Managing Uncertainty in Bidirectional Model Transformations

Romina Eramo
Università degli Studi dell'Aquila
I-67100 L'Aquila, Italy
romina.eramo@univaq.it

Alfonso Pierantonio
Università degli Studi dell'Aquila
I-67100 L'Aquila, Italy
alfonso.pierantonio@univaq.it

Gianni Rosa
Università degli Studi dell'Aquila
I-67100 L'Aquila, Italy
gianni.rosa@univaq.it

Abstract

In Model-Driven Engineering bidirectionality in transformations is regarded as a key mechanism. Recent approaches to non-deterministic transformations have been proposed for dealing with non-bijectivity. Among them, the JTL language is based on a relational model transformation engine which restores consistency by returning all admissible models. This can be regarded as an uncertainty reducing process: the unknown uncertainty at design-time is translated into known uncertainty at run-time by generating multiple choices. Unfortunately, little changes in a model usually correspond to a combinatorial explosion of the solution space. In this paper, we propose to represent the multiple solutions in a intensional manner by adopting a model for uncertainty. The technique is applied to JTL demonstrating the advantages of the proposal.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.3.0 [*Programming languages*]: General

Keywords Model-Driven Engineering, Bidirectional Model Transformations, Uncertainty

1. Introduction

In Model-Driven Engineering [22] (MDE) bidirectionality in transformations is regarded as a key mechanism. Its employment comprises mapping models to other models to focus on particular features of a system, simulate/validate a given application, and primarily keeping a set of interrelated models synchronized or in a consistent state. Most current bidirectional tools, such as like OMG's QVT Relations [19] (QVT-R) and Triple Graph Grammars [23] (TGGs), prescribe writing a bidirectional transformation as a declarative *consistency relation* between two metamodels. Since most interesting examples of bidirectional transformations are non-bijective, restoring consistency is an inherently ambiguous process [24, 25]. In fact, reversing non-injective mappings may give place to multiple choices, requiring that the transformation designer decide on a general consistency-restoration strategy at design-time [1]. However, when this is not possible, current tools typically consider only one particular strategy out of the many possible alternatives, of which developers have little or no control [25]. For instance, a process model can be transformed into an unordered set of activities. Then, adding a new activity to the set cannot be univocally back-propagated to the process model, as there are multiple ways to accommodate the activity in the process. A possible consistency-restoration strategy could be to append the activity to the end or beginning of the process. Omitting such a general rule renders the result almost unpredictable with current tools.

Recently, novel approaches to bidirectionality [3, 4, 18] have been proposed for dealing with *non-deterministic* transformations. In this kind of transformations

> *"... programmers need only specify a consistency relation, allowing the bx[1] engine to resolve the under-specification non-deterministically"* [1].

Among them, the Janus Transformation Language [4] (JTL) is a model transformation language specifically tailored to support bidirectionality and change propagation. Its relational semantics relies on Answer Set Programming [14] (ASP): given a change to one source, JTL uses the DLV [17] 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. In other words, the unknown uncertainty at design-time, which prevents the implementor from writing a deterministic transformation, is translated into known uncertainty in the outcome.

Unfortunately, little changes in a source usually correspond to a combinatorial explosion of alternatives in the other source, as shown in Sect. 2. This represents a major

[1] In this context, *bx* usually stays for bidirectional, bidirectionality, or bidirectional transformations.

shortcoming because manually inspecting models, which largely overlap each other, is prone to errors and omissions. In particular, it might be too difficult for the designer to detect, register, and keep track of the differences among the alternatives, whose number grows exponentially with the changes. In this paper, we propose to deal with the multitude of models generated by non-deterministic transformations by leveraging *uncertainty* to a first-class status. In particular, as in other approaches (e.g., [11]) by known uncertainty we mean *multiple possibilities*. To this end, we propose a metamodel-independent approach (based on preliminary work in [10]) that enables to denote a solution space by means of a model with uncertainty. The JTL engine is therefore revised to accommodate the new *intensional* semantics. This permits a transformation to natively generate a model with uncertainty instead of a myriad of models. An evaluation is provided where the approach proves to be efficient and scalable. In fact, it considerably reduces the impact of the combinatorial explosion of updates as demonstrated by the reduced number of generated model elements.

Structure of the paper. The paper is organized as follows. Next section illustrates the problem we intend to solve by means of an example. In Sect. 3 a characterization of the approach is given. Then, the metamodel-independent extension to uncertainty is presented. In Sect. 4, the technique for generating a model with uncertainty instead of the complete set of solutions is formalized. In the next section, an evaluation of the approach is given and finally Sect. 8 draws some conclusions and future work.

2. Little changes, big repercussion

Consider the *Collapse/Expand State Diagrams* round-trip benchmark [5]. The forward transformation $\overrightarrow{T}$ translates the hierarchical state machine in Fig. 1(a) into its flattened version in Fig. 1(b). The transformation T is non-bijective because different hierarchical machines can be translated into the same model.

Let us suppose now that the designer modifies the target model by means of the changes Δ, which are highlighted in bold and with thicker lines in Fig. 1(c). More in details, Δ consists of the following modifications:

- a new state `Printing` is added,
- a new transition `print` from `Active` to `Printing` is added,
- the transition `done` from `Active` to `Idle` is deleted and replaced by a new transition `completed`,
- a new transition `done` from `Printing` to `Idle` is added, and finally
- a new transition `critical_error` from `Out of services` to the initial state `Off` is added.

