Enhancing the JTL Tool for Bidirectional Transformations

Romina Eramo
University of L'Aquila
L'Aquila, Italy
romina.eramo@univaq.it

Alfonso Pierantonio
University of L'Aquila
L'Aquila, Italy
alfonso.pierantonio@univaq.it

Michele Tucci
University of L'Aquila
L'Aquila, Italy
michele.tucci@univaq.it

ABSTRACT

In Model-Driven Engineering, the potential advantages of using bidirectional transformations in various scenarios are largely recognized; as for instance, assuring the overall consistency of a set of interrelated models which requires the capability of propagating changes *back* and *forth* the transformation chain.

Among the existing approaches, JTL (Janus Transformation Language) is a constraint-based bidirectional transformation language specifically tailored to support change propagation and non-deterministic transformations. In fact, its relational and constraint-based semantics allows to restore consistency by returning all admissible models. This paper introduces the new implementation of the language and presents the tools and its features by means of a running example.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**;

KEYWORDS

MDE, Bidirectional model transformation, JTL

1 INTRODUCTION

In Model Driven Engineering (MDE), bidirectional model transformations are considered a core ingredient for managing both the consistency and synchronization of two or more related models. Their relevance has been advocated by a number of approaches that 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 [4, 9].

An aspect which is still largely underestimated is the multiplicity of solutions: if the forward mapping of a bidirectional transformation is non-bijective, i.e., it maps two distinct models to the same target model, then the corresponding backward mapping must be a *one-to-many* mapping (*non-determinism*) [6]. Consequently, since current languages are able to generate only one model, non-deterministic transformations involved in round-tripping can give rise to results, which are somewhat unpredictable. In these cases, the solution is normally identified according to heuristics or to the order the rules are written [16, 17]. A typical example is the *Collapse/Expand State Diagrams* benchmark defined in [3]: starting from a hierarchical state diagram (involving some one-level nesting), a flat view has to be provided. Then adding a transition between two flatten states cannot be back-propagated in a unique way, since the corresponding transition in the hierarchical state diagram can be added to any nested states as well as to the container state itself.

The Janus Transformation Language (JTL) [2, 6] is a constraint-based model transformation language specifically tailored to support bidirectionality and change propagation. Its relational semantics relies on a constraint solver to find a consistent choice for the other source; there might be multiple choices. Thus, the responsibility of choosing the right model among the generated ones is left to the designer.

The development of JTL started in 2010 with major goal including the specification of bidirectional transformations within the engine based on Answer Set Programming (ASP) [7]. Its prototype version was presented in [2] as a set of independent Eclipse plugins for academic use. Recently, there has been a lot of progress in the Eclipse Modeling Framework (EMF) [1] and it has established itself as a de facto standard, offering stable and well-tested components. We are convinced to reengineering the existing features and integrate them in a dedicated Eclipse product. In particular, the Eclipse Rich Client Platform (RCP) [2] has been used as a basis to create a feature-rich stand-alone application. Going in more details, the following improvements have been realized: the semantic anchoring between the JTL syntax and the ASP engine is completely restructured using ATL [10] and EMFText [3] technologies; the existing transformation and traceability engines have been modified to solve shortcomings that emerged from test cases; the DLV solver system [13] has been updated and integrated in the overall environment to provide interoperability with EMF; moreover, usability, performance and multi-platform support have been also improved in the process. This paper presents the new tool for JTL [4] by using a running example.

The paper is organized as follows. Sect. 2 presents how bidirectional transformations can be specified within the JTL tool. Sect. 3 describes the JTL engine and its semantics. Sect. 4 show the tool in practice by executing an example. Finally, Sect. 5 describes related work and Sect. 6 draws some conclusion and future work.

2 JTL BIDIRECTIONAL TRANSFORMATIONS

Within the JTL tool, models and metamodels can be described by exploiting the EMF environment. Whereas, bidirectional transformations can be specified in a textual way by using the JTL syntax.

