The Journal of Systems and Software 116 (2016) 113–132

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Backwards reasoning for model transformations: Method and applications

Robert Clarisó[a,*], Jordi Cabot[b], Esther Guerra[c], Juan de Lara[c]

[a] Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya, Rambla del Poblenou 156, Barcelona 08018, Spain
[b] ICREA, Barcelona, Spain
[c] Universidad Autónoma de Madrid, Madrid, Spain

A R T I C L E I N F O

Article history:
Received 4 October 2014
Revised 7 August 2015
Accepted 14 August 2015
Available online 22 August 2015

Keywords:
Model transformation
OCL
Weakest pre-condition

A B S T R A C T

Model transformations are key elements of model driven engineering. Current challenges for transformation languages include improving usability (i.e., succinct means to express the transformation intent) and devising powerful analysis methods.

In this paper, we show how backwards reasoning helps in both respects. The reasoning is based on a method that, given an OCL expression and a transformation rule, calculates a constraint that is satisfiable before the rule application if and only if the original OCL expression is satisfiable afterwards.

With this method we can improve the usability of the rule execution process by automatically deriving suitable application conditions for a rule (or rule sequence) to guarantee that applying that rule does not break any integrity constraint (e.g. meta-model constraints). When combined with model finders, this method facilitates the validation, verification, testing and diagnosis of transformations, and we show several applications for both in-place and exogenous transformations.

1. Introduction

1.1. Overview

The advent of model driven engineering (MDE) has prompted the need to manipulate models in an automated way. Common manipulations include model-to-model transformations (or exogenous, where typically input and target models in the transformation conform to different meta-models), as well as in-place transformations like refactorings, animations and optimisations. Many transformation languages and approaches have been proposed for both kinds of transformations, where research is mostly directed towards usable languages providing good integration with MDE standards (e.g. UML, MOF, OCL) and supporting some kind of analysis (Rahim and Whittle, 2015).

We propose to use backwards reasoning to achieve both goals. Backwards reasoning methods have been applied in different domains, for example to analyse logic programs (Howe et al., 2004), Petri nets (Yang et al., 2005) or timed automata (Kwiatkowska et al., 2007). The unifying idea is that, instead of starting with an initial system configuration and exploring possible reachable states, backwards reasoning assumes some (un)desirable target state and computes the corresponding source state(s). In this paper, we present a method that enables backwards reasoning for model transformations, and show its applications both to achieve a better usability of transformation languages and to provide increased analysis capabilities.

Many model transformation languages are based on rules (Ehrig et al., 2006b; Jouault et al., 2008; Kolovos et al., 2008) whose applicability is given by an object pattern complemented with a guard, typically given as an OCL expression. Our backwards reasoning is based on the automated calculation of such guards, given an OCL expression that the model is expected to fulfil after the rule application. Hence, given a constraint C that a model M must satisfy after the application of a rule r, the method generates the weakest constraint C'_r such that if the model satisfies it before applying r, then the resulting model is guaranteed to satisfy C.

The method is agnostic with respect to the particular model transformation language employed, and therefore applicable to many of them—like graph transformation (GT) (Ehrig et al., 2006b), ATL (Jouault et al., 2008) or ETL (Kolovos et al., 2008)—because it only requires the list of atomic actions performed by the rule. Moreover, it can be applied both to in-place and exogenous transformations.

1.2. Running example

As a running example, let us consider an in-place transformation to animate a Domain Specific Visual Language (DSVL) for production systems. The meta-model for the language is shown to the left of

* Corresponding author. Tel.: +34 933263410; fax: +34 933568822.
E-mail addresses: rclariso@uoc.edu (R. Clarisó), jordi.cabot@icrea.cat (J. Cabot), Esther.Guerra@uam.es (E. Guerra), Juan.deLara@uam.es (J. de Lara).

http://dx.doi.org/10.1016/j.jss.2015.08.017
0164-1212/© 2015 Elsevier Inc. All rights reserved.

