4.1 presents the JTL engine, the syntax

```
1metanode(HSM, state).
2metanode(HSM, transition).
3metaprop(HSM, name, state).
4metaprop(HSM, trigger, transition).
5metaprop(HSM, effect, transition).
6metaedge(HSM, association, source, transition, state).
7metaedge(HSM, association, target, transition, state).
8[...]
```

Listing 1.1. Fragment of the State Machine metamodel

of transformation language is described in Section 4.2 by using a running example, whereas the semantic anchoring is described in Section 4.3.

4.1 The Janus Transformation Engine

As previously said the Janus transformation engine is based on a relational and declarative approach implemented using the ASP language to specify bidirectional transformations. The approach exploits the benefits of logic programming that enables the specification of relations between source and target types by means of predicates, and intrinsically supports bidirectionality [9] in terms of unification-based matching, searching, and backtracking facilities.

Starting from the encoding of the involved metamodels and the source model (see the *serialize* arrows in the Figure 4), the representation of the target one is generated according to the JTL specification (as shown in Section 4.2). The computational process is performed by the JTL engine (as depicted in Figure 4) which is based on an ASP bidirectional transformation program executed by means of an ASP solver called DLV [13].

Encoding of models and metamodels. In the proposed approach, models and metamodels are defined in a declarative manner by means of a set of logic assertions. In particular, they are considered as graphs composed of nodes, edges and properties that qualify them. The metamodel encoding is based on a set of *terms* each characterized by the predicate symbols `metanode`, `metaedge`, and `metaprop`, respectively. A fragment of the hierarchical state machine metamodel considered in Section 2 is encoded in Listing 1.1. For instance, the `metanode(HSM,state)` in line `1` encodes the metaclass `state` belonging to the metamodel `HSM`. The `metaprop(HSM,name,state)` in line `3` encodes the attribute named `name` of the metaclass `state` belonging to the metamodel `HSM`. Finally, the `metaedge(HSM,association,source,transition,state)` in line `6` encodes the association between the metaclasses `transition` and `state`, typed `association`, named `source` and belonging to the metamodel `HSM`. The terms induced by a certain metamodel are exploited for encoding models conforming to it. In particular, models are sets of entities (represented through the predicate symbol `node`), each characterized by properties (specified by means of `prop`) and related together by relations (represented by `edge`). For instance, the state machine model in Figure 1

```
1 node(HSM, "s1", state).
2 node(HSM, "s2", state).
3 node(HSM, "t1", transition).
4 prop(HSM,"s1.1","s1",name,"begin␣installation").
5 prop(HSM,"s2.1","s2",name,"install␣software").
6 prop(HSM,"t1.1","t1",trigger,"install␣software").
7 prop(HSM,"t1.2","t1",effect,"start␣install").
8 edge(HSM,"tr1",association,source, "s1","t1").
9 edge(HSM,"tr1",association,target, "s2","t1").
10 [...]
```

Listing 1.2. Fragment of the State Machine model in Figure 1

is encoded in the Listing 1.2. In particular, the `node(HSM,"s1",state)` in line `1` encodes the instance identified with `"s1"` of the class `state` belonging to the metamodel `HSM`. The `prop(HSM,"s1",name,"start")` in line `4` encodes the attribute `name` of the class `"s1"` with value `"start"` belonging to the metamodel `HSM`. Finally, the `edge(HSM,"tr1",association,source,"s1","t1")` in line `7` encodes the instance `"tr1"` of the association between the state `"s1"` and the transition `"t1"` belonging to the metamodel `HSM`.

Model transformation execution. After the encoding phase, the deduction of the target model is performed according to the rules defined in the ASP program. The transformation engine is composed of *i) relations* which describe correspondences among element types of the source and target metamodels, *ii) constraints* which specify restrictions on the given relations that must be satisfied in order to execute the corresponding mappings, and an *iii) execution engine* (described in the rest of the section) consisting of bidirectional rules implementing the specified relations as executable mappings. Relations and constraints are obtained from the given JTL specification, whereas the execution engine is always the same and represents the bidirectional engine able to interpret the correspondences among elements and execute the transformation.

The transformation process logically consists of the following steps:

(i) given the input (meta)models, the execution engine induces all the possible solution candidates according to the specified relations;
(ii) the set of candidates is refined by means of constraints.

