Translating target to source constraints in model-to-model transformations

Jesús Sánchez Cuadrado
Universidad de Murcia, Spain

Esther Guerra, Juan de Lara
Universidad Autónoma de Madrid, Spain

Robert Clarisó, Jordi Cabot
Universitat Oberta de Catalunya, Spain

***Abstract*—Model transformations are used to automate model manipulation in Model-Driven Engineering (MDE). In particular, model-to-model transformations produce target models (conformant to a target meta-model) from source ones (conformant to a source meta-model). While transformation correctness is crucial in MDE, developing transformations is error-prone due to the difficulty in testing them. This problem is further aggravated if the source and target meta-models contain OCL integrity constraints, as every transformed source model should satisfy the target integrity constraints.**

In order to attack this problem, we present a novel method that translates target OCL constraints to the source meta-model using the transformation definition. This way, if a source model satisfies the advanced constraint, the transformed model will satisfy the target constraint. The method has been implemented for the ATL transformation language and integrated with the anATLyzer tool. We show its benefits in combination with model finders, and the promising results of its validation using mutation techniques and transformations developed by third parties.

***Index Terms*—Model-driven engineering; model transformations; integrity constraints; OCL; quality**

I. Introduction

Model-Driven Engineering (MDE) is a software development methodology where the central assets are *models*. The structure of models is defined by a *meta-model*, which describes the catalogue of modelling elements (e.g. using a class diagram) and their well-formedness rules expressed as a set of integrity constraints (e.g. using an assertion language like OCL). It is critical to ensure that a model *conforms* to its meta-model, because non-conformant models may go unnoticed and produce errors when used at a later stage.

Many MDE activities require using *model transformations* [1], [2]. These receive *source* models and produce *target* models, either in-place or out-place, and are typically described using rule-based languages, like ATL [3] or QVT [4]. A major concern in MDE is the correctness of transformations, as this directly impacts the outcome of the development process. Unfortunately, transformations are error-prone [5] and hard to test and debug [6], because their operation is divided among several rules that may interact. Furthermore, integrity constraints can have an effect on transformation rules. For example, a rule may be impossible to apply in a source model if the rule application conditions contradict a source integrity constraint, while certain rule applications may produce ill-formed models that violate the target meta-model constraints.

In this work, we focus on a specific kind of constraints: those defined over the target meta-model of a model-to-model transformation. In particular, we propose a method that rewrites the OCL constraints in the target meta-model as constraints over the source meta-model, using the transformation definition backwards. We call a source constraint obtained in this way an *advanced constraint*. Advanced constraints can be employed in a variety of ways. They can be used to analyse transformation correctness, from a strong executability point of view (i.e. given any valid source model, the produced target model conforms to the target meta-model and satisfies its integrity constraints). In this scenario, we can use model finders (a constraint solver over models, like Alloy [7] or the USE Validator [8]) to check if some source model leading to an incorrect target model can possibly exist. If it can, then the advanced constraints can be used as a *transformation pre-condition* to rule out source models whose transformation will be incorrect. Our method can also be used for testing (to generate interesting source models taking into account the target constraints), and for iterative transformation development as developers can specify expected target properties which are advanced and used for generating (counter-)example models.

The applicability of our method is demonstrated with an implementation for the ATL transformation language (one of the most widely used nowadays) on top of the static analyzer anATLyzer [5]. We report on several experiments which show that: (a) the method is useful, as it can detect errors in real transformations developed by third-parties; (b) the method has almost perfect precision and recall, as demonstrated by a preliminary evaluation using mutation testing; and (c) the computation of pre-conditions has little computational effort.

Paper organization. Section II motivates our work and presents a running example using ATL. Then, Section III introduces our method, while Section IV discusses its applicability. Section V describes tool support, and Section VI evaluates its effectiveness. Finally, Section VII discusses related work and Section VIII draws some conclusions.

II. Background and Running Example

In this section, we motivate the need for the method, provide some background information, and introduce a running example that will be used throughout the paper.

Let us assume we are interested in analysing manufacturing plants. To specify the layout of these factories, we have a domain-specific language (DSL) with primitives for modelling different types of machines, conveyors connecting them, and parts transported by conveyors and processed by machines.

The left of Fig. 1 shows the meta-model of this DSL. It considers three types of machines: generators that inject a given amount of parts at a time into the factory up to max times, assemblers that consume and process parts, and terminators that remove parts from the factory. Moreover, class Generator declares the OCL invariant posAmnt which demands any generator object to have a non-negative amount.

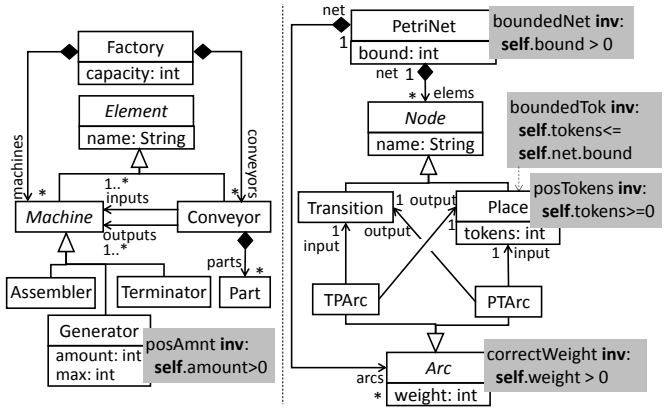

Fig. 1. Meta-models of factories (left) and Petri nets (right).

In order to analyse bottlenecks and plant optimizations, we decide to transform factory models into Petri nets, a formalism supported by powerful analysis methods [9]. Fig. 1 shows to the right the meta-model for Petri nets. A Petri net is a bipartite graph with two kinds of vertices: places and transitions. Places can be connected to transitions via PTArcs, and transitions can be connected to places using TPArcs. According to invariants correctWeight and posTokens, arcs have a positive weight, and places may contain zero or more tokens.

Roughly, the transformation from factories to Petri nets should transform machines into transitions, conveyors into places, and parts into tokens. Moreover, every valid factory model should be transformed into a valid Petri net that satisfies the target invariants boundedNet, boundedTok, posTokens and correctWeight, and fulfils the cardinality bounds of references. These latter constraints can be recasted as OCL invariants to allow their uniform treatment with the rest of invariants [10].

In order to tackle this scenario (and others described in Section IV), we propose a method that, given a model-to-model transformation and a set of invariants of the target meta-model, translates those invariants (using the transformation) into the source meta-model context. In this way, the translated invariant can be added to the source meta-model to constrain the valid input models, can be converted into a transformation pre-condition, or can be used with model finders [7] to analyse whether some/all source models can be transformed into target models with (un)desirable properties. A model finder is a program – frequently based on SMT or SAT solving – which takes a meta-model and a set of constraints and produces an instance model satisfying all the constraints, if such a model exists within the given search bounds.