Because of these modifications, the original source model and the revised target model are not consistent any longer. Thus, the backward transformation $\overleftarrow{T}$ can be used to restore the consistency by propagating the changes in Δ. However, there is not a unique way of updating the source model. In fact, the added transitions in the target may be mapped to either of the nested states as well as to the container state itself, as illustrated in Fig. 1(d), where the dotted edges represent the alternative transitions. Despite the changes are relatively simple, their impact on the source model is relevant in terms of choices. In fact, the cardinality of the solution space is given by

$$|\texttt{print}| \times |\texttt{completed}| \times |\texttt{critical\_error}| = 4 \times 4 \times 3 = 48$$

where $|\texttt{name}|$ is the number of alternative model elements called `name`. It is worth noting that the models in Fig. 1(d) are represented by means of the dotted notation, which is informal: in this case a bidirectional transformation implemented in JTL would generate a collection of 48 distinguished models. A fragment of the above transformation is illustrated in Appendix A.

The lack of knowledge, which prevented the implementor from including a consistency-restoration strategy in the transformation, has been translated into known uncertainty, i.e., multiple possibilities. The scope of this paper is to represent the multitude of generated models into a model with uncertainty capable of encoding all the alternatives in one instance.

3. Overview of the Approach

A non-bijective bidirectional transformation is inherently ambiguous. Most bidirectional tools, e.g., QVT-R [19] and TGGs [23], require that the transformation designer or implementor specify a consistency-restoration strategy in advance. A formal characterization of such class of transformations have been provided by Stevens in [24]. However, in *non-deterministic* approaches [3, 4, 18] programmers need only specify a consistency relation, allowing the engine to resolve the underspecification non-deterministically [1]. This corresponds to generating all valid choices and letting the designer to identify the right one. In order to characterize also non-deterministic approaches, we generalize the notation proposed in [24] by adopting multivalued functions [16] to capture the multiplicity of the solution space

Definition. A *multivalued function* f from a set A to a set B is a function $f : A \to \mathcal{P}(B)$, with $\mathcal{P}(B)$ power set of B, such that $f(a)$ is non-empty for every $a \in A$.

Multivalued functions are left-total relations in which inputs are associated with multiple outputs. In order to distinguish them from singled-valued functions, we denote a multivalued function f from A to B with $f : A \Rightarrow B$.

If M and N are metamodels related by a consistency relation $R \subseteq M \times N$, then it holds for a pair of models

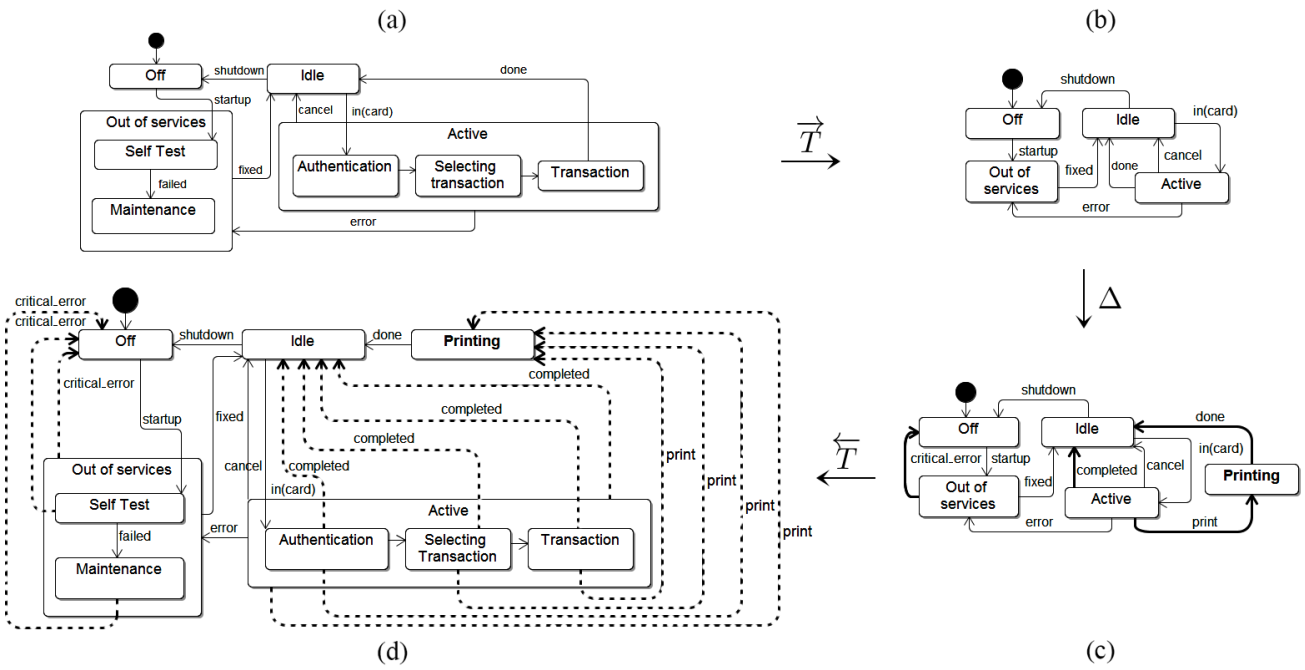

Figure 1. Collapse/expand state diagrams in a round-trip process

(m, n) whenever they are consistent[2]. The relation R can be realized by specifying the following *multivalued* directional mappings:

$$\overrightarrow{T_R} : M \times N \Rightarrow N \quad (1)$$

$$\overleftarrow{T_R} : M \times N \Rightarrow M \quad (2)$$

Analogously to the deterministic case, the idea is that $\overrightarrow{T_R}$ looks at a pair of models (m, n) and works out how to modify n so as to enforce the relation R: it returns the modified version, however n can be modified according to different strategies therefore $\overrightarrow{T_R}$ returns a set of models $\{n_1, \ldots, n_v\}$ all satisfying R. Similarly, $\overleftarrow{T_R}$ propagates changes in the opposite direction.