The Listing 1.3 contains a fragment of the ASP code implementing relations and constraints of the HSM2NHSM transformation discussed in Section 2. In particular, the terms in lines `1-2` define the relation called `"r1"` between the metaclass `State machine` belonging to the `HSM` metamodel and the metaclass `State machine` belonging to the `NHSM` metamodel. An ASP constraint expresses an invalid condition: for example, the constraints in line `3-4` impose that each time a state machine occurs in the source model it has to be generated also in the target model. In fact, if each atoms in its body is true then the correspondent solution candidate is eliminated. In similar way, the relation between the metaclasses `State` of the involved metamodels is encoded in line `6-7`. In this case, constraints in line `8-11` impose that each time a state occurs in the HSM model, the correspondent one in the NHSM model is generated only if the source element is not a sub-state, vice versa, each state in the NHSM model is

```
1 relation ("r1",  HSM, stateMachine).
2 relation ("r1", NHSM, stateMachine).
3 :- node(HSM, "sm1", stateMachine), not  node'(HSM, "sm1", stateMachine).
4 :- node(NHSM, "sm1", stateMachine), not  node'(NHSM, "sm1", stateMachine).
5
6 relation ("r2", HSM, state).
7 relation ("r2", NHSM, state).
8 :- node(HSM, "s1", state), not edge(HSM, "ow1", owningCompositeState, "s1", "cs1
      "), not node'(NHSM, "s1", state).
9 :- node(HSM, "s1", state), edge(HSM, "ow1", owningCompositeState, "s1", "cs1"),
      node(HSM, "cs1", compositeState), node'(NHSM, "s1", state).
10 :- node(NHSM, "s1", state), not trace_node(HSM, "s1", compositeState), not node
      '(HSM, "s1", state).
11 :- node(NHSM, "s1", state), trace_node(HSM, "s1", compositeState), node'(HSM, "
      s1", state).
12
13 relation ("r3", HSM, compositeState).
14 relation ("r3", NHSM, state).
15 :- node(HSM, "s1", compositeState), not node'(NHSM, "s1", state).
16 :- node(NHSM, "s1", state), trace_node(HSM, "s1", compositeState), not node'(HSM
      , "s1", compositeState).
17 [...]
```

Listing 1.3. Fragment of the HSM2NHSM transformation

mapped in the HSM model. Finally, the relation between the metaclasses `Composite state` and `State` is encoded in line `13-14`. Constraints in line `15-16` impose that each time a composite state occurs in the HSM model a correspondent state in the NHSM model is generated, and vice versa. Missing sub-states in a NHSM model can be generated again in the HSM model by means of trace information (see line `10-11` and `16`). Trace elements are automatically generated each time a model element is discarded by the mapping and need to be stored in order to be regenerated during the backward transformation.

Note that the specification order of the relations is not relevant as their execution is bottom-up; i.e., the final answer set is always deduced starting from the more nested facts.

Execution engine. The specified transformations are executed by a generic engine which is (partially) reported in Listing 1.4. The main goal of the transformation execution is the generation of target elements as the *node'* elements in line 11 of Listing 1.4. As previously said transformation rules may produce more than one target models, which are all the possible combinations of elements that the program is able to create. In particular, by referring to Listing 1.4 target node elements with the form `node'(MM,ID,MC)` are created if the following conditions are satisfied:

- the considered element is declared in the input source model. The lines `1-2` contain the rules for the source conformance checking related to `node` terms. In particular, the term `is_source_metamodel_conform(MM,ID,MC)` is true if the terms `node(MM,ID,MC)` and `metanode(MM,MC)` exist. Therefore, the term `bad_source` is true if the corresponding `is_source_metamodel_con- form(MM,ID,MC)` is valued to false with respect to the `node(MM,ID,MC)` source element.

```
1 is_source_metamodel_conform(MM,ID,MC) :- node(MM,ID,MC), metanode(MM,MC).
2 bad_source :- node(MM,ID,MC), not is_source_metamodel_conform(MM,ID,MC).
3 mapping(MM,ID,MC) :- relation(R,MM,MC), relation(R,MM2,MC2), node(MM2,ID,MC2), MM
      !=MM2.
4 is_target_metamodel_conform(MM,MC) :- metanode(MM,MC).
5 {is_generable(MM,ID,MC)} :- not bad_source, mapping(MM,ID,MC),
      is_target_metamodel_conform(MM,MC), MM=mmt.
6 node'(MM,ID,MC) :- is_generable(MM,ID,MC), mapping(MM,ID,MC), MM=mmt.
```