Our method is generally applicable to most rule-based transformation languages, but requires the following features:

```
1  module factory2pn;
2  create OUT : PN from IN : FAC;
3
4  rule Factory2PN1 {
5   from f : FAC!Factory ( f.capacity > 0 )
6   to   pn : PN!PetriNet (
7     elems ← f.conveyors→union(f.machines),
8     bound ← f.capacity
9   )
10 }
11
12 rule Factory2PN2 {
13  from f : FAC!Factory ( f.capacity <= 0 )
14  to   pn : PN!PetriNet (
15    elems ← f.conveyors→union(f.machines),
16    bound ← 1
17  )
18 }
19
20 rule Input2TPArc {
21  from m : FAC!Machine, c : FAC!Conveyor (c.inputs→includes(m))
22  to   a : PN!TPArc (
23    weight ← if (m.oclIsKindOf(FAC!Generator)) then m.amount else 1 endif,
24    input ← m,
25    output ← c,
26    net ← m.factory
27  )
28 }
29
30 -- similar rule for outputs omitted for brevity
31
32 rule Machine2Transition {
33  from m : FAC!Machine (m.oclIsKindOf(FAC!Assembler) or m.oclIsKindOf(
          FAC!Terminator))
34  to   t : PN!Transition (  name ← m.name )
35 }
36
37 rule Gen2Transition {
38  from m : FAC!Generator
39  to   t : PN!Transition  ( name ← 'tr'+m.name  ),
40  p : PN!Place (
41    name ← 'pl'+m.name,
42    tokens ← m.max, -- possible error, as max can be negative
43    net ← m.factory
44  ),
45  p2t : PN!PTArc (
46    weight ← 1,
47    input ← p,
48    output ← t,
49    net ← m.factory
50  )
51 }
52
53 rule Conv2Place {
54  from c : FAC!Conveyor
55  to   plc : PN!Place (
56    name ← c.name,
57    tokens ← c.parts→size(),
58    net ← c.factory
59  )
60 }
```

Listing 1. Excerpt of example transformation.

- The source model is read-only
- The target elements cannot be rewritten once created
- Declarative rules with no imperative constructs like loops
- Every rule is applied once per match of the input pattern

Many languages or relevant subsets of them fulfil these criteria, like ATL [3], ETL [11], QVT [4] and TGGs [12]. To ground the discussion, we instantiate the method for ATL, a widely used representative of rule-based languages.

Listing 1 shows part of the ATL transformation from factories to Petri nets. An ATL transformation typically transforms a model conforming to a source meta-model into a model

conforming to a target meta-model (see line 2 of the listing). For this purpose, it defines rules that state how to create target objects given a pattern of source objects. Rules can also have guards: OCL expressions that restrict the rule applicability to the matched source objects that satisfy them. For example, rule Factory2PN1 in line 4 creates a PetriNet object for each Factory object in the source, and its guard f.capacity $>$ 0 ensures this rule is applied only to factories with positive capacity.

The value of the fields of the created objects is defined using bindings of the form field $\leftarrow$ expression. Fields in bindings can be of primitive type (like weight $\leftarrow$ 1 in line 46), or references to which it is possible to assign target objects (like input $\leftarrow$ p in line 47) or source objects (like elems $\leftarrow$ f.conveyors$\rightarrow$union(f.machines) in line 7). In the latter case, a binding resolution algorithm assigns to the field the target objects created from the assigned source objects.

The rules in Listing 1 are matched rules, which are executed once per match. A run-time error is raised if an object in the source model is matched by more than one matched rule with one element in its source pattern. In contrast, lazy rules must be explicitly called with the objects to which apply the rule as parameters. ATL also supports helpers, which are operations defined globally or in the context of a meta-model class, and return the result of evaluating an OCL expression.

Altogether, we would like to ensure that transformations (like the one in Listing 1) do not produce incorrect target models. For this purpose, we have developed a method which is presented in the next section.

III. Method

Our method rewrites an integrity constraint defined over the target meta-model (C_{tgt}) into an integrity constraint in the source meta-model (C_{src}). The new constraint is equivalent to the original, in the sense that for any source model (s) and the output model produced by the transformation ($trafo(s)$) the following equivalence holds: $C_{src}(s) \leftrightarrow C_{tgt}(trafo(s))$.

Example. Consider the invariant posTokens inv: self.tokens>=0 defined in Place. Our aim is to verify whether any output model of the transformation will satisfy the constraint. To this end, our method derives the advanced constraint in Listing 2, formulated in a global scope over the source meta-model.

```
1 Generator.allInstances()→forAll(p | p.max >= 0) and
2 Conveyor.allInstances()→forAll(p | p.parts→size() >= 0)
```

Listing 2. Generated advanced constraint for posTokens inv: self.tokens>=0

Intuitively, our method applies the transformation backwards to the target constraint. As a preprocessing step, the constraint is recasted to a global scope, i.e., Place.allInstances()→forAll(p | p.tokens >= 0). Then, we identify the rules that create objects of the types used in the invariant. In the example, two rules create objects of type Place, which are Gen2Transition (line 37) and Conv2Place (line 53). This means the transformation has two paths for which the forAll quantifier in the invariant must be checked. Our method rewrites the invariant for each path, and concatenates the resulting expressions using a suitable operator, and in this case. The rewriting of the first path corresponds to the execution of rule Gen2Transition, and hence, it replaces Place.allInstances() by Generator.allInstances(), and the attribute access p.tokens by its source counterpart. Since the attribute access is within a forAll over Place objects, it can be simply replaced by the right-hand side of the binding tokens $\leftarrow$ m.max. The same strategy is applied for the second path corresponding to rule Conv2Place.

Fig. 2 summarizes the main steps in our method, namely:

1) *Convert into global scope*: First, the method translates the constraint into a global scope as follows:

$$context\ T\ inv:\ p(self) \Rightarrow (rewritten\ into)$$
$$T.allInstances() \rightarrow forAll(x|p(x))$$

2) *Compute constraint transformation tree*: Next the method computes an expression tree stating the path conditions under which a source model will create and initialize the target types and features used in the constraint. We call this expression the constraint transformation tree (CTT).
3) *Rewrite*: Finally, the method replaces references to target model elements with equivalent references to source model elements, based on the transformation definition.

A last phase for optimizing and simplifying the resulting constraint can be added, but is left for future work. In the remaining of the section, we detail steps 2 and 3 of the method.

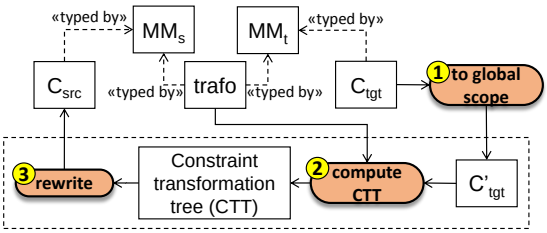

Fig. 2. Steps for advancing C_{tgt} into C_{src}.

A. Computing the constraint transformation tree

Starting from the target constraint expressed in a global scope, we build its CTT. This makes explicit all execution paths leading to the creation and initialization of types and features referenced in the constraint, and identifies the rules and bindings executed in each path. The CTT is the basis to perform the subsequent rewriting which actually produces the advanced constraint.

Example. Consider the constraint boundedTok inv: self.tokens <= self.net.bound in Fig. 1. It is slightly more complex than the previously used posTokens, as it contains a reference navigation (self.net.bound). Fig. 3 shows the constraint formulated in a global scope (upper part) and the generated CTT (middle).