By considering the *Collapse/Expand State Diagrams* benchmark [3], List. 1 shows a fragment of the HSM2SM bidirectional transformation, which relates hierarchical and flat state machines one with another. JTL adopts a QVT-R like syntax to specify a number of *relations* among elements of the two involved *domains*. In particular, the Line 1 of the listing declares the variable `hsm` that matches models conform to the metamodel `HSM` and the variable `sm`

[1] EMF: https://www.eclipse.org/modeling/emf/
[2] Eclipse RCP: https://wiki.eclipse.org/Rich_Client_Platform
[3] EMFText: http://www.emftext.org
[4] JTL Tool: http://jtl.di.univaq.it/

that matches models conform to the metamodel SM. The relation StateMachine2StateMachine in Lines 2-13 relates state machines in the different metamodels. The *when* and *where* clauses specify pre- and post-conditions on the relation. In particular, the where clause in Lines 7-12 invokes a set of relations that are held if the calling relation is enforced. For instance, the relation ownedState2ownedState, declared in Lines 14-24, defines a correspondence between the reference ownedState of type State belongs to the hierarchical domain and the reference ownedState of type State belongs to the flat domain. Note that, the states have the same variable name (s) and the relation State2State(s,s) is invoked in the where clause. It means that if a state machine have a reference to a state s in the source domain, a correspondent reference to a state s must be created in the target domain; the correspondent state s is created by enforcing the relation State2State. Similarly, the relation CompositeState2State relates composite states of the hierarchical domain to state in the flat domain, and vice-versa. Note that, the two relations State2State and CompositeState2State make the transformation *non-injective*. In fact, in the backward direction an object of type State may be equally mapped to objects of type State or CompositeState.

```
1  transformation HSM2SM(hsm : HSM, sm : SM) {
2    top relation StateMachine2StateMachine {
3      enforce domain hsm hm : HSM::StateMachine {  };
4      enforce domain sm m : SM::StateMachine {  };
5      where {
6        ownedState2ownedState(hm,m);
7        ownedCompositeState2ownedState(hm,m);
8        ownedTransition2ownedTransition(hm,m);
9        ...
10     }
11   }
12   relation ownedState2ownedState {
13     enforce domain hsm hm : HSM::StateMachine {
14       ownedState = s : HSM::State { }
15     };
16     enforce domain sm m : SM::StateMachine {
17       ownedState = s : SM::State { }
18     };
19     where {
20       State2State(s,s);
21     }
22   }
23   relation State2State {
24     varName : String;
25     enforce domain hsm hs : HSM::State {
26       name = varName
27     };
28     enforce domain sm s : SM::State {
29       name = varName
30     };
31   }
32   relation ownedCompositeState2ownedState {
33     ...
34   }
35   relation CompositeState2State {
36     varName : String;
37     enforce domain hsm hs : HSM::CompositeState {
38       name = varName
39     };
40     enforce domain sm s : SM::State {
41       name = varName
42     };
43   }
44   ...
45   relation ownedTransition2ownedTransition {
46     ...
47   }
48   relation Transition2Transition {
49     varTrigger : String;
50     varEffect : String;
51     enforce domain hsm ht : HSM::Transition {
52       trigger = varTrigger,
53       effect = varEffect
54     };
55     enforce domain sm t : SM::Transition {
56       trigger = varTrigger,
57       effect = varEffect
58     };
59     where {
60       TransitionSource2TransitionSource(ht, t);
61       TransitionTarget2TransitionTarget(ht, t);
62       TransitionSourceComposite2TransitionSource(ht, t);
63       TransitionTargetComposite2TransitionTarget(ht, t);
64       TransitionTargetIntoComposite2TransitionTargetComposite(
           ht, t);
65     }
66   }
67   relation TransitionSource2TransitionSource {
68     enforce domain hsm ht : HSM::Transition {
69       source = s : HSM::State { }
70     };
71     enforce domain sm t : SM::Transition {
72       source = s : SM::State { }
73     };
74     when {
75       State2State(s, s);
76     }
77   }
78   relation TransitionTarget2TransitionTarget {
79     enforce domain hsm ht : HSM::Transition {
80       target = hs : HSM::State { }
81     };
82     enforce domain sm t : SM::Transition {
83       target = s : SM::State { }
84     };
85     when { State2State(hs, s); }
86   }
87   relation TransitionSourceComposite2TransitionSource {
88     ...
89   }
90   relation TransitionTargetComposite2TransitionTarget {
91     ...
92   }
93   relation TransitionTargetIntoComposite2TransitionTarget
94             Composite {
95     enforce domain hsm ht : HSM::Transition {
96       target = hs : HSM::State {
97         owningCompositeState = s:HSM::CompositeState { }
98       }
99     };
100    enforce domain sm t: SM::Transition {
101      target = s : SM::State { }
102    };
103    when { CompositeState2State(s,s); }
104  }
105  ...
106 }
```