Listing 1.4. Fragment of the *Execution engine*

- at least a relation exists between a source element and the candidate target element. In particular, the term `mapping(MM,ID,MC)` in line `3` is true if exists a relation which involves elements referring to `MC` and `MC2` metaclasses and an element `node(MM2,ID,MC2)`. In other words, a mapping can be executed each time it is specified between a source and a target, and exists the appropriate source to compute the target.
- the candidate target element conforms to the target metamodel. In particular, the term `is_target_metamodel_conform(MM,MC)` in line `6` is true if the `MC` metaclass exists in the `MM` metamodel (i.e. the target metamodel).
- finally, any constraint defined in the *relations* in Listing 1.3 is valued to false.

The invertibility of transformations is obtained by means of trace information that connects source and target elements; in this way, during the transformation process, the relationships between models that are created by the transformation executions can be stored to preserve mapping information in a permanent way. Furthermore, all the source elements lost during the forward transformation execution (for example, due to the different expressive power of the metamodels) are stored in order to be generated again in the backward transformation execution.

4.2 Specifying Model Transformation with Janus

Due to the reduced usability of the ASP language, we have decided to provide support for specifying transformations by means of a more usable syntax inspired by QVT-R. In Listing 1.5 we report a fragment of the HSM2NHSM transformation specified in JTL and it transforms hierarchical state machines into flat state machines and the other way round. The forward transformation is clearly non-injective as many different hierarchical machines can be flattened to the same model and consequently transforming back a modified flat machine can give place to more than one hierarchical machine. Such a transformation consists of several relations like *StateMachine2StateMachine*, *State2State* and *CompositeState2State* which are specified in Listing 1.5. They define correspondences between *a)* state machines in the two different metamodels *b)* atomic states in the two different metamodels and *c)* composite states in hierarchical machines and atomic states in flat machines. The relation in lines 11-20 is constrained by means of the *when* clause such that only atomic states are considered. Similarly to QVT, the *checkonly* and *enforce* constructs are also provided: the former is used to check if the

domain where it is applied exists in the considered model; the latter induces the modifications of those models which do not contain the domain specified as *enforce*. A JTL relation is considered bidirectional when both the contained domains are specified with the construct *enforce*.

```
1 transformation hsm2nhsm(source : HSM, target : NHSM) {
2
3  top relation StateMachine2StateMachine {
4
5     enforce domain source sSM : HSM::StateMachine;
6     enforce domain target tSM : NHSM::StateMachine;
7
8   }
9
10 top relation State2State {
11
12    enforce domain source sourceState : HSM::State;
13    enforce domain target targetState : NHSM::State;
14
15    when {
16      sourceState.owningCompositeState.oclIsUndefined();
17    }
18
19  }
20
21 top relation CompositeState2State {
22
23   enforce domain source sourceState : HSM::CompositeState;
24   enforce domain target targetState : NHSM::State;
25
26 }
27}
```

Listing 1.5. A non-injective JTL program

The JTL transformations specified in the QVT-like syntax are mapped to the correspondent ASP program by means of a semantic anchoring operation as described in the next section.

4.3 ASP Semantic Anchoring

According to the proposed approach, the designer task is limited to specifying relational model transformations in JTL syntax and to applying them on models and metamodels defined as EMF entities within the Eclipse framework.

Designers can take advantage of ASP and of the transformation properties discussed in the previous sections in a transparent manner since only the JTL syntax is used. In fact, ASP programs are automatically obtained from JTL specifications by means of ATL transformations as depicted in the upper part of Figure 4. Such a transformation is able to generate ASP predicates for each relation specified with JTL. For instance, the relation *State2State* in Listing 1.5 gives place to the *relation* predicates in lines 6-7 in Listing 1.3.

The JTL *when* clause is also managed and it induces the generation of further ASP constraints. For instance, the JTL clause in line 16 of Listing 1.5 gives place to a couple of ASP constraints defined on the *owningCompositeState* feature of the state machine metamodels (see lines 8-9 in Listing 1.3). Such constraints are able to filter the states and consider only those which are not nested.