The CTT is an expression tree with two types of nodes: *link nodes* and *split nodes*. Link nodes are abstract syntax nodes similar to those of the OCL abstract syntax, but extended with a context element $\langle Rule, OutputPatternElement \rangle$ that indicates how the transformation translates the node in the original constraint. For instance, the expression PN!Place.allInstances()

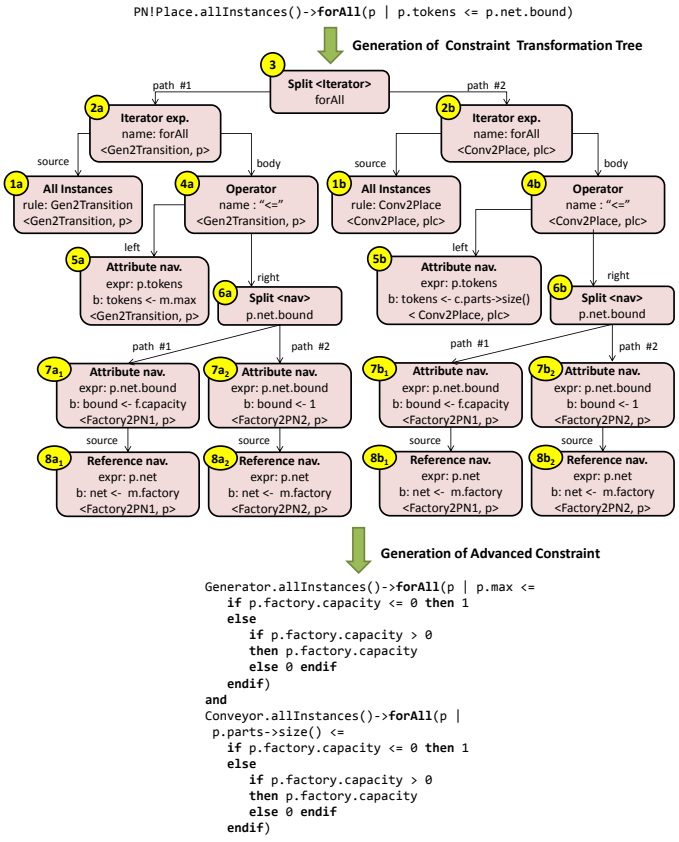

Fig. 3. Generation of advanced constraint from invariant boundedTok.

is represented as a special kind of operation call node, called AllInstances (label 1a), and defines the context ⟨Gen2Transition, p⟩ because rule Gen2Transition creates Place objects using the output pattern p : PN!Place (line 40). Split nodes represent a fork in the control flow that occurs when evaluating some other node (e.g., Split <Iterator>, see label 3).

The CTT is created by performing a depth-first traversal of the original invariant. The key point is that the context information is propagated bottom-up, starting from the nodes corresponding to allInstances calls. The processing of each node depends on its type (whether it is a specific operation call, a navigation, an attribute access, etc.)

Next, we focus on how to handle the basic types of expressions, using the running example to illustrate the technique.

1) allInstances operation: For each call $T.allInstances()$ in the invariant, we look for rule's output pattern elements whose type T_{ope} is the same or a subtype of T. In the example, these are the output pattern elements that generate Place objects in rules Gen2Transition and Conv2Place (lines 40 and 55). For each output pattern element, we create an AllInstances node initialized with the context $\langle Rule, OutputPatternElement\rangle$. Nodes labelled 1a and 1b in Fig. 3 have contexts ⟨Gen2Transition, p⟩ and ⟨Conv2Place, plc⟩.

As we have just seen, the evaluation of an OCL element may yield several link nodes. For example, the evaluation of Place.allInstances returns nodes 1a and 1b. These nodes are propagated to the parent node for their processing. When the parent node receives several nodes, it applies their creation rules once for each node, and then, it introduces a proper split node intended to merge all the paths in the rewriting phase.

2) Iterator expression: An iterator expression has the form $\langle sourceExp\rangle -> iteratorName(var|\langle body\rangle)$. Following the depth-first tree traversal, we first translate the source expression. In the example, the source expression is Place.allInstances, which is translated into the two AllInstances nodes 1a and 1b. Hence, we need to generate two IteratorExp nodes (labels 2a and 2b), which are merged using a split iterator node (label 3). The body of the iterator is also evaluated twice, passing the corresponding context to each path, that is, ⟨Gen2Transition, p⟩ and ⟨Conv2Place, plc⟩.

This strategy allows representing the fact that any Place object must fulfil the body of the forAll, and that there are two rules that generate Place objects. In the rewriting phase, both paths will be merged with an and connective to ensure that whatever path is taken by the execution, the target constraint will be satisfied.

3) Attribute navigation: This is a feature access of the form $\langle sourceExp\rangle.feature$ with primitive type. It is translated to an Attribute navigation node that refers to the binding that initializes the feature. The binding is sought within the context obtained from the evaluation of sourceExp. The same attribute navigation may be evaluated several times with different contexts, depending on the path that is traversed during the execution. This is the case of the translation of the expression p.tokens. In the left branch of the example (label 5a), the context is ⟨Gen2Transition, p⟩, and therefore, the relevant binding is tokens ← m.max. In the right branch (label 5b), the relevant binding is tokens ← c.parts→size() with context ⟨Conv2Place, plc⟩.

4) Reference navigation: When the navigated feature is a reference, we perform a static check to obtain the set of rules that may resolve the binding. For instance, the resolving rules for the binding net ← m.factory are Factory2PN1 and Factory2PN2 (lines 4 and 12 in Listing 1). Then, we create a Reference navigation node for each resolving rule. This node represents a context change, since the existence of an object in the given reference requires the execution of the resolved rule. For instance, the target sub-expression p.net will always yield a PetriNet object if either rules Factory2PN1 or Factory2PN2 are executed. Therefore, the parent subexpression p.net.bound will use these rules as its new context (e.g., nodes $7a_1$ and $8a_1$).

In a navigation expression like p.net.bound, the split node is placed at the end of the call chain (i.e. when all sub-expressions of the navigation have been processed). This is needed to represent the different ways of obtaining this target expression from the source paths (see nodes 6a and 6b).

Other ATL features. In addition to the basic ATL features described above, our method also handles more advanced constructs like lazy rules, helpers, and bindings assigning target objects. Due to space limits, we just sketch them next.

To handle the direct assignment of target objects in bindings (e.g. line 48 in Listing 1) we introduce a special kind of Reference navigation node. This node does not resolve any rule,