Listing 1: A fragment of the HSM2SM transformation

Thereafter, the Transition2Transition in Lines 51-69 relates transitions in the two different metamodels. The where clause invokes a list of relations that have the scope to set the correspondent source and target elements of the transitions. For instance, the relation TransitionSource2TransitionSource involves transitions of the two metamodels and relates their references source of type State. The relation is constrained by means of the when clause; it implies that only states that have been generated from the relation State2State are considered (it allows to exclude sub-states). Finally, the relation TransitionTargetIntoComposite2TransitionTargetComposite manages the case in which the target of a transition refers to a sub-state and must be mapped in a correspondent transition that targets the

state that correspond to the composite state that the sub-state belongs to. Even in this case, the relations involving transitions may cause multiple solutions when the transformation is executed in backward direction; in fact, a transition that involves a state can be equally mapped in a transition that involves a state or a composite state.

The described relations are bidirectional, in fact both the contained domains are specified with the construct *enforce*.

3 JTL ENGINE AND SEMANTICS

The JTL engine is based on a relational and declarative approach implemented using ASP, that 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. 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 [4] in terms of unification-based matching, searching, and backtracking facilities. More precisely, 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 consistent with the considered rules and supported by a deductive process.

Fig. 1 depicts the overall environment supporting the execution of JTL transformations. The *JTL engine* is written in the ASP language and makes use of the *DLV solver* 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 (see the *map* arrows in Fig. 1). Such a semantic anchoring has been implemented in terms of an ATL transformation defined on the JTL and ASP metamodels. Moreover, the source and target metamodels of the considered transformation (MM_1, MM_2) are automatically encoded in ASP and managed by the engine during the execution of the considered transformation and to generate the output models.

Starting from the encoding of the involved metamodels and the source model M_1 [5], the representation of the target one (M_2) is generated according to the JTL specification (see the *serialize* and *deserialize* arrows). The execution of the backward direction is performed by giving as input the obtained M_2 and optionally the trace model T_{M2}. Thus, the correspondent M_1 is re-generated together with the related trace model T_{M1}. Note that, trace models are generated during the execution and each one refers to a specific execution and is related to a specific solution model. That is, in case of multiple solutions, multiple trace models (one for model) are generated.

Model transformation execution. Going in more details, the computational process is performed by the JTL engine (as depicted in Fig. 1) which is based on the ASP program obtained from the given JTL specification. It is composed of *i) rules* which describe mappings and correspondences among element types of the source and target metamodels, and *ii) constraints* which specify restrictions on the given relations that must be satisfied in order to execute the corresponding mappings. The transformation process logically consists of the following steps:
1) given the input metamodels and the input model (that is the *stable model*), the execution engine induces all the possible solution candidates (*answer sets*) according to the specified rules; *2)* the set of candidates is refined by means of constraints.

The invertibility of transformations is obtained by means of trace information that connects source and target elements. Traceability can be considered as an intrinsic property of ASP, in fact each obtained answer set is composed of elements that are explicitly derived from elements of the initial stable model. In essence, tracing information describes how a target element has been generated starting from a source one.

Furthermore, elements involved in non-bijective relationships (that are source of non-determinism) can be maintained by means of tracing information. For instance, if we consider the backward execution of the HSM2SM transformation and the relations `State2State` and `CompositeState2State`, thus a state in the *sm* domain can be translated and linked in both a state or a composite state in the *hsm* domain. Exploiting this, during the transformation process, the relationships between models that are created by the transformation executions can be stored to permanently preserve mapping information.

Finally, all the source elements lost during the forward transformation execution (for example, due to the different expressive power of the metamodels) are stored in tracing information in order to be generated again in the backward transformation execution.

Constraining the solution space. The number of alternatives that we may obtain depends on the intrinsic characteristics of the transformation and on the model elements which are matched by the non-bijective rules. In other words, the number of alternatives depends on the degree of non-determinism of the involved model transformations. As said above, the execution engine may generate a large number of alternatives when only the information encoded in the transformation is used. The constraints play a key role in the transformation process. In particular, transformations can be mapped into logical rules which are constrained by context information which consistently narrow the solution space to only those models which are relevant.