The following correctness definition postulates that $\overrightarrow{T_R}$ and $\overleftarrow{T_R}$ enforce the relation R

Definition. A transformation T_R is said to be *correct* if for any model $m \in M$ and $n \in N$

$$R(m, n') \quad \forall n' \in \overrightarrow{T_R}(m, n) = \{n_1 \ldots n_v\} \quad (3)$$

$$R(m', n) \quad \forall m' \in \overleftarrow{T_R}(m, n) = \{m_1 \ldots m_w\} \quad (4)$$

In other words, regardless of the transformation direction the transformation T_R is considered correct only if it generates consistent solutions.

Another relevant aspect in bidirectionality is hippocraticness. Applying a transformation to a source model in the presence of an existing target model will not in general be equivalent to applying it in the presence of an empty target model [24]. In other words, a transformation must not modify either of a pair of models if they are already in the specified relation.

Definition. A transformation T_R is said to be *hippocratic* is for any model $m \in M$ and $n \in N$

$$R(m, n) \implies \overrightarrow{T_R}(m, n) = \{n\} \quad (5)$$

$$R(m, n) \implies \overleftarrow{T_R}(m, n) = \{m\} \quad (6)$$

It is worth noting how (5-6) are the same as the unary case (see [24]) because ambiguity occurs only when consistency has to be restored.

The above discussion captures the fact that whenever the implementor is unable to find a general consistency restoring strategy because of unknown uncertainty, non-deterministic approaches resolve the underspecification by producing all valid choices. This can be regarded as an uncertainty reducing process: the unknown uncertainty at design-time is translated into known uncertainty at run-time by generating all candidate solutions. In order to identify the *right* solution additional information is required, which is typically detained and used by the designer while inspecting and validating the solution space. Regrettably, this knowledge cannot easily be leveraged to specify a general update strategy at design-time.

As illustrated in the previous section dealing with a myriad of models is difficult, if not impractical. What is needed is a construction capable of intensionally representing the set

[2] Although different methods can be given for defining a set of models, in this paper we assume that a metamodel corresponds to a set of models.

of models generated by a transformation. Hence, we distinguish among two different engine behaviors:

– *extensional*, the directional mappings $\overrightarrow{T_R}$ and $\overleftarrow{T_R}$ generate all the models satisfying the relation as described above;

– *intensional*, the directional mappings $\overrightarrow{T_R}$ and $\overleftarrow{T_R}$ generate a model with uncertainty which is semantically equivalent to the models of the extensional case, i.e., the corresponding set of models can be derived from it. For the sake of clarity, we call these mappings $\overrightarrow{\mathbb{T}_R}$ and $\overleftarrow{\mathbb{T}_R}$.

The models with uncertainty are obtained by means of an automated construction U presented in Sect. 4. In particular, given a metamodel N, the corresponding uncertainty metamodel $U(N)$ is automatically defined. A model with uncertainty $\bar{n}$ is used to intensionally represent all the alternative models solution of a non-deterministic transformation. Therefore, the mappings (1-2) can be reformulated as follows

$$\overrightarrow{\mathbb{T}_R} : M \times N \rightarrow U(N) \tag{7}$$

$$\overleftarrow{\mathbb{T}_R} : M \times N \rightarrow U(M) \tag{8}$$

As said, the new intensional mappings must be semantically equivalent to the extensional ones, i.e., the models encoded by the models with uncertainty generated by $\mathbb{T}_R$ must correspond to the models generated by T_R. However, as we will see in the next section, the models with uncertainty over-approximate the extensional mappings because of their combinatorial nature. More formally, let $[\bar{n}]$ be the set of instances encoded in $\bar{n}$, then

$$\overrightarrow{T_R}(m,n) \subseteq \left[\overrightarrow{\mathbb{T}_R}(m,n)\right] \tag{9}$$

for any $m \in M$ and $n \in N$. Similarly, this holds also for $\overleftarrow{T_R}$ and $\overleftarrow{\mathbb{T}_R}$.

In order to exclude unwanted models, the engine must contextually generate also a predicate p, which constraints the available combinations in order to exclude the non valid ones, i.e.

$$\overrightarrow{T_R}(m,n) = \left\{ t \in \left[\overrightarrow{\mathbb{T}_R}(m,n)\right] \mid t \models p \right\} \tag{10}$$

For instance, in Fig. 1(d) not all the `print` transitions are compatible with all the `completed` ones. Thus, not valid combinations must be eliminated by means of emphad-hoc predicates.

In the next section, we present a metamodel-independent approach to uncertainty. This is the starting point for extending a language like JTL and its transformation engine towards an uncertainty-aware solution capable of representing the solution space with a single model with uncertainty.

4. A metamodel-independent approach to uncertainty

Uncertainty is increasingly important in today's software-based systems. Rather than ignoring uncertainty, we should consider it as a first-class concern in the design, implementation, and deployment of those systems [13]. In modeling this corresponds to a set of possible models without being sure about the one we want [11]. In order to reduce the burden of managing a collection of models, we present a metamodel-independent approach to uncertainty representation based on preliminary work in [10].

The uncertainty metamodel $U(M)$ is obtained by extending a *base* metamodel M with specific connectives to represent the multiple outcomes of a transformation. These connectives denote the uncertainty points where alternative model elements are attached. Moreover, such points of uncertainty are traceable in order to ease the traversal of the solution space and to permit the identification of specific solution models. For instance, let us consider the metamodel *HSM* of the hierarchical state machines given in Fig. 2. The uncertainty metamodel $U(HSM)$ in Fig. 3 is automatically obtained by extending the base metamodel as follows:

1) the abstract metaclass `TracedClass` with attributes `trace` and `ref` is added,
2) for each metaclass *c* in *HSM*, such that it is non-abstract and does not specialize other metaclasses:
   2.1) a direct sub-metaclass *uc* of *c* is added,
   2.2) *c* is generalized by `TracedClass`,
3) each metaclass *uc* is composed with *c*, enabling the representation of a point of uncertainty and its alternatives, finally
4) the cardinality of attributes and references are relaxed and made optional in order to permit to express uncertainty also over them.