TABLE I
TRANSLATION OF A SPLIT NODE WITH N PATHS

Id	Iterator	Advanced constraint
1	<split−expr>→forAll(x|p(x))	<path1>→forAll(x|p(x)) **and** ... **and** <pathN>→forAll(x|p(x))
2	<split−expr>→exists(x|p(x))	<path1>→exists(x|p(x)) **or** ... **or** <pathN>→exists(x|p(x))
3	<split−expr>→one(x|p(x))	<path1>→select(x|p(x))→union(....)→union(<pathN>→select(x|p(x)))→size()=1
4	<split−expr>→select(x|p(x))	<path1>→select(x|p(x))→union(....)→union(<pathN>→select(x|p(x)))
5	<split−expr>→reject(x|p(x))	<path1>→reject(x|p(x))→union(....)→union(<pathN>→reject(x|p(x)))
6	<split−expr>→collect(x|p(x))	<path1>→collect(x|p(x))→union(....)→union(<pathN>→collect(x|p(x)))
Id	**Operation**	**Advanced constraint**
7	<split−expr>→size()	<path1>→size() + ... + <pathN>→size()
8	<split−expr>→includes(expr)	<path1>→includes(expr) **or** ... **or** <pathN>→includes(expr)
9	<split−expr>→excludes(expr)	<path1>→excludes(expr) **and** ... **and** <pathN>→excludes(expr)
10	<split−expr>→count(expr)	<path1>→count(expr) + ... + <pathN>→count(expr)
11	<split−expr>→includesAll(expr)	expr→forAll(x | <path1>→includes(x) **or** ... **or** <pathN>→includes(x))
12	<split−expr>→excludesAll(expr)	expr→forAll(x | <path1>→excludes(x) **and** ... **and** <pathN>→excludes(x))
13	<split−expr>→isEmpty()	<path1>→isEmpty() **and** ... **and** <pathN>→isEmpty()
14	<split−expr>→notEmpty()	<path1>→notEmpty() **or** ... **or** <pathN>→notEmpty()
15	<split−expr>→min()	{<path1>→min(), ..., <pathN>→min()}→min()
16	<split−expr>→max()	{<path1>→max(), ..., <pathN>→max()}→max()
17	<split−expr>→union(expr)	<path1>→union(...)→union(<pathN>)→union(expr)
18	<split−expr>→intersection(expr)	<path1>→union(...)→union(<pathN>)→intersection(expr)
19	<split−expr> − expr	(<path1>→union(...)→union(<pathN>)) − expr
20	<split−expr>→symmetricDifference(expr)	(<path1>→union(...)→union(<pathN>))→symmetricDifference(expr)
21	<split−expr>→including(expr)	(<path1>→union(...)→union(<pathN>))→including(expr)
22	<split−expr>→excluding(expr)	(<path1>→union(...)→union(<pathN>))→excluding(expr)
Id	**References resolved by more than one rule**	**Advanced constraint**
23	<exp>.<mono−valued−reference>	**if** <rule1−condition> **then** <path1> **else** ... **if** <ruleN−condition> **then** <pathN> **else** <default−value> **endif endif**
24	<exp>.<multi−valued−reference>	Sequence { <path1>, ..., <pathN>}→flatten()

but just points to the input pattern element from which the assigned target element is created.

ATL supports explicit rule invocation via *lazy rules*. These can be considered functions which receive a source object as input and return a target object as output. If the output pattern element required to translate an *allInstances* operation belongs to a lazy rule, we introduce an AllInstancesLazy node that represents all possible execution paths that may reach the lazy rule. The strategy to compute this path is similar to [5], but with translation rules adapted to our setting.

To handle helpers in target constraints, we identify the helpers call locations and translate their body using the same method as for regular constraints. For regular helpers defined in the transformation, we identify which ones are used by the generated advanced constraint, and attach them to it so that they are available when required.

B. Rewriting the OCL constraint

After creating the CTT, we traverse it to generate the advanced constraint. In each step of the traversal, we apply translation rules specific to the visited kind of node. Each child node generates an OCL expression (a fragment of the new source constraint) that its parent node combines to produce its own result. As an example, Fig. 4 shows the generation of the source constraint starting from the root node of the CTT (node 3), which is a split node.

A *split node* identifies a point in the target constraint where a target value may be obtained from two or more paths of the transformation execution. All these paths must be calculated and their result must be combined using an operator, which depends on the operation performed by the split node. Typical operators are and/or for boolean operations, + for numeric operations, or union for operators returning a collection.

In Fig. 4, split node 3 represents a forAll iterator with two paths, which are combined with the and operator because this part of the constraint must be fulfilled by any execution path.

Table I contains the combinations that we support for split nodes. The rewriting of the split-expr in the split node combines the result of rewriting each path (path1 to pathN) with some operator kind. On the other hand, Table II shows the translation rules for the *link nodes* of the example (we omit other node types for brevity). For example, the rewriting rule for node 2a assembles a new iterator for the nodes generated from the source and body parts of the original iterator.

In Fig. 4, node 1a is handled as shown in row #1 of Table II. If the rule matches a single source object (i.e. the from section of the rule has only one object), then it will create one target object for each source object that satisfies the filter. However, if the rule has several objects in the from section, it will create one target object for each element in the cartesian product of the source objects that satisfy the filter (row #2 in Table II).

The translation of attribute navigations (row #4 in Table II)

TABLE II
TRANSLATION OF LINK NODES

Node	Invariant	ATL	Advanced constraint	Description
AllInstances	T.allInstances() **context** = <r, t>	**rule** r { **from** s : S (filter) **to** t : T }	S.allInstances()→ select(s | filter)	All objects of the source type are obtained via allInstances, and a select expression applies the original filter.
AllInstances	T.allInstances() **context** = <r, t>	**rule** r { **from** s1 : S1, s2 : S2 (filter) **to** t : T }	S1.allInstances()→ product(S2.allInstances())→ select(s1, s2 | filter)	A rule with multiple input elements is translated to a cartesian product operation.
Iterator	<sourceExp>→iteratorName (var | <body>) **context** = <r, t>	–	gen(<sourceExp>)→iteratorName (var | gen(<body>))	Function *gen* generates the code of a node. Nodes $\langle sourceExp \rangle$ and $\langle body \rangle$ are assembled in a new iterator expression.
Feature navigation (attribute)	<sourceExp>.feature **context** = <r, t>	**rule** r { **from** s : S **to** t : T (feature ← <value>)	<value>	The result is the source part of the binding, $\langle value \rangle$. If no binding is found, a default value is returned according to the type of the expression (e.g., 0 for integer, empty string, etc.).

Feature navigation (reference)	<sourceExp>.feature
	context = <r, t>
		rule r {
		from s : S
		to t : T (
		feature ← <value>
		)
		
		−− resolving rule
		rule r2 {
		from s2 : S2 (filter) ... }
			−− multi−valued
			<value>→selectByType(S2)→
			select(s2 | <filter>)
			
			−− mono−valued
			if <value>.oclIsKindOf(S2)
			then if <filter> **then** <value>
			else defaultValue **endif**
			else defaultValue **endif**
				The type comparison (i.e., selectByKind, oclIsKindOf) is not required if the type of $\langle value \rangle$ equals S2.

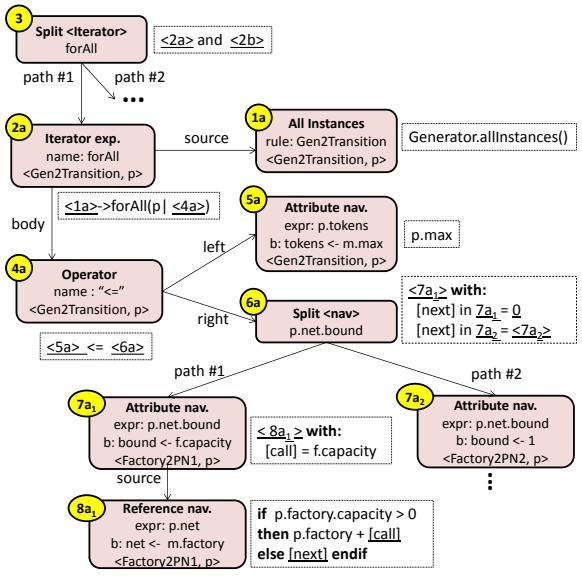

Fig. 4. Rewriting example.

requires copying the value of the corresponding binding, taking care of the variable names. In the example (node 5a), the input variable m of the rule is bound to the iterator variable p of node 2a, and hence, m.max is translated into the correct expression p.max.

The translation of references is more challenging, as it requires encoding all possible combinations of resolving rules. When there is only one resolving rule, a single Reference navigation node suffices, and the translation is as shown in row #5 of Table II. However, if there are two or more resolving rules, the translation of the corresponding split node must encode all possible paths in OCL (rows 23 and 24 in Table I).