The answer sets may be refined in subsequent steps: *i)* the answer set is filtered according to the constraints induced by the source metamodel, *ii)* the answer set is further reduced by considering the constraints induced by the tracing information and *iii)* additional user-defined constraints can be added to browse the space of solutions.

The use of constraints may reduce back-tracking because it allows for early detection of dead-ends and permits to reduce the space of solutions in subsequent steps. In practice, constraints can be written a posteriori and given as input of the transformation engine; the designer can decide to use such constraints for a specific execution or to integrate them with the transformation specification itself to avoid discontinuity.

Managing the solution space. Even if adding constraints can help to discard alternatives, non-determinism may generally cause a

[5] Note that, since JTL provides declarative specifications, the role of source and target models is determined at execution time

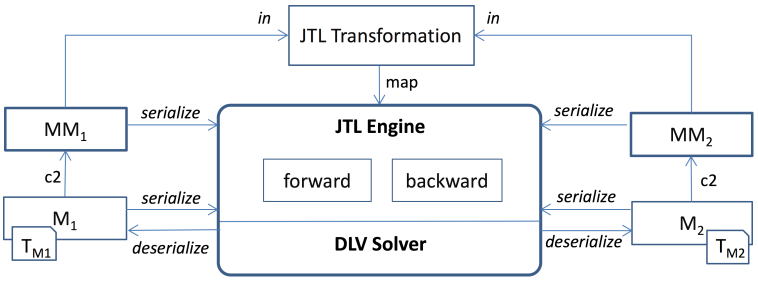

Figure 1: Overview of the JTL engine

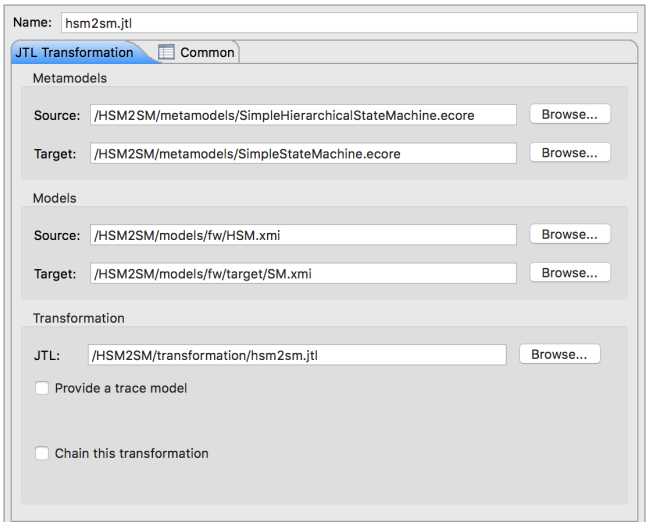

Figure 2: Run configuration (forward execution)

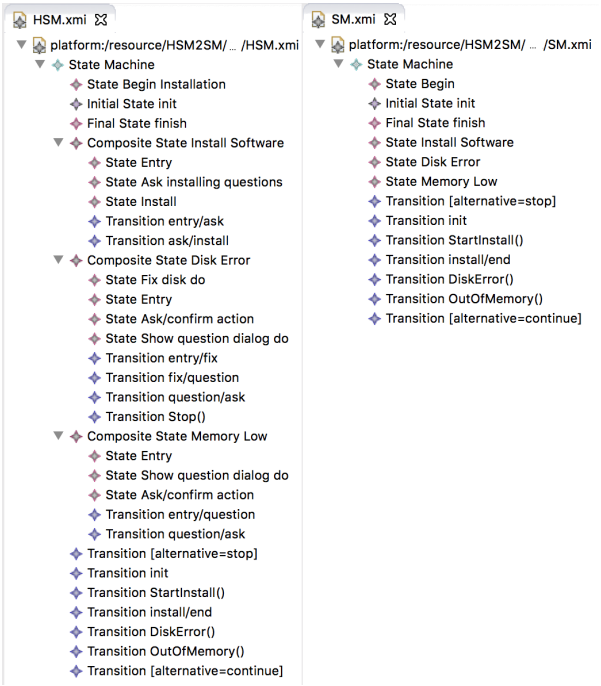

Figure 3: Forward execution of HSM2SM

combinatorial explosion of solutions. In order to reduce the burden of managing a collection of models, we adopted an approach to represent multiple solutions in a intensional manner by adopting a model for uncertainty [6]. Furthermore, in [5] we proposed a design-time approach to analyze JTL transformations with the purpose to detect ambiguities and support designers in solving non-determinism in their specification.