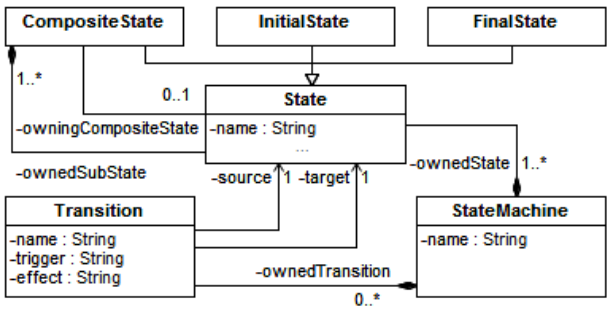

Figure 2. The *HSM* metamodel

In particular, the metaclasses `UStateMachine`, `UState` and `UTransition` in $U(HSM)$ are the uncertainty points for instances of `StateMachine`, `State` and `Transition`, which are in turn generalized by `TracedClass`. The purpose of

`TracedClass` is to maintain information about how the alternatives in the uncertainty points can be combined to obtain valid solution models. The discussion about the attribute `ref` in `TracedClass` is postponed in Sect. 5 where their utility will become evident.

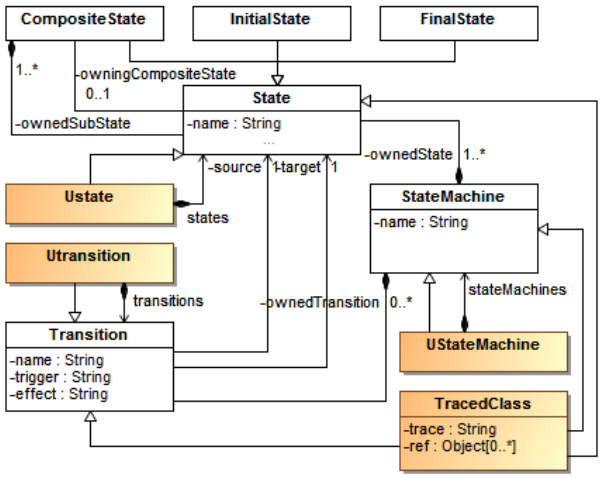

Figure 3. The $U(HSM)$ metamodel

The above metamodel enables the intensional representation of a set of state machines by mean of a model with uncertainty. For instance, consider the state machines in Fig. 1(d). The "intersection" of such models (all the nodes and part of the transitions) represents what is not subject to uncertainty. Whilst, the dotted alternative transitions denoted by `print`, `completed`, and `critical_error` corresponds to three different uncertainty points of type `UTransition`, each connected to the corresponding alternative `Transition` elements. The above construction is realized by means of a model transformation and applies to any Ecore[3] metamodel [10].

The main advantage of such representation technique is that the uncertainty represented by a set of alternative models is leveraged to a first-class status. Hence, a complete set of models can therefore be manipulated as whole, for instance with an automated transformation (as done for instance in [12]), without iterating over each individual in the set.

At this point, in order to let the JTL engine generate models with uncertainty, we need to extend the language according to the characterization given in Sect. 3.

5. The solution space as a model with uncertainty

In order to let JTL generate the model with uncertainty representing the solution space we have to identify *i)* the "intersection" shared among all choices and *ii)* the uncertain parts. Referring to Fig. 1(d), the latter consists of all elements in the model represented by the dotted edges. This is obtained by executing the non-bijective portion of the transformation, thus a set of uncertain elements in the source model are originated from the same target element. Since the JTL engine is a logic program written in ASP, it can derive how models are related by means of a deductive process. In this respect, the existing traceability management offers enough information to understand how the models can be factorized by identifying those elements which are connected among them. In particular, tracing information stores relevant details about the linkage between source and target model elements at execution-time (including the applied transformation rules). To better understand how JTL works, let us consider the following definitions

Definition. A *trace link* l associated to $\overrightarrow{T_R}(m,n) \in R$ consists of a tuple (a,b,r) with a and b model elements in $m \in M$ and $n \in N$, respectively, such that b is obtained from a by applying the rule r in T_R.

For the sake of brevity, a trace link $l = (a,b,r)$ is denoted by $r(a) = b$ whenever it does not give place to confusion. A similarly characterization holds for the opposite direction. We can give now the notion of tracing domain, as follows

Definition. Let $\overrightarrow{T_R}(m,n) \in R$, then its *tracing domain* D consists of all associated trace links $r(a) = b$, with $a \in m$, $b \in n$, and r in T_R.

Each element in the tracing domain represents the linkage, which describes how a target element has been generated starting from a source one. Similarly, the definition can be given also for $\overleftarrow{T_R}(m,n)$. Trace links are used to determinate the set of alternative elements that compose an uncertainty point, as follows

Definition. An *uncertainty point* u is a set of trace links, such that if $r_1(a_1) = b_1$ and $r_2(a_2) = b_2$ are two distinguished trace links in u, then $a_1 = a_2$, $b_1 \neq b_2$, and $r_1 \neq r_2$.

In essence, an uncertainty point is characterized by all *diverging* trace links originated by the same source element via different rules. The different target elements in the same uncertainty point represent the alternatives.