For mono-valued references, our strategy is to generate nested ifs, each one checking the application conditions of one resolving rule. Fig. 4 illustrates this scheme for node 6a. The navigation p.net.bound involves two paths, as p.net can be resolved by two rules. For the first path, node $8a_1$ generates a piece of code (an if) with two holes: [call] and [next]. The [call] hole represents an expression (provided by node $7a_1$) that will be concatenated to the translation of p.net (i.e. the binding value stored in the reference navigation node is combined with the value provided for [call]). In this way, the expression p.net.bound gets translated into p.factory.capacity. The [next] hole is filled by node 6a. For the first path, [next] is bound to a default value. The expression generated for the first path is used to fill [next] in the second path. This strategy generates the following expression, which checks the rules' filters to determine the path to follow at runtime.

```
if p.factory.capacity <= 0 then 1 −− Rule Factory2PN2
else −− Rule Factory2PN1
  if p.factory.capacity > 0 then p.factory.capacity else 0 endif
endif
```

The translation of multi-valued references is simpler, as it only requires translating each path and concatenating the results using the template Sequence $\{\langle path_1 \rangle, ..., \langle path_n \rangle\}$ → flatten(). This reflects the semantics of ATL multi-valued bindings.

Altogether, the advanced constraint that results from the example is shown (slightly simplified) at the bottom of Fig. 3.

IV. APPLICATIONS

In this section, we analyse some scenarios where our method is useful, and provide additional examples.

Ensure transformation strong executability. Our method automatically translates target meta-model invariants (like boundedTok) in terms of the source meta-model. Then, a model finder can check if there is some source model satisfying the source meta-model invariants but not the translated ones. If there is some, the advanced constraint can be used as a transformation

pre-condition to forbid the problematic source models and ensure strong executability of the transformation [13].

As an example, Fig. 3 shows the translation of boundedTok to the source. As two of the rules produce places (Gen2Transition and Conv2Place), the constraint has two conjoined parts. As the Petri net bound can be calculated in two ways (given by rules Factory2PN1 and Factory2PN2), each part has a conditional. We can use a model finder to search for factory models satisfying the original source meta-model invariants but not the advanced constraint. Fig. 5(a) shows an example: the conveyor has two parts, which leads to a Petri net that violates boundedTok (see bottom of the figure). Another source of errors are source models (like the one in Fig. 6) where some Generator has a max value bigger than the factory capacity. To solve the problem, we can add the advanced constraint as an invariant of the source meta-model, or as a transformation pre-condition.

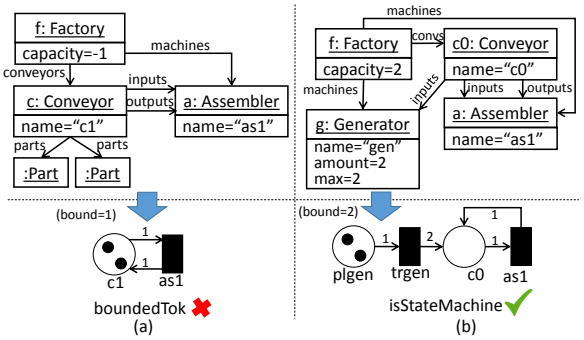

Fig. 5. (a) Witness model showing that boundedTok can be violated. (b) Model transformed into a state machine Petri net. Petri nets are shown using their usual representation: places as white circles, tokens as black dots, transitions as rectangles and arcs as arrows labelled with their weight.

Characterize classes of target models. Another usage scenario is being able to identify which source models yield a certain class of target models of interest. To this aim, the developer provides an OCL expression characterizing the class of target models, and this is translated into the source by our method. Any source model satisfying the advanced constraint corresponds to the class of target models.

Regarding our example, the strength of the Petri net analysis methods depends on the net structure. Simpler classes of Petri nets can be analysed with more powerful methods than nets with more intricate structure [9]. For example, a state machine is a kind of Petri net where every transition has one input place and one output place. Hence, if the result of a transformation is a Petri net with state machine characteristics, we can apply a particular set of analysis methods. In this scenario, it is interesting to characterize the source models (factories) that get transformed into state machine nets. Listing 3 shows the characterization of state machines, and the resulting constraint (simplified). Fig. 5(b) shows a factory model leading to a state machine net. One could also negate the advanced constraint to obtain factories leading to nets that violate the property.

```
1  Transition.allInstances()→forAll(t |
2    TPArc.allInstances()→one(arc | arc.input = t) and
3    PTArc.allInstances()→one(arc | arc.output = t))
4
5  Machine.allInstances()→select(m |
6    m.oclIsKindOf(Assembler) or m.oclIsKindOf(Terminator))→
7    forAll(t |
8      -- TPArc.allInstances()→one(...)
9      Machine.allInstances()→product(Conveyor.allInstances())→
10       select(m, c | c.inputs→includes(m))→
11       one(m, c |
12         (m.oclIsKindOf(Assembler) or m.oclIsKindOf(Terminator)) and m = t)
13     and
14     -- PTArc.allInstances()→one(...)
15     -- PTArc creation in Output2PTArc
16     Machine.allInstances()→product(Conveyor.allInstances())→
17     select(m, c | c.outputs→includes(m))→
18     select(m, c |
19       -- arc.output => mapped to binding output <- m
20       -- which is resolved by rules Machine2Transition and Gen2Transition
21       (m.oclIsKindOf(Assembler) or m.oclIsKindOf(Terminator)) and m = t)→
22       -- PTArc creation in Gen2Transition
23       union(Generator.allInstances()→select(g | g = t))→size() = 1)
24 and ... (similar path for rule Gen2Transition)
```

Listing 3. Characterizing state machine nets (lines 1-3), and equivalent source constraint (lines 5-23).

In the same way, if the target constraint characterizes a single target model, it is possible to use this method to execute the model transformation backwards, i.e. use a model finder to compute source models that produce a specific target model.

Property-based testing and development. During transformation development, it is useful to be able to specify target model properties that the transformation ought to obey. Our method can transform such properties to the source, and then a model finder can check if the transformation preserves them and provide (counter-)examples. These can be used as test cases to test the transformation and its pre-conditions.

In our example, we might want to express that any resulting Petri net should not contain isolated places or transitions. For places, the obtained advanced constraint is unsatisfiable (conveyors cannot be isolated due to the cardinality constraints of their references). For isolated transitions, the advanced constraint is satisfiable by lone assemblers and terminators (but not by lone generators, which are transformed differently).

Contract-based transformation development. Similar to the previous scenario, our method can translate transformation contracts to the source context, which then can be checked using model finders. However, as a difference from the previous scenario, contracts can specify relations between source and target meta-models elements, instead of properties of the target models alone. Nowadays, transformation contracts need to be checked by exhaustive testing, while our method permits checking them using model finding [14], [15].

In our example, we might have a contract stating that the number of transitions in a Petri net should be the same as the number of conveyors in the source factory model. Listing 4 shows the contract and its translation. We can negate the resulting source constraint and invoke the solver. In this case, no witness model is output, indicating that the assumption of the contract is correct.

```
1  Transition.allInstances()→size() = Conveyor.allInstances()→size()
2
3  Machine.allInstances()→
4    select(m | m.oclIsKindOf(FAC!Assembler) or m.oclIsKindOf(Terminator))→
5    size() + Generator.allInstances()→size()
6  = Conveyor.allInstances()→size()
```

Listing 4. Contract (line 1), and equivalent source constraint (lines 3-6).