To support the backward application of the specified transformation, for each JTL relation additional ASP constraints are generated in order to support the management of trace links. For instance, the *State2State* relation in Listing 1.5 induces the generation of the constraints in lines 10-11 of Listing 1.3 to deal with the non-bijectivity of the transformation. In particular, when the transformation is backward applied on a *State* element of the target model, trace links are considered to check if such a state has been previously generated from a source *CompositeState* or *State* element. If such trace information is missing all the possible alternatives are generated.

5 JTL in Practice

In this section we show the application of the proposed approach to the *Collapse/Expand State Diagrams* case study presented in Section 2. The objective is to illustrate the use of JTL in practice by exploiting the developed environment, and in particular to show how the approach is able to propagate changes dealing with non-bijective and non-total scenarios. The following sections present how after the definition of models and metamodels (see Section 5.1), the JTL transformation may be specified and applied over them (see Section 5.2). Finally, the approach is also applied to manage changes occurring on the target models which need to be propagated to the source ones (see Section 5.3).

5.1 Modelling State Machines

According to the scenario described in Section 2, we assume that in the software development lifecycle, the designer is interested to have a behavioral description of the system by means of hierarchical state machine, whereas a test expert produces non-hierarchical state machine models. The hierarchical and non-hierarchical state machine matamodels (respectively HSM and NHSM) are given by means of their Ecore representation within the EMF framework. Then a hierarchical state machine model conforming to the HSM metamodel can be specified as the model reported in the left-hand side of Figure 5. Models can be specified with graphical and/or concrete syntaxes depending on the tool availability for the considered modeling language. In our case, the adopted syntaxes for specifying models do not affect the overall transformation approach since models are manipulated by considering their abstract syntaxes.

5.2 Specifying and Applying the HSM2NHSM Model Transformation

Starting from the definition of the involved metamodels, the JTL transformation is specified according to the QVT-like syntax described in Section 4.2 (see Listing 1.5). By referring to the Figure 4, the *JTL program*, the *source* and *target metamodel*s and the *source model* have been created and need to be translated in their ASP encoding in order to be executed from the transformation engine. The corresponding ASP encodings are automatically produced by the mechanism illustrated in Section 4. In particular, the ASP encoding of both source model and source and target metamodels is generated according to the Listing 1.2 and 1.1, while the JTL program is translated to the corresponding ASP program (see Listing 1.3).

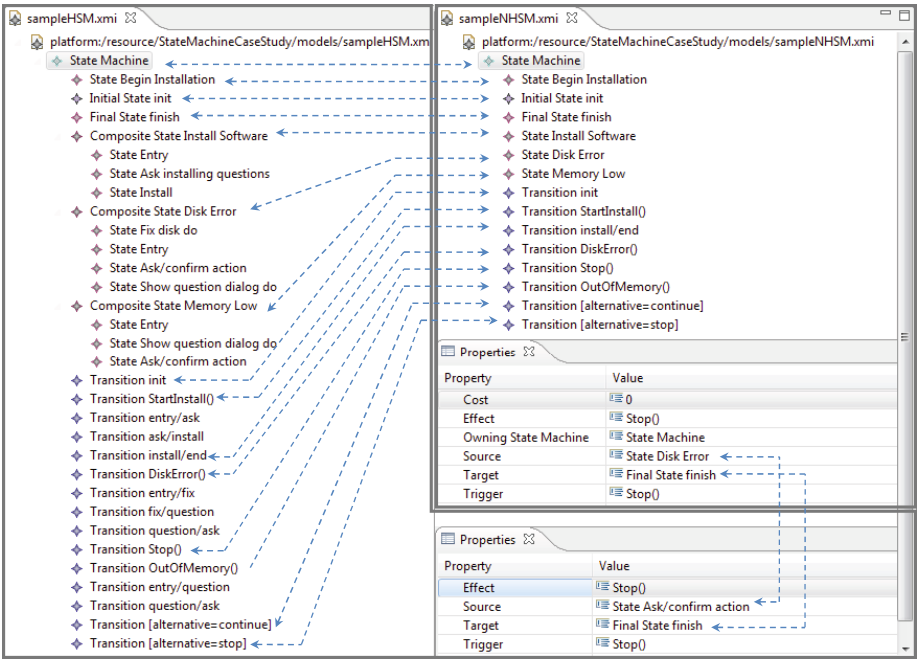


Fig. 5. HSM source model and the correspondent NHSM target model

After this phase, the application of the HSM2NHSM transformation on *sampleHSM* generates the corresponding *sampleNHSM* model as depicted in the right part of Figure 4. Note that, by re-applying the transformation in the backward direction it is possible to obtain again the *sampleHSM* source model. The missing sub-states and the transitions involving them are restored by means of trace information.