Let us to consider the scenario described in Sect. 2. The transition *print* in the modified machine in Fig. 1.(c) is transformed by the backward execution of the transformation, in a set of *print* in the hierarchical machine in Fig. 1.(d). The corresponding trace links (in their XMI representation) are given in the following fragment:

```
1 <tracelink name="l1">
2     <source><transitions name="print" source="Active"
          target="Printing" /></source>
3     <target><transitions name="print" source="Active"
          target="Printing" /></target>
4     <rule name="TransitionComposite2Transition" />
5 </tracelink>
6
7 <tracelink name="l2">
```

[3] http://www.eclipse.org/modeling/emf/

```
8     <source><transitions name="print" source="Active"
          target="Printing" /></source>
9     <target><transitions name="print" source="
          Authentication" target="Printing" /></target>
10    <rule name="Transition2Transition" />
11</tracelink>
12
13<tracelink name="l3">
14    <source><transitions name="print" source="Active"
          target="Printing" /></source>
15    <target><transitions name="print" source="Selecting␣
          Transaction" target="Printing" /></target>
16    <rule name="Transition2Transition" />
17</tracelink>
18
19<tracelink name="l4">
20    <source><transitions name="print" source="Active"
          target="Printing" /></source>
21    <target><transitions name="print" source="Transaction
          " target="Printing" />/target>
22    <rule name="Transition2Transition" />
23</tracelink>
```

Each `tracelink` is composed of a `source` transition, a `target` transition, and the transformation `rule`. For instance, the trace link `l1` documents how the source transition `print` from `Active` to `Printing` is transformed by means of the rule `TransitionComposite2Transition` into the target `print` transition from the composite state `Active` to the state `Printing`. During the execution of the transformation, the engine is able to infer the model with uncertainty by analizing the tracing information. For example, the point of uncertainty for the transition `print` is derived by calculating the alternatives among model elements involved in the above trace links. For example, the trace link `l1` in lines `1-5` and the trace link `l2` in lines `7-11` refer to the same source element (the transition `print` from the state `Active` to the state `Printing`), but the application of two different rules (`TransitionComposite2Transition` and `Transition2Transition`) generates a pair of different target elements (the transition `print` from `Active` to `Printing` and the transition `print` from `Authentication` to `Printing`). Thus, the two target elements are the alternatives of the uncertainty point for the transition `print`. Similarly, the trace links `l3` and `l4` in lines `13-23` contribute to the same uncertainty point.

The resulting model with uncertainty corresponding to the admissible 48 solutions illustrated in Sect. 2, is given in Fig. 4. For each uncertainty point a corresponding model elements is created in the appropriate uncertainty metaclass. For instance, the alternative transitions corresponds to the uncertainty points (typed as `UTransition`) namely `UPrint`, `UCompleted` and `UCritical error`, which contains the transitions targeting each one of the nested states within `Active` as well as to the composite state itself.

As already mentioned in Sect. 3, models with uncertainty can over-approximate the solution space. For instance, the scenario in Fig. 1 suggests that only one `print` transition can exist in the final model. However, the generated model with uncertainty could admit also models with multiple `print` transitions giving place to more solutions than those expected. Therefore, in order to avoid multiple `print` transitions, the model with uncertainty in Fig. 3 is refined with reference values that restrict to cases with one `print` transition only.

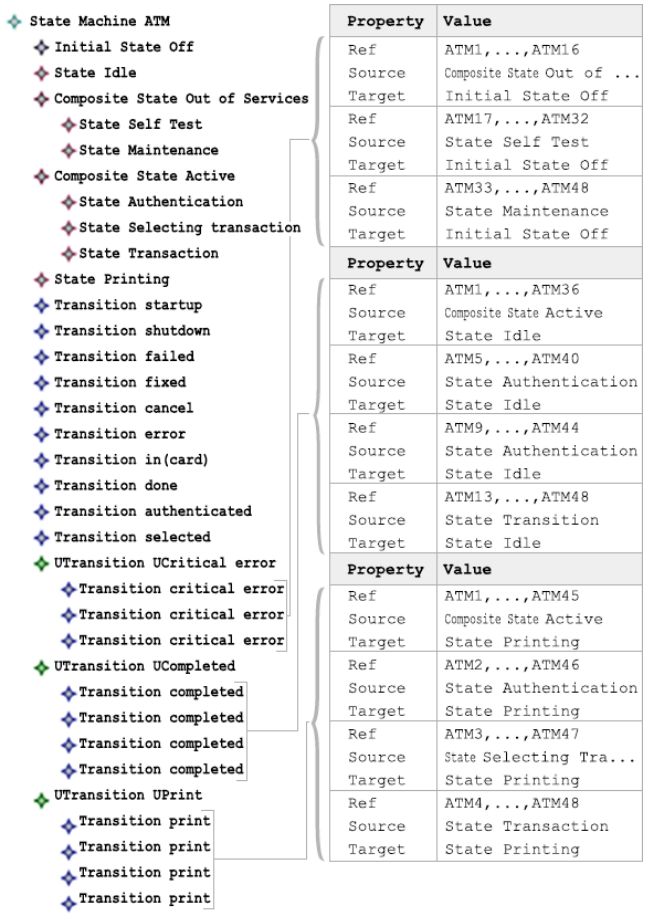

Figure 4. UHSMm model

According to the uncertainty metamodel described in Sect. 4, each metaclass provides an attribute `ref` to maintain a link with the possible solution models. When a transformation is executed, the JTL engine simultaneously derives both the model with uncertainty and the corresponding alternatives, which give place to the solution space. More in details, for each element in the model with uncertainty in Fig. 3, the attribute `ref` contains a list of reference objects, each one refers to the corresponding solution model. For instance, the uncertain transition `print` between the composite state `Active` and the state `Printing` is marked with the references {`ATM1,ATM5,ATM9,...`}; actually, it contains a complete subset of references that represents only the models to which the considered uncertain element can belong to.