4 EXECUTION AND RESULTS

In this section we show the execution of the HSM2SM transformation within the JTL tool [6]. We considered the scenario presented in [2]; starting from the definition of the involved metamodels, the JTL transformation is specified as described in List. 1). By referring to Fig. 1, the transformation, the source and target metamodels and the source model have been created and need to be translated in their ASP encoding in order to be executed from the JTL engine.

After this phase, the application of the HSM2SM transformation (the run configuration window is showed in Fig. 2) on the source model `HSM.xmi` generates the corresponding target model `SM.xmi`, as depicted in Fig. 3. Togheter with the `.xmi` model, the engine generates the correspondent ASP encoding `SM.aspm` and the trace

[6]The HSM2SM execution is described as a tutorial at http://jtl.di.univaq.it/.

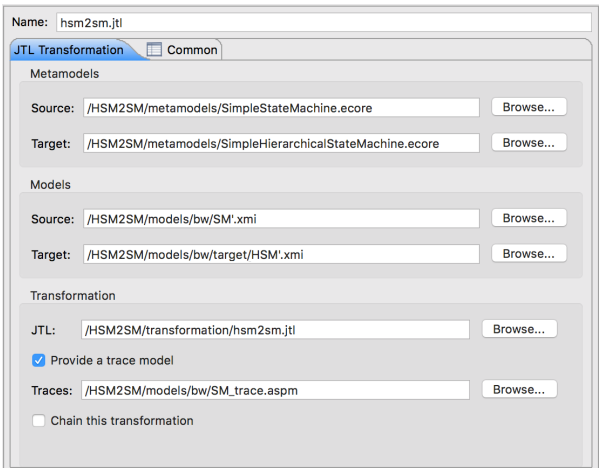

Figure 4: Run configuration (backward execution)