5.3 Propagating Changes

Suppose that in a refinement step the designer needs to manually modify the generated target by the changes described in Section 2 (see Δ changes depicted in Figure 2), that is:

1. renaming the initial state from `Begin Installation` to `Start Install shield`;
2. adding the alternative `try again` to the state `Disk Error` to come back to `Install software`;
3. changing the attributes related to memory requirements (`m=500`) in the state `Install software` and cost (`c=200`) of the transition from `Memory low` to `Install software`.

The target model including such changes (*sampleNHSM'*) is shown in the left part of the Figure 6. If the transformation HSM2NHSM is applied on it, we expect changes to be propagated on the source model. However, due to the different expressive power of the

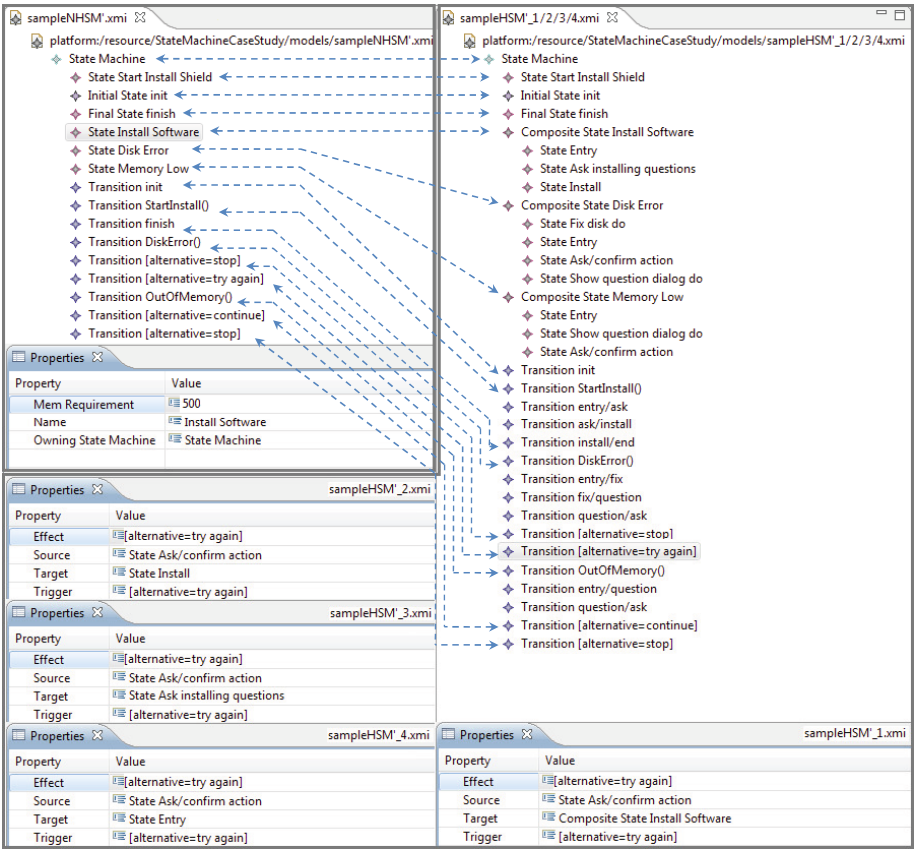


Fig. 6. The modified NHSM target model and the correspondent HSM source models

involved metamodels, target changes may be propagated in a number of different ways, thus making the application of the reverse transformation to propose more solutions. The generated sources, namely *sampleHSM'_1/2/3/4* can be inspected through Figure 6: the change (1) has been propagated renaming the state to `Start Install shield`; the change (2) gives place to a non-bijective mapping and for this reason more than one model is generated. As previously said, the new transition can be equally targeted to each one of the nested states within `Install Software` as well as to the super state itself (see the properties *sampleHSM'_1/2/3/4* in Figure 6). For example, as visible in the property of the transition, *sampleHSM'_1* represents the case in which the transition is targeted to the composite state `Install Software`; finally, the change (3) is out of the domain of the transformation. In this case, the new values for memory and cost are not propagated on the generated source models.