Starting from the model with uncertainty and the references, all the correspondent solution models can be obtained. For instance, the following OCL constraint is automatically generated and applied to the model with uncer-

tainty to select all the elements that give place to the solution model with a certain reference value.

```
TracedClass.allInstances()
->select( e:TracedClass | e.ref = <value> )
```

Once the model with uncertainty is obtained, the designer can decide which are the wanted alternatives. The process iterates over each uncertainty point and requires the designer to select for each of them among the available choices. Each designer's decision is recorded by means of an OCL constraint. For instance, if in the uncertainty point `Uprint` the transition targeting the composite state `Active` is selected, the following OCL constraint is used

```
UTransition.allInstances()
->select( e:UTransition | e.name = 'UPrint' )
->collect( e | e.transitions )
->select( t:Transition | t.source.name = 'Active' )
```

As said, the selected element is marked with a list of reference values. Consequently, the invalid alternative may be eliminated from the other uncertainty points, as for instance in the following constraint

```
UTransition.allInstances()
->select( e:UTransition | e.name <> 'Uprint' )
->collect( e | e.transitions )
->select( t:Transition |
   t.ref = <value1> or ... or <valueN> )
```

The idea is to endow the model with uncertainty with an increasing number of constraints for recording designer's decisions and for eliminating inadmissible combinations of alternatives.

6. Evaluation

In this section, we describe an experimental study in order to answer the research question "How does our approach scale with respect to increasing the size of the solution space (i.e., uncertainty)?". In particular, we studied how the approach generates models with uncertainty and compared the results with those obtained by generating the complete solution space.

The experiment is conducted by applying the approach to the Collapse/Expand State Diagrams round-trip benchmark described in Sect. 2. First, the involved metamodels and models are specified in their Ecore format within EMF, then the *HSM2SM* bidirectional transformation has been implemented by means of JTL. The result consists of a transformation with 10 bidirectional rules, which relate hierarchical and flat state machines, and traceability links. An initial version of the *HSM* model contains 20 classes, 29 associations and 30 attributes. The forward execution of the *HSM2SM* transformation generated an *SM* model composed of 12 classes, 18 associations and 19 attributes. At this point, increasingly complex changes have been applied to the generated model. For each modification, the *HSM2SM* transformation has been executed in backward direction both in extensive and intensive mode. In particular,

– *extensional execution*, the first modification on the generated *SM* model does not introduce any uncertainty and the model is (backward) translated into single instance; we gradually increased the degree of uncertainty by changing the *SM* model such that the back-propagation of changes generates a growing number of solution models. With 5 steps, the modification in the target model caused 1, 4, 16, 64, 192 solution models.

– *intensional execution*, the target *SM* model is translated in an uncertainty model without uncertainty points; by operating the same changes as for the extensive case we manage to generated a sequence of models with uncertainty, the last and most complex one with 4 uncertainty points, each with 4, 4, 4, 3 alternative candidates, respectively.

The study aimed at assessing how the approach scales with increasing sets of models. Several measurements have been performed on different parameters including the number of models, the size of models and the execution time of the backward transformations for both the intensive and extensive case. The program has been executed on a machine with an Intel Core i7-4790K 4.00Ghz processor and 8Gb RAM, running Windows 8.1. The results are summarized in Table 1.

In particular, the results demonstrate (as expected) that the models sizes and runtime increase with the increase of the uncertainty. Nevertheless, it is interesting to observe how these values increase. For instance, the model with uncertainty grows proportionally to the number of uncertainty points, instead the number of solution models increases exponentially until reaching 192 models against only 4 uncertainty points. Analogously, the number of elements (i.e., classes, associations and attributes) of the uncertainty model is linear with respect to the number of uncertainty points (and the correspondent alternatives). While, the overall number of model elements of the generated instance exponentially increases leading to a situation whose management is impractical as a. final number of 19,584 elements is reached. The total CPU time for the intensive execution grows mildly from 218 to 531 msec, on the contrary the total runtime for the extensive execution grows until almost 27 seconds showing that the extensive case requires increasing CPU resources. On the contrary, the memory usage is almost comparable and shows only minimal differences in both the executions [4].

The advantages of generating directly models with uncertainty are evident also from the curves given in Fig. 5. The evaluation does not assess any effort required in finding a model among the solution space or in the model with uncertainty as this was outside the scope of this paper.

Threats to Validity. The first threat to validity is the choice of input models. Although our intention is to validate the approach with a more extended set of (randomly generated)

[4] The interested reader can access the implementation at http://jtl.di.univaq.it/

		Step 1	Step 2	Step 3	Step 4	Step 5
Extensive	# solution models	1	4	16	64	192
	# elements	79	348	935	6,208	19,584
	CPU time (msec)	265	453	1,156	6,265	26,640
	memory (bytes)	3,272,704	3,284,992	3,371,008	3,649,536	3,674,112
Intensive	# uncertainty points	0	1	2	3	4
	# elements	79	103	124	145	161
	CPU time (msec)	218	234	281	406	531
	memory (bytes)	3,432,448	3,543,040	3,573,408	3,747,840	3,792,896