V. Tool Support

The presented method has been implemented for ATL and is available as part of anATLyzer [5]. The source code, update site for Eclipse, usage instructions and screencasts can be found at http://www.miso.es/tools/anATLyzer.html.

Fig. 6 shows a screenshot of the tool. It is fully integrated in the ATL editor. Transformation pre-conditions, source and target meta-model constraints, and contracts are written as regular ATL helpers with additional anATLyzer annotations (label 1). The Analysis View provides an option to start the analysis of target constraints and contracts (label 2). They are converted into advanced constraints which are fed to the USE validator to search counter-examples. If a counter example is found, the constraint is violated and a witness model can be visualized or saved as an XMI file for its debugging (label 3; advanced constraint and witness for the boundedTok constraint).

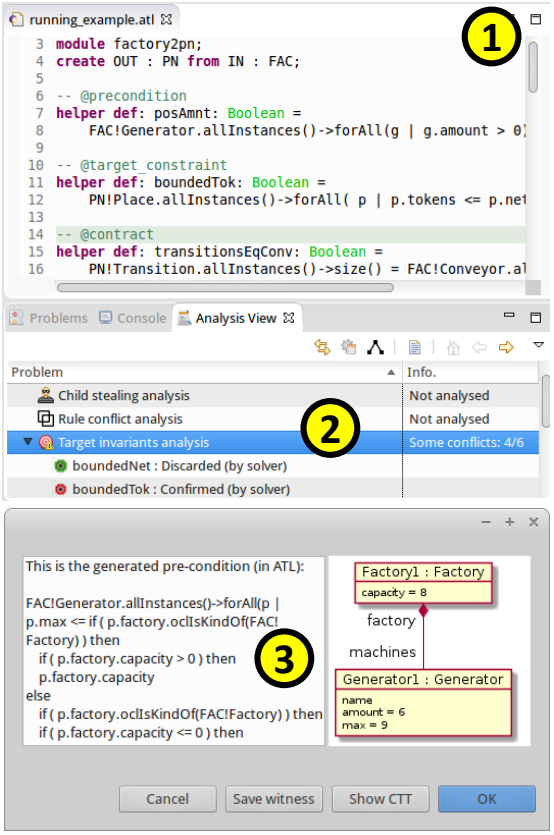

Fig. 6. Screenshot of anATLyzer checking target constraints

Finally, we have specific translators to adapt the advanced constraint output by the method to specific OCL technologies like ATL and USE (e.g., in ATL the product operation is not supported and we rewrite the expression to emulate the behaviour via nested collects and tuples).

VI. Experiments

We have evaluated three aspects of our method. First, we assess whether advancing target constraints and using model finders (first scenario in Section IV) is useful to find errors in transformations developed by third parties. Second, as the method uses model finding, we check whether it leads to acceptable error finding times. Both aspects are analysed in Section VI-A. Finally, in Section VI-B, we assess the accuracy of our method by checking if source models satisfying the advanced constraint lead to correct target models, and conversely, if source models not satisfying it lead to incorrect target ones. The complete experimental data is available at http://miso.es/adv_models.html

A. Usefulness and performance

First, we evaluate the usefulness of our approach to advance target constraints from transformations developed by third-parties. We use the case studies put forward by Büttner and Cheng in [16], [17]: *hsm2fsm*, which flattens hierarchical state machines, and *er2rel*, which transforms ER diagrams into relational schemas. Our method has been able to advance all target meta-model invariants in these transformations. Moreover, we have fed the advanced constraints into the USE Validator, and we have successfully verified them obtaining the same results as the original case studies.

The performance of our method for these cases is shown in Table III. The solving time includes (i) the overhead of translating EMF meta-models into USE meta-models, and (ii) the time of invoking the solver with increasing upper bounds, from 2 to 5 objects per class. Compared to [16] and [17], our method is faster (sometimes an order of magnitude) as the model finder only needs to explore a fragment of the source meta-model [5], instead of the whole transformation model. Moreover, our approach is completely automated and produces a witness model if found. Finally, the time to compute the advanced constraint (difference between total and solving time) is negligible.

TABLE III
Performance analysis (time in seconds). The target invariants marked with * are not ensured by the transformation.

hsm2fsm	Solving time	Total time
unique_fsm_sm_names	0.24	0.26
unique_fsm_state_names	0.40	0.38
fsm_state_multi_lower	0.35	0.36
fsm_state_multi_upper	0.44	0.46
fsm_transition_multi_lower	0.39	0.41
fsm_transition_multi_upper	0.42	0.44
fsm_transition_src_multi_lower *	0.11	0.13
fsm_transition_src_multi_upper	1.35	1.43
fsm_transition_trg_multi_lower	0.63	0.67
vfsm_transition_trg_multi_upper	0.95	1.03
er2rel	Solving time	Total time
unique_rel_schema_names	0.15	0.16
unique_rel_relation_names	0.23	0.24
unique_rel_attribute_names *	0.18	0.22
exist_rel_relation_iskey	0.13	0.15

B. Accuracy

Next, we present the results of an evaluation aimed to assess the accuracy of our method, for which we rely on the *precision* and *recall* metrics. The precision gives a measure of correctness, and in our case, it answers the following question: how

many of the input models that satisfy the advanced constraint generated by our method, actually yield a well-formed target model? The recall gives a measure of completeness, and in our case, it answers the following question: how many of the input models that yield a well-formed target model actually satisfy the advanced constraint generated by our method? We also measure the *true negative rate*, which answers the question: how many of the input models that yield an incorrect target model actually do not satisfy the advanced constraint?

In the evaluation, we have considered the transformations *er2rel* and *hsm2fsm* used in Section VI-A, the running example (fact2pn), and a variant which uses lazy rules (fact2pn$_{lz}$). Table IV shows their characteristics.

TABLE IV
SEED TRANSFORMATIONS IN THE EVALUATION.

	er2rel	hsm2fsm	fact2pn	fact2pn$_{lz}$
No. classifiers (src/tar)	5/3	6/6	8/7	8/7
No. attributes (src/tar)	6/4	3/3	4/5	4/5
No. associations (src/tar)	6/2	5/5	5/6	5/6
No. ATL rules (matched/lazy)	6/0	7/0	7/0	5/2
No. ATL rule filters	3	5	5	3
No. invariants (source/target)	7/4	14/10	1/5	1/5
Invariant complexity (total/avg)	88/8	194/8	27/4,5	27/4,5

Since this is a small sample to obtain meaningful results that can be generalised, we have automatically created new test cases by mutation of the OCL invariants in the target meta-models. We have considered the set of OCL mutations presented in [18], extended with operators that mutate the name of operations and attributes appearing in the constraints by others that are compatible. In this way, starting from a transformation and its meta-models, we have generated new test cases, each one of them obtained by applying a single mutation operator to some target meta-model constraint.