Even in this case, if the transformation is applied on one of the derived *sampleHSM'* models, the appropriate *sampleNHSM'* models including all the changes are generated. However, this time the target will preserve information about the chosen *sampleHSM'*

source model, thus causing future applications of the backward transformation to generate only *sampleHSM'*.

With regard to the performances of our approach, we performed no formal study on its complexity yet, since that goes beyond the scope of this work; however, our observations showed that the time required to execute each transformation in the illustrated case study is more than acceptable since it always took less than one second. In the general case, when there are a lot of target alternative models the overall performance of the approach may degrade.

6 Related Work

Model transformation is intrinsically difficult, and its nature poses a number of obstacles in providing adequate support for bidirectionality and change propagation [14]. As a consequence, despite a number of proposals, in general they impose relevant restrictions of the characteristics of the involved transformations. For instance, approaches like [20,21,22,23] require the mappings to be total, while [20,21,5] impose the existence of some kind of bijection between the involved source and target. Such comparison is discussed in [14].

Stevens [7] discusses bidirectional transformations focusing on basic properties which such transformations should satisfy. In particular, (i) *correctness* ensures a bidirectional transformation does something useful according to its consistency relation, (ii) *hippocraticness* prevents a transformation from does something harmful if nether model is modified by users, (iii) finally, *undoability* is about the ability whether a performed transformation can be canceled. The paper considers QVT-R as applied to the specification of bidirectional transformation and analyze requirements and open issues. Furthermore, it points out some ambiguity about whether the language is supposed to be able to specify and support non-bijective transformations.

Formal works on bidirectional transformations are based on graph grammars, especially triple graph grammars (TGGs) [15]. These approach interpret the models as graphs and the transformation is executed by using graph rewriting techniques. It is possible to specify non-bijective transformations; however, attributes are not modeled as part of the graphs.

In [5] an attempt is proposed to automate model synchronization from model transformations. It is based on QVT Relations and supports concurrent modifications on both source and target models; moreover, it propagates both sides changes in a non destructive manner. However, some issues come to light: in fact, conflicts may arise when merging models obtained by the propagation with the ones updated by the user. Moreover, it is not possible to manage manipulations that makes the models to go outside the domain of the transformation.

The formal definition of a round-trip engineering process taking into account the non-totality and non-injectivity of model transformations is presented in [14]. The valid modifications on target models are limited to the ones which do not induce backward mappings out the source metamodel and are not operated outside the transformation domain. The proposal discussed in this paper is capable also to manage target changes inducing extensions of the source metamodel by approximating the exact source as a

set of models; i.e., the set of possible models which are the closest to the ideal one from which to generate the previously modified model.

In [16] the author illustrates a technique to implement a change propagating transformation language called PMT. This work supports the preservation of target changes by back propagating them toward the source. On the one hand, conflicts may arise each time the generated target should be merged with the existing one; on the other hand, the back propagation poses some problems related to the invertibility of transformations, respectively.

Declarative approaches to model transformations offer several benefits like for example implicit source model traversal, automatic traceability management, implicit target object creation, and implicit rule ordering [9,24]. A number of interesting applications is available, varying from incremental techniques [25] to the automation of transformation specifications by means of the inductive construction of first-order clausal theories from examples and background knowledge [26]. One of the closest works to our approach is xMOF [27]. It aims to provide bidirectionality and incremental transformations by means of an OCL constraint solving system which enables the specification of model transformations. However, transformation developers have no automation support to derive constraints from source/target metamodel conformance and transformation rules, which may make their task hard in case of complex mappings [16]. Moreover, it has not been clarified how such technique would deal with multiple choices and the requirements described in Sect. 3.

We already introduced in [28] an ASP based transformation engine enabling the support for partial and non injective mappings. However, the inverse transformation has to be given by the developer and a valid round-trip process is not guaranteed, as already discussed throughout the paper. For this purpose, we introduced JTL, a transformation language specifically designed for supporting change propagation with model approximation capabilities.