Table 1. Summary of the evaluation

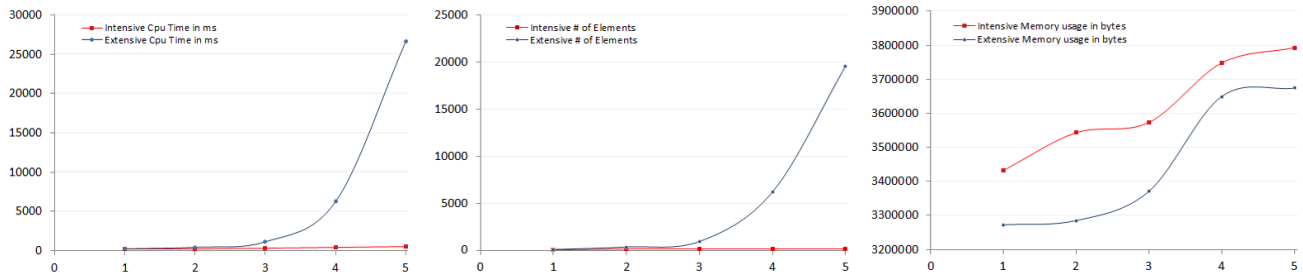

Figure 5. Comparing results

models, the results encourage us in thinking that they can be at some extent independent of the input models. The second threat is given by the choice of the transformation which is relatively small. However, a more complex transformation would correspond to possibly more non-deterministic rules which can only worsen the problem, although it might be not that straightforward that the CPU occupation follows the same trend.

7. Related work

Uncertainty is ubiquitous within contexts such as requirements engineering [7], software processes [15] and adaptive systems [21]. Uncertainty management has been studied in many works, often with the intention to express and represent it in models. In [11], the notion of *partial model* is introduced in order to let the designer specify uncertain information by means of a base model enriched with annotations and first-order logic. Model transformation techniques typically operate under the assumption that models do not contain uncertainty. Nevertheless, the work in [12] proposes a technique for adapting existing model transformations in order to deal with models containing uncertainty. In particular, a lifting operation permits to adapt unidirectional transformations for being used over models with uncertainty preserving their original behavior. In [20] a formal approach called MAVO is proposed and applied to design models in order to express and allow automated reasoning in presence of uncertainty.

Concerning the multiplicity of solutions in bidirectional transformations, most of the existing languages are deterministic, i.e., they produce one model at time. However, in [2] the authors propose PROGRES, a bidirectional transformation solution based on Triple Graph Grammars (TGGs) which is able to recognize ambiguous mappings and in case resolve them by interactively asking the user, who needs to be an experts in these techniques. In [24] the QVT-R bidirectional transformation language is discussed. In particular, the author observes that the formal semantics of QVT-R is ambiguous and it is not possibile to conclude that QVT-R supports non-bijective transformations. In [1] formal and generalized study about bidirectional transformations is given. The proposed techniques formally describes not only traditional bidirectional transformations but extends the discussion to non-deterministic transformations as well. In this respect, it represents a possible alternative to the discussion we proposed in Sect. 3, however the scope of our proposal is limited to the discussion about the uncertainty. An attempt in making bidirectional transformation deterministic by means of intentional updates is represented by the BiFluX language [25], however the problem that a transformation cannot be tested for non-determinism at static-time reduces its effectiveness. Recently some interesting solutions based on *lenses* have been proposed: [6] illustrates a

technique to support bidirectional transformations relying no more on mapping between models but across manipulations (or differences) operable on them. However, the management of non-bijective problems is not clearly addressed. Finally, the ability to deduce and generate all the possible solutions of an uncertain transformation has been achieved by few approaches, including JTL [8, 9]. In [18] the authors propose a bidirectional transformation approach in which the QVT-R semantics is implemented by means of Alloy. Different generated alternatives may be obtained from the execution of a model transformation and reduced by adding extra OCL constraints or by limiting the upper-bound search criteria. While in [3] similar results are obtained by using a variety of integer linear programming. These approaches introduced over the last few years on one hand demonstrate that there is a need for an in-depth discussion about the nature of bidirectionality; and on the other hand, shown how mature techniques and semantics are available for dealing with an intrinsically difficult problem.

8. Conclusion and Future Work

In order to make non-bijective transformation deterministic implementors must provide a general consistency-restoration strategy. Due to lack of knowledge, this is not always possible. In these cases, languages like JTL can resolve the underspecification non-deterministically: given a change to one source, external solvers find one or more consistent choices for the other source. Interestingly, this can be seen as an uncertainty reducing process: the unknown uncertainty at design-time is translated into known uncertainty at run-time. Unfortunately, this mechanism often generates a large number of alternatives, which are difficult to be manually inspected and analyzed by the designer. In this paper, we proposed a metamodel-independent approach that enables to represent a solution space by means of a model with uncertainty. A refinement of the JTL engine is proposed to accommodate the new intensional semantics. Finally, an evaluation has been conducted demonstrating that models with uncertainty are more scalable (with respect to space and time) compared to their extensional counterpart.

Future plans include the possibility to assiste the modeler with proper visualization, which is of great relevance for enhancing usability. Moreover, we would like to investigate model slicing techniques for hiding, partitioning, and abstracting models with uncertainty and record the operations in terms of constraints.

Acknowledgments

This research was supported by the EU through the Model-Based Social Learning for Public Administrations (Learn Pad) FP7 project (619583).

References

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

[2] S. M. Becker, S. Herold, S. Lohmann, and B. Westfechtel. A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *Software and System Modeling*, 6(3):287–315, 2007.

[3] G. Callow and R. Kalawsky. A Satisficing Bi-Directional Model Transformation Engine using Mixed Integer Linear Programming. *JOT*, 12(1):1: 1–43, 2013.

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

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

[6] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations. pages 61–76, 2010.

[7] C. Ebert and J. D. Man. Requirements uncertainty: influencing factors and concrete improvements. In *Procs. of ICSE*, pages 553–560. ACM Press, 2005.