Then, for each test case, the evaluation proceeds as shown in Fig. 7. First, we advance the target constraint to the source. Next, we use constraint solving to generate two sets of input models: the first one contains models that satisfy the advanced constraint, and the second one contains models that violate it. In both cases, we generate models ensuring coverage of all classes in the input meta-model [19]. Finally, we execute the transformation with each input model and check whether the resulting output model satisfies the original constraints of the target meta-model. There are four possible outcomes for each execution: *true positive* (TP), if the input model satisfies the advanced constraint and the output model satisfies the target constraint; *false positive* (FP), if the input model satisfies the advanced constraint but the output model does not satisfy the target constraint; *true negative* (TN), if the input model does not satisfy the advanced constraint and the output model does not satisfy the target constraint; and *false negative* (FN), if the input model does not satisfy the advanced constraint, but the output model satisfies the target constraint. Then, we calculate precision as $\frac{\#TP}{\#TP+\#FP}$, recall as $\frac{\#TP}{\#TP+\#FN}$, and true negative rate as $\frac{\#TN}{\#TN+\#FP}$.

Table V shows the results of the evaluation for each seed

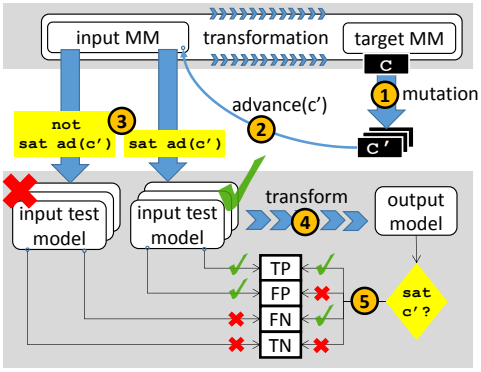

Fig. 7. Main steps in the evaluation process.

transformation and its mutants (first four rows) as well as the aggregated results (last row). The first columns display the number of mutants generated from the target meta-model constraints, and the number of input test models that satisfy (or not) the advanced constraint. Altogether, we have evaluated our method with 232 target invariants over more than 52.500 input models, and our method has only failed 13 times giving a single type of FP. In average, each target invariant was tested with 227 input models, 99 satisfying the source meta-model invariants and the advanced constraint, and 128 satisfying the source meta-model invariants but not the advanced constraint. In all cases, the obtained value for the recall is 1, though the overall precision and recall is 0,99 due to the FPs in cases fact2pn and fact2pn$_{lz}$. This means we obtained almost perfect accuracy, providing a high level of confidence in our method.

The detected FPs are actually caused by the same mutation and refer to a limitation of our method to handle 1-to-n rules. It happens when syntactically correct target invariants test for impossible reachabilities between target objects (like the one created by mutation), which however are satisfiable in the source. Details about this kind of FP are available at http://miso.es/adv_models.html. While this is a rare case, it can be detected statically, and we will improve our method accordingly.

Threats to validity. The experiment has considered a low number of original target invariants (24), though they were mutated to generate variants, so the final number of evaluated invariants is higher (232). The problem is that finding transformations between meta-models with invariants is difficult, and this is the reason why we used mutation. Moreover, by using mutation, we have ensured a higher coverage of the OCL language. On the other hand, the generated input models ensure coverage of the meta-model, but not of the transformation (which is interesting because our method uses the transformation to advance the target constraint, so it makes sense to generate models that exercise all parts of the transformation); this is up to future work. Finally, the transformations in the evaluation cover the most typical ATL features, including matched rules with several elements in the input pattern and lazy rules; however, some less frequent ATL features have not been tested (e.g., rule inheritance and the resolveTemp operation).

TABLE V
EVALUATION RESULTS.

	mutants	input (sat/nosat)	TP	TN	FP	FN	Precision	Recall	Tr. neg. rate
er2rel	46	555 (331/224)	331 (59,64%)	224 (40,36%)	0 (0%)	0 (0%)	1	1	1
hsm2fsm	98	11.376 (6.117/5.259)	6.117 (53,77%)	5.259 (46,23%)	0 (0%)	0 (0%)	1	1	1
fact2pn	44	13.182 (5.439/7.743)	5.438 (41,25%)	7.743 (58,74%)	1 (0,01%)	0 (0%)	0,99	1	0,99
fact2pn$_{lz}$	44	27.569 (11.055/16.514)	11.043 (40,06%)	16.514 (59,90%)	12 (0,04%)	0 (0%)	0,99	1	0,99
total	232	52.682 (22.942/29.740)	22.929 (43,52%)	29.740 (56,45%)	13 (0,0002%)	0 (0%)	0,99	1	0,99

VII. RELATED WORK

The MDE community has devoted many efforts in transformation verification [20], [21], [22], proposing methods based on model checking [23], transformation into formal frameworks [24], [25], graph transformation theory [26], deduction rules and logic [27], and model finders [13], [28], [29], among others. Next we compare with the closest verification approaches, as well as with approaches manipulating OCL conditions in the context of model transformations.

The expected outcome of a transformation can be expressed as a *contract* that may include a pre-condition, a post-condition and other elements [30]. Guerra et al. [31] derive both input test models and oracle functions from transformation contracts. Gogolla et al. [32] propose the notion of *tract*, a contract that includes the information needed to test the transformation (i.e. an input test-suite and a procedure to generate the corresponding outputs). Tracts were originally intended for testing. Our approach can complement them to automatically draw pre-conditions from post-conditions, as seen in Listing 4.

To check transformation contracts, some approaches translate model transformations into transformation models [33] that contain the source and target meta-models, plus OCL conditions derived from the transformation rules. Then, a model finder can give completions of partial transformations, or find source models satisfying or violating some property. However, the scalability of model finders is very sensitive to the meta-model size, and big meta-models enlarge the search space prohibitively [5]. Our method can also be combined with model finders to synthesize source models that violate some target constraint when transformed. However, it has the advantage of a reduced search space, as it only needs to consider the source meta-model. Moreover, our generated source constraints can be used in further scenarios, e.g. as transformation pre-conditions. VeriATL verifies ATL transformations w.r.t. target invariants [28] using a translation to Boogie. The system includes a proof strategy based on natural deduction to localize the faulty rules [27]. Our method can be used for the same purposes with improved performance (see Section VI-A) and covering a wider range of ATL features. To help localizing the fault, we output a witness model.

A few works have targeted the synthesis of pre-conditions for model transformations. For example, Mottu et al. generate input tests models based on input domain partitioning, and then, designers must provide a pre-condition for each *failed input* [34]. While these pre-conditions are specified by hand, our approach generate them automatically. Also, even though its focus is not the synthesis of pre-conditions, *model transformation by example* [35] aims to infer transformation rules from examples of source and target models. As a part of this process, they infer application conditions for the transformation rules from the set of examples. However, given that target models are described via examples, the kind of pre-conditions that can be inferred is very limited.

Our previous work [36] pursues a similar goal but focusing on endogenous graph transformations. As such, the backwards reasoning strategy is completely different from the one presented herein and based on analysing the modification actions that the rule performs. Instead, in this paper, we analyse the possible source objects creating the context of objects of a target constraint, and then use the transformation to translate the constraint into the source context. In the area of graph transformation, the advancement of graph constraints has been studied for in-place transformation [26], [37]. As an exception, Ehrig et al. [38] propagate constraints from source to target along transformations specified with triple graph grammars. However the supported constraints can express only properties of the form "if `P` then `C`", where `P` and `C` are graphs.

Overall, our method has the following advantages: (i) it can produce pre-conditions/source invariants that ensure target model correctness, (ii) it can produce source invariants characterizing source models that get transformed into a given class of target models, (iii) model finding can be used to ensure transformation properties with a reduced state space compared to a more direct approach based on transformation models.