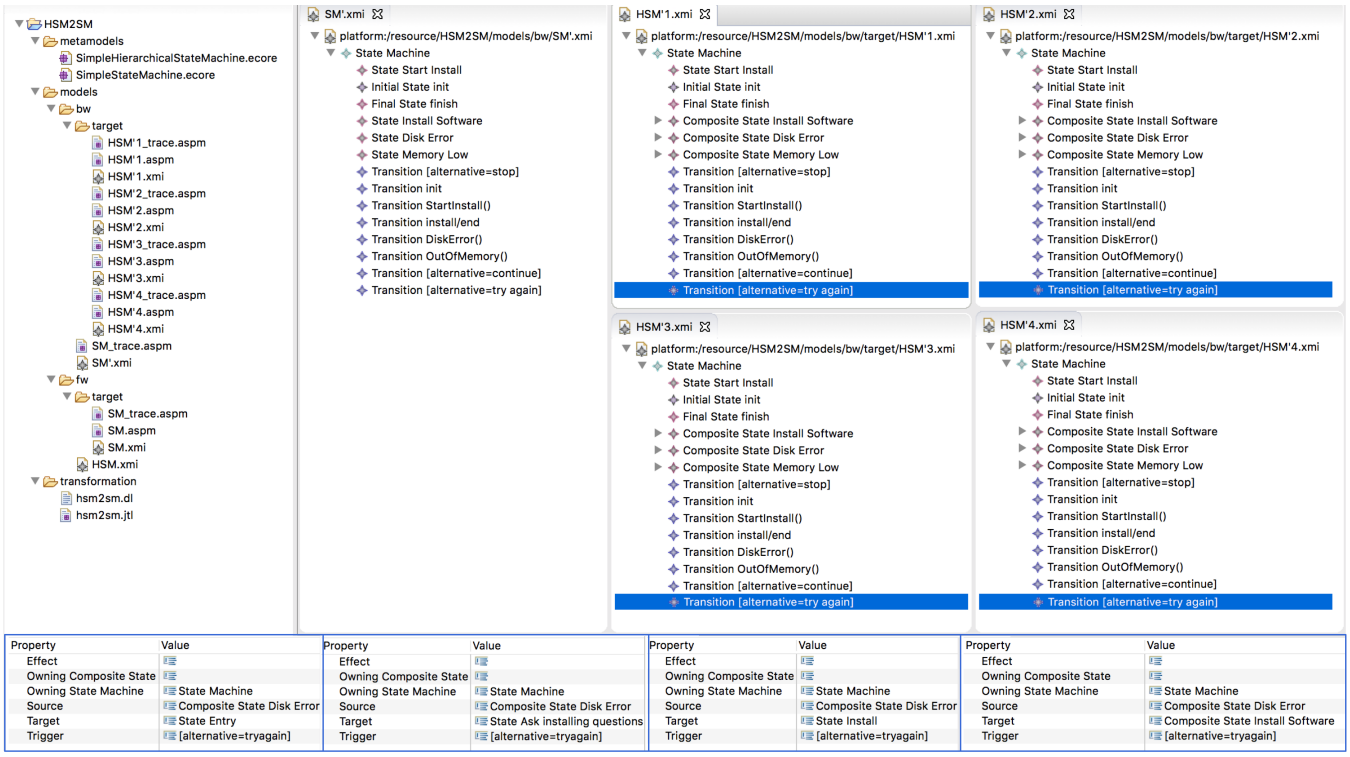

Figure 5: Backward execution of the HSM2SM transformation in JTL

model `SM_trace.aspm`. The latter is composed of a set of elements encoded in ASP with the role to maintain relations between source and target elements of the execution and store also elements lost during the transformation or deriving from non-injective mappings. In fact, by re-applying the transformation in the backward direction it is possible to obtain again the `HSM` source model. The missing sub-states and the transitions involving them are restored by means of trace information.

Suppose that, in a refinement step, the obtained target model is manually modified and an updated model `SM'.xmi` is obtained as shown in the left part of Fig. 5. In particular, the state `Begin Installation` is renamed in `Start Install` and a new transition `[alternative = try again]` between the state `Disk Error` and the state `Install Software` is added.

If the transformation HSM2SM is applied on it (the run configuration window is showed in Fig. 4), we expect changes to be propagated on the source model. However, as said, target changes may be propagated in a number of different ways, thus making the application of the reverse transformation to propose more solutions. As previously said, the new transition can be equally targeted to each one of the 3 nested states within `Install Software` as well as to the super state itself and also equally sourced to each one of the 4 nested states within `Disk Error` as well as to the super state itself. The resulting alternative models are 20.

For sake of legibility, four of the generated sources, namely `HSM'_1/2/3/4`, can be inspected through Fig. 5: the change (1) has been propagated renaming the state to `Start Install`; the change (2) has been propagated by creating the new transition, that has been targeted to each one of the nested states within `Install Software` as well as to the super state itself (see the properties `HSM'_1/2/3/4` in Fig. 5). For example, as visible in the property of the transition, `HSM'_1` represents the case in which the transition is targeted to the composite state `Install Software`. This restriction of the solution space was possible through the addition of an ASP constraint, a posteriori. The pseudocode of the constraint that eliminate the transitions that sources sub-states is as following:

$$:-sm.Transition(T1).source(t1).State(S1),\ hsm.Transition(T1).source(t1).State(S2),\ hsm.State(S2).owningCompositeState.CompositeState(S1).$$

It eliminates all the alternative models such that: a transition $T1$ that targets the state $S1$ exists in the *sm* domain, and also a transition $T1$ that targets the state $S2$ (that is a sub-state of $S1$) exists in the *hsm* domain.

Even in this case, if the transformation is applied on one of the derived `HSM'` models, the appropriate `SM'` models including all the changes are generated. However, this time the target will preserve information about the chosen `HSM'` source model, thus causing future applications of the backward transformation to generate only `HSM'`.

5 RELATED WORK

Over the last decade, a number of bidirectional approaches and tools have been proposed. Concerning the multiplicity of solutions,

most of the existing languages are deterministic, i.e., they produce one model at time.

The QVT-R bidirectional transformation language [16] does not supports non-bijective transformations. In [1] a formal discussion about non-deterministic transformations is given. Despite the heavy emphasis placed on model transformation by the OMG's, tool support for bidirectional transformations expressed in QVT-R remains limited [14, 15]. Medini QVT [7] is an Eclipse plugin for a subset of the QVT-R language. Its semantics admittedly disregards the semantics from the QVT standard (it does not have a checkonly mode for instance).

Within the available tools for Triple Graph Grammars (TGGs), eMoflon [8] received wide acceptance and it is one of the most prominent TGG platforms. It features both batch and incremental model transformations [12]. Moreover, eMoflon proposes to manage non-determinism by allowing designers to make decisions as early. In particular, it provides a Java-based API via which the designer can choose which matches she prefers/prioritises as soon as there are multiple choices available in the transformation process.