7 Conclusion and Future Work

Bidirectional model transformations represent at the same time an intrinsically difficult problem and a crucial mechanism for keeping consistent and synchronized a number of related models. In this paper, we have refined an existing definition of bidirectional model transformations (see [7]) in order to better accommodate non-bijectivity and model approximation. In fact, existing languages fail in many respect when dealing with non-bijectivity as in many cases its semantic implications are only partially explored, as for instance in bidirectional QVT transformations whose standard does not even clarify whether valid transformations are only bijective transformations. Naturally, non-bijective transformations can possibly map a number of source models to the same target model, therefore whenever a target model is manually modified, the changes must be back propagated to the related source models.

This paper presented the Janus Transformation Language (JTL), a declarative model transformation approach tailored to support bidirectionality and change propagation which conforms to the requirements presented in Section 3. JTL is able to map a model into a set of semantically related models in both forward and backward directions,

moreover whenever modifications to a target model are making it unreachable from the transformation an approximation of the ideal source model is inferred. To the best of our knowledge these characteristics are unique and we are not aware of any other language which deals with non-bijectivity and model approximation in a similar way. The expressivity and applicability of the approach has been validated against a relevant benchmark, i.e., the transformation among hierachical and non-hierarchical state machines as prescribed by [10]. The language has been given abstract and concrete syntax and its semantics is defined in terms of Answer Set Programming; a tool is available which renders the language interoperable with EMF [7].

As future work we plan to extend the framework with a wizard helping the architect to make decisions among proposed design alternatives. The alternatives are initially partitioned, constrained, abstracted, and graphically visualized to the user. Then, when decisions are made, they are stored and used to drive subsequent decisions. Another interesting future work is to investigate about incremental bidirectional model transformations. If a developer changes one model, the effects of these changes should be propagated to other models without re-executing the entire model transformation from scratch. In the context of bidirectional transformation it should coexist with the ability to propagate changes in both the directions but preserves information in the models and, in our case, also allows the approximation of models.

References

1. Schmidt, D.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (2006)
2. Bézivin, J.: On the Unification Power of Models. Jour. on Software and Systems Modeling (SoSyM) 4(2), 171–188 (2005)
3. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software 20(5), 42–45 (2003)
4. Stevens, P.: A Landscape of Bidirectional Model Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
5. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 213–228. Springer, Heidelberg (2009)
6. Object Management Group (OMG): MOF QVT Final Adopted Specification, OMG Adopted Specification ptc/05-11-01 (2005)
7. Stevens, P.: Bidirectional model transformations in qvt: semantic issues and open questions. Software and Systems Modeling 8 (2009)
8. Steven Witkop: MDA users' requirements for QVT transformations. OMG document 05-02-04 (2005)
9. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems J. 45(3) (June 2006)
10. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)
11. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Kowalski, R.A., Bowen, K. (eds.) Proceedings of the Fifth Int. Conf. on Logic Programming, pp. 1070–1080. The MIT Press, Cambridge (1988)

[7] `http://www.di.univaq.it/romina.eramo/JTL`

12. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
13. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning (2004)
14. Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)
15. Konigs, A., Schurr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electronic Notes in Theoretical Computer Science 148, 113–150 (2006)
16. Tratt, L.: A change propagating model transformation language. Journal of Object Technology 7(3), 107–126 (2008)
17. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
18. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison-Wesley, Reading (2003)
19. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In: Aßmann, U., Liu, Y., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005)
20. Van Paesschen, E., Kantarcioglu, M., D'Hondt, M.: SelfSync: A Dynamic Round-Trip Engineering Environment. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 633–647. Springer, Heidelberg (2005)
21. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
22. Mu, S.-C., Hu, Z., Takeichi, M.: An Injective Language for Reversible Computation. In: Kozen, D., Shankland, C. (eds.) MPC 2004. LNCS, vol. 3125, pp. 289–313. Springer, Heidelberg (2004)
23. Foster, J., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
24. Mens, T., Gorp, P.V.: A Taxonomy of Model Transformation. Electr. Notes Theor. Comput. Sci. 152, 125–142 (2006)
25. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
26. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Software and Systems Modeling (2009)
27. Compuware and Sun: XMOF queries, views and transformations on models using MOF, OCL and patterns, OMG Document ad/2003-08-07 (2003)
28. Cicchetti, A., Di Ruscio, D., Eramo, R.: Towards Propagation of Changes by Model Approximations. In: 10th IEEE Int. Enterprise Distributed Object Computing Conf. Workshops (EDOCW 2006), vol. 0, p. 24 (2006)