[8] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. A model-driven approach to automate the propagation of changes among Architecture Description Languages. *SOSYM*, 1(25):1619–1366, 2010.

[9] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo. Change management in multi-viewpoint system using asp. In *EDOCW08*, pages 433–440. IEEE Computer Society, 2008.

[10] R. Eramo, A. Pierantonio, and G. Rosa. Uncertainty in bidirectional transformations. In *Procs. of MiSE 2014*, 2014.

[11] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE*, pages 573–583, 2012.

[12] M. Famelis, R. Salay, A. D. Sandro, and M. Chechik. Transformation of models containing uncertainty. In *MoDELS'13*, pages 673–689, 2013.

[13] D. Garlan. Software engineering in an uncertain world. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 125–128. ACM, 2010.

[14] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Procs of ICLP*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[15] H. Ibrahim, B. H. Far, A. Eberlein, and Y. Daradkeh. Uncertainty management in software engineering: Past, present, and future. In *CCECE*, pages 7–12. IEEE, 2009.

[16] K. Knopp. *Theory of Functions, Parts I and II*. Courier Corporation, July 2013.

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

[18] N. Macedo and A. Cunha. Implementing QVT-R Bidirectional Model Transformations Using Alloy. In *FASE*, pages 297–311, 2013.

[19] Object Management Group (OMG). MOF 2.0 QVT Final Adopted Specification v1.1, 2011. OMG Adopted Specification formal/2011-01-01.

[20] R. Salay, M. Chechik, J. Horkoff, and A. D. Sandro. Managing requirements uncertainty with partial models. *Requir. Eng.*, 18(2):107–128, 2013.

[21] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *RE*, pages 95–103. IEEE, 2010.

[22] D. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[23] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.

[24] P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *SOSYM*, 8, 2009.

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

A. HSM2SM bidirectional transformation in JTL

In this appendix, we illustrate a fragment of the HSM2SM bidirectional transformation, which relates hierarchical and flat state machines one with another. It is implemented by means of the constraint-based model bidirectional transformation language, called JTL. It adopts a QVT-R like syntax to specify a number of *relations* among elements of the two involved *domains*. In particular, in Listing 1 the following relations are reported:

a) *Transition2Transition*, which relates transitions of the hierarchical metamodel (where non composite state are involved) and transitions of the flat metamodel,

b) *TransitionComposite2Transition*, which relates transitions of the hierarchical metamodel (where composite state are involved) and transitions of the flat metamodel,

c) *TransitionSourceComposite2TransitionSource*, which relates source composite states of transitions of the hierarchical metamodel and the corresponding source states of transitions of the flat metamodel, and finally

d) *TransitionTargetComposite2TransitionTarget*, which relates target composite states of transitions of the hierarchical metamodel and correspondent target states of transitions of the flat metamodel.

```
1 transformation hsm2sm(source: HSM, target: SM) {
2 ...
3  top relation Transition2Transition {
4   enforce domain source sourceTrans: HSM::Transition{
5     owningStateMachine = sourceSM: HSM::StateMachine {},
6     source = sourceState : HSM::State {},
7     target = targetState: HSM::State {}
8   };
9
10  enforce domain target targetTrans: SM::Transition{
11    owningStateMachine = targetSM: SM::StateMachine {},
12    source = sourceState : SM::State {},
13    target = targetState: SM::State {}
14  };
15
16  when {State2State(sourceState, targetState) and
17   sourceState.owningCompositeState.oclIsUndefined();
18  }
19
20  where {...}
21 }
22
23 top relation TransitionComposite2Transition {
24  enforce domain source sourceTrans: HSM::Transition{
25    owningStateMachine = sourceSM: HSM::StateMachine {}
26  };
27
28  enforce domain target targetTrans: SM::Transition{
29    owningStateMachine = targetSM: SM::StateMachine {}
30  };
31
32  when {sourceTrans.source.oclIsTypeOf(CompositeState)
33   or sourceTrans.target.oclIsTypeOf(CompositeState)
34  }
35
36  where {TransitionSourceComposite2TransitionSource
37    (sourceTrans,targetTrans),
38    TransitionTargetComposite2TransitionTarget
39    (sourceTrans,targetTrans)
40  }
41 }
42
43 relation TransitionSourceComposite2TransitionSource {
44  enforce domain source sourceTrans: HSM::Transition {
45    source = sourceState : HSM::CompositeState {}
46  };
47
48  enforce domain target targetTrans: SM::Transition {
49    source = targetState : SM::State {}
50  };
51
52  when {
53   CompositeState2State(sourceState, targetState);
54  }
55 }
56
57 relation TransitionTargetComposite2TransitionTarget {
58  enforce domain source sourceTrans: HSM::Transition {
59    target = sourceState : HSM::CompositeState { }
60  };
61
62  enforce domain target targetTrans: SM::Transition {
63    target = targetState : SM::State { }
64  };
65
66 when {
67  CompositeState2State(sourceState, targetState);
68 }
69 }
70 ...
```

Listing 1. A fragment of the HSM2SM transformation written in JTL

The *when* and *where* clauses specify conditions on the relation. In particular, the when clause in lines `16-18` implies that the `source` of the `Transition` belongs to the hierarchical model must be a `State` and not a `CompositeState`. Whereas, the when clause in lines `32-34` implies that the `source` or the `target` of the `Transition` belongs to the hierarchical model must be a `CompositeState`. The where clause in lines `36-40` triggers the relations

`TransitionSourceComposite2TransitionSource` and `TransitionTargetComposite2TransitionTarget`.

The described relations are bidirectional, in fact both the contained domains are specified with the construct *enforce*. Besides the main language constructs, the transformation comprises also OCL operators, like `oclIsUndefined()` which returns true whenever invoked on a defined feature.