Fully control and therefore completely avoid non-determinism is recently proposed by the following functional approaches. An attempt in making bidirectional transformation deterministic by means of intentional updates is represented by the BiFluX language [17], however the problem that a transformation cannot be tested for non-determinism at static-time reduces its effectiveness. Recently, BiGUL [11] has been proposed as a revision of the core of BiFlux, designed to be closer to practical programming languages. GRoundTram (Graph Roundtrip Transformation for Models) [8] is an integrated framework for developing well-behaved bidirectional model transformations based on the functional programming language OCaml. It is a compositional, functional and algebraic approach based on graph algebra and structural recursion.

Similarly to JTL, in [14] the authors proposes un approach able to enumerate the possible solutions of a non-deterministic specification. The proposed QVT-R tool is based on a SAT solver (Alloy). In contrast with JTL, it supports metamodels enriched with OCL constraints to limit the possible edits and control non-determinism; moreover, the enforcement semantics is based on the principle of least change. Tool support is available, but development seems to be discontinued.

6 CONCLUSION AND FUTURE WORK

This paper introduced the improved EMF-based tool for specifying and execute bidirectional model transformation in JTL. The tool has been presented by means of a non-deterministic round-trip scenario. As future work we plan to consolidate and extend the tool with new features. In particular, we are interested in defining a user-friendly syntax to allow designers to specifies constraints. Then, we plan to implement EMF-based facility for store and manage traceability information. Furthermore, we plan to address the issues related to the management of a multitude of models by integrate within the tool the following components: (i) representing models as a model with uncertainty capability and (ii) static bidirectional specification analyzer to detect non-determinism at design-time.

[7] Medini QVT: http://projects.ikv.de/qvt
[8] eMoflon: http://www.moflon.org

REFERENCES

[1] F. Abou-Saleh, J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. 2015. Notions of Bidirectional Computation and Entangled State Monads. *MPC* (2015), 187–214.

[2] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. 2010. JTL: a bidirectional and change propagating transformation language. In *SLE10*. 183–202.

[3] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. [n. d.]. Bidirectional Transformations: A Cross-Discipline Perspective - GRACE meeting. In *Procs. of ICMT2009*.

[4] K. Czarnecki and S. Helsen. 2006. Feature-based Survey of Model Transformation Approaches. *IBM Systems J.* 45, 3 (June 2006).

[5] Romina Eramo, Romeo Marinelli, Alfonso Pierantonio, and Gianni Rosa. 2014. Towards Analyzing Non-Determinism in Bidirectional Transformations. In *Procs. of AMT 2014*.

[6] R. Eramo, A. Pierantonio, and G. Rosa. 2015. Managing uncertainty in bidirectional model transformations. In *Proceedings of the Int. Conference on Software Language Engineering, SLE 2015*. 49–58.

[7] M. Gelfond and V. Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Procs of ICLP*. 1070–1080.

[8] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. 2011. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In *ASE 2011*.

[9] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. 2015. Feature-based classification of bidirectional transformation approaches. *SOSYM*, 1–22.

[10] F. Jouault and I. Kurtev. 2005. Transforming Models with ATL. In *MoDELS Satellite Events*, Vol. 3844. 128–138.

[11] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: A Formally Verified Core Language for Putback-based Bidirectional Programming. In *Procs of PEPM '16*. 61–72.

[12] M. Lauder, A. Anjorin, G. Varró, and A. Schürr. [n. d.]. Efficient Model Synchronization with Precedence Triple Graph Grammars. In *ICGT 2012*. 401–415.

[13] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. 2004. The DLV System for Knowledge Representation and Reasoning. *TOCL*.

[14] Nuno Macedo and Alcino Cunha. 2013. Implementing QVT-R Bidirectional Model Transformations Using Alloy.. In *FASE*. 297–311.

[15] Perdita Stevens. 2008. A Landscape of Bidirectional Model Transformations. In *GTTSE 2007 (LNCS 5235)*. 408–424.

[16] Perdita Stevens. 2009. Bidirectional model transformations in QVT: semantic issues and open questions. *SOSYM* 8 (2009).

[17] T. Zan, H. Pacheco, and Z. Hu. 2014. Writing bidirectional model transformations as intentional updates.. In *ICSE Companion*. 488–491.