VIII. CONCLUSIONS AND FUTURE WORK

We have presented an automated technique to translate target constraints into the source meta-model of a declarative model transformation. Hence, any source model satisfying the advanced constraints will be transformed into a target model respecting the target constraints. Our method can be used for several quality tasks around transformations including bounded verification, property-based testing, and contract-based development. We have evaluated our method obtaining very good performance, precision and recall values.

In the future, we will work in the simplification of the advanced constraints to make them more readable [39], [40]. We will investigate heuristics to convert the derived source constraints into rule filters, and analyse the possibility of translating source constraints into the target meta-model.

ACKNOWLEDGMENT

Work partially funded by projects MegaM@Rt2 (H2020 ECSEL, #737494), "Open Data for All", RECOM and FLEXOR (Spanish MINECO, TIN2016-75944-R, TIN2015-73968-JIN (AEI/FEDER/UE) and TIN2014-52129-R) and the R&D programme of the Madrid Region (S2013/ICE-3006).

REFERENCES

[1] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, pp. 621–645, July 2006.

[2] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, March 2006.

[3] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31 – 39, 2008, see also http://www.emn.fr/z-info/atlanmod/index.php/Main_Page. Last accessed: Nov. 2010.

[4] QVT, `http://www.omg.org/spec/QVT/`, 2016.

[5] J. S. Cuadrado, E. Guerra, and J. de Lara, "Static analysis of model transformations," *IEEE Transactions on Software Engineering (IEEE Computer Society)*, vol. in press, no. 1, pp. 1–32, 2017.

[6] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Commun. ACM*, vol. 53, no. 6, pp. 139–143, 2010.

[7] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[8] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *MODELS*, ser. LNCS, vol. 7590. Springer, 2012, pp. 415–431.

[9] T. Murata, "Petri nets: Properties, analysis and applications." *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, April 1989.

[10] M. Gogolla and M. Richters, "Expressing UML class diagrams properties with OCL," in *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, ser. LNCS, vol. 2263, 2002, pp. 85–114.

[11] D. S. Kolovos, R. F. Paige, and F. Polack, "The Epsilon Transformation Language," in *ICMT*, ser. LNCS, vol. 5063. Springer, 2008, pp. 46–60.

[12] A. Schürr, "Specification of graph translators with triple graph grammars," in *Proc WG'94*, ser. LNCS, vol. 903. Springer, 1994, pp. 151–163.

[13] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara, "Verification and validation of declarative model-to-model transformations through invariants," *Journal of Systems and Software*, vol. 83, no. 2, pp. 283–302, 2010.

[14] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *Formal Methods for Model-Driven Engineering SFM*, ser. LNCS, vol. 7320. Springer, 2012, pp. 399–437.

[15] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Automated verification of model transformations based on visual contracts," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 5–46, 2013.

[16] F. Büttner, M. Egea, and J. Cabot, "On verifying ATL transformations using 'off-the-shelf' SMT solvers," in *MoDELS'12*, ser. LNCS, vol. 7590. Springer, 2012, pp. 432–448.

[17] Z. Cheng, "Formal verification of relational model transformations using an intermediate verification language," Ph.D. dissertation, Maynooth University, 2016.

[18] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor, "Mutation operators for UML class diagrams," in *CAiSE*, ser. LNCS, vol. 9694. Springer, 2016, pp. 325–341.

[19] F. Fleurey, B. Baudry, P.-A. Muller, and Y. L. Traon, "Qualifying input test data for model transformations," *Software and System Modeling*, vol. 8, no. 2, pp. 185–203, 2009.

[20] M. Amrani, B. Combemale, L. Lucio, G. M. K. Selim, J. Dingel, Y. L. Traon, H. Vangheluwe, and J. R. Cordy, "Formal verification techniques for model transformations: A tridimensional classification," *Journal of Object Technology*, vol. 14, no. 3, pp. 1:1–43, 2015.

[21] D. Calegari and N. Szasz, "Verification of model transformations: A survey of the state-of-the-art," *Electr. Notes Theor. Comput. Sci.*, vol. 292, pp. 5–25, 2013.

[22] L. A. Rahim and J. Whittle, "A survey of approaches for verifying model transformations," *Software and System Modeling*, vol. 14, no. 2, pp. 1003–1028, 2015.

[23] A. Rensink, Á. Schmidt, and D. Varró, "Model checking graph transformations: A comparison of two approaches," in *Proc, ICGT*, ser. LNCS, vol. 3256. Springer, 2004, pp. 226–241.

[24] E. Guerra and J. de Lara, "Colouring: execution, debug and analysis of QVT-relations transformations through coloured Petri nets," *Software and System Modeling*, vol. 13, no. 4, pp. 1447–1472, 2014.

[25] K. Lano, T. Clark, and S. K. Rahimi, "A framework for model transformation verification," *Formal Asp. Comput.*, vol. 27, no. 1, pp. 193–235, 2015.

[26] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.

[27] Z. Cheng and M. Tisi, "A deductive approach for fault localization in ATL model transformations," in *Proc. FASE*, ser. LNCS, vol. 10202. Springer, 2017, pp. 300–317.

[28] Z. Cheng, R. Monahan, and J. F. Power, "A sound execution semantics for ATL via translation validation," in *Proc. ICMT*, ser. LNCS, vol. 9152. Springer, 2015, pp. 133–148.

[29] F. Büttner, M. Egea, J. Cabot, and M. Gogolla, "Verification of ATL transformations using transformation models and model finders," in *ICFEM*, ser. LNCS, vol. 7635. Springer, 2012, pp. 198–213.

[30] G. M. K. Selim, J. R. Cordy, and J. Dingel, "Model transformation testing: The state of the art," in *AMT '12*. ACM, 2012, pp. 21–26.

[31] E. Guerra and M. Soeken, "Specification-driven model transformation testing," *Software and System Modeling*, vol. 14, no. 2, pp. 623–644, 2015.

[32] M. Gogolla and A. Vallecillo, *Tractable Model Transformation Testing*. Berlin, Heidelberg: Springer, 2011, pp. 221–235.

[33] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow, "Model transformations? Transformation models!" in *MoDELS'06*, ser. LNCS, vol. 4199. Springer, 2006, pp. 440–453.

[34] J. Mottu, S. Sen, J. J. Cadavid, and B. Baudry, "Discovering model transformation pre-conditions using automatically generated test models," in *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015*. IEEE Computer Society, 2015, pp. 88–99.

[35] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, *Model Transformation By-Example: A Survey of the First Wave*. Springer, 2012, pp. 197–215.

[36] R. Clarisó, J. Cabot, E. Guerra, and J. de Lara, "Backwards reasoning for model transformations: Method and applications," *Journal of Systems and Software*, vol. 116, pp. 113–132, 2016.

[37] A. Habel, K.-H. Pennemann, and A. Rensink, "Weakest preconditions for high-level programs," in *IGT06*. Springer, 2006, pp. 445–460.

[38] H. Ehrig, F. Hermann, H. Schölzel, and C. Brandt, "Propagation of constraints along model transformations using triple graph grammars and borrowed context," *J. Vis. Lang. Comput.*, vol. 24, no. 5, pp. 365–388, 2013.

[39] M. Giese and D. Larsson, "Simplifying transformations of OCL constraints," in *MODELS05*. Springer, 2005, pp. 309–323.

[40] J. Cabot and E. Teniente, "Transformation techniques for OCL constraints," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 179–195, 2007.