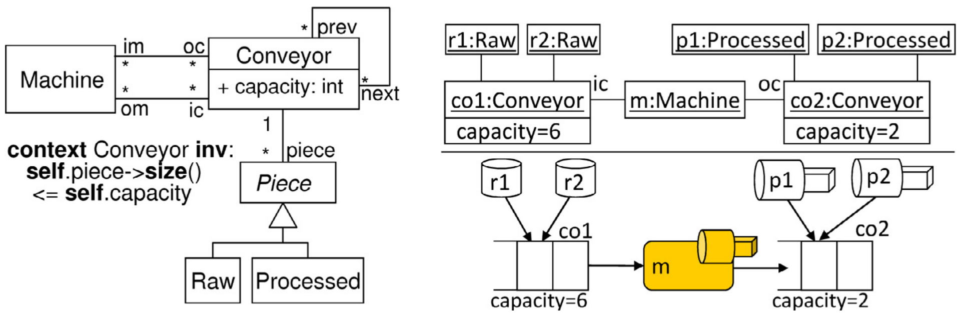

Fig. 1. Meta-model (left). A model (right).

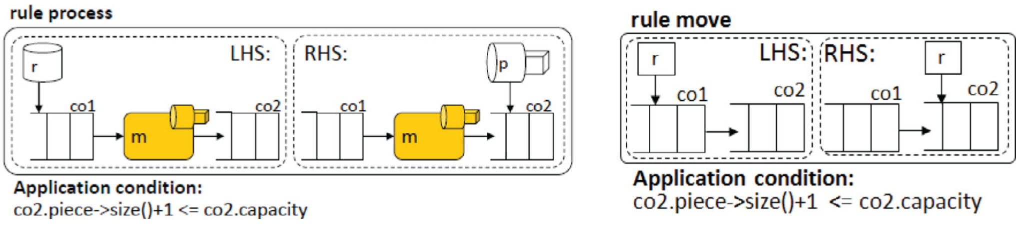

Fig. 2. Two simulation rules.

Fig. 1. It defines machines with input and output conveyors that can be interconnected. Conveyors may contain pieces, and an OCL constraint ensures that the number of pieces the conveyors actually hold does not exceed their capacity. The right of the same figure shows a model with one machine and two conveyors, in abstract (top) and concrete syntax (bottom). The left conveyor has two raw pieces, while the right one has two processed ones.

In this example, the semantics of the DSVL is defined using GT. In this approach, rules are made of two graphs, the left and the right hand sides (LHS/RHS), which encode the pre- and post-conditions for rule application. Intuitively, a rule can be applied to a model whenever an occurrence of its LHS is found in it. Then, applying the rule consists in deleting the elements of LHS−RHS, and creating those of RHS−LHS. In this way, the graphical part of the GT rule `process` on the left of Fig. 2 describes how machines behave, consuming and producing pieces: the `Raw` piece is deleted and a `Processed` one is created in the output conveyor. Rule `move` in the same figure moves pieces of any kind (we use an "abstract object" labelled `r` of type `Piece`, which can get instantiated for both types `Raw` and `Processed`) between two conveyors.

1.3. Benefits of our backwards reasoning method

To improve the usability of transformation languages, each transformation rule should be consistent with the integrity constraints of the meta-model. Otherwise, users would be forced to use some kind of integrity checking mechanism that verifies that the output model is correct after every rule execution. Hence, the guard of each rule needs to ensure that, for every possible model where the rule is applicable, the result after applying the rule satisfies all meta-model invariants (a property called *strong executability* in Cabot et al. (2010b)). For instance, in the running example of Fig. 1 the OCL integrity constraint in the meta-model forbids creating pieces in output conveyors that are already full. Below each rule, we provide an application condition that restricts the applicability of the rule to the cases where the output conveyor has enough capacity for the newly created piece.

Unfortunately, in current practice, the engineer has to encode a constraint for the same purpose *twice*: once in the meta-model, and another as the guard of each rule in the transformation to ensure that rule applications do not yield inconsistent models. Even worse, the designer has the burden of calculating an application condition that, given the rule's actions, forbids applying the rule if the execution has any chance to break some meta-model constraint. Then, this work has to be repeated for every rule in the grammar, as done for example in rule `move` of Fig. 2.

Instead, our method would derive the application condition for a rule starting from the OCL constraints of the meta-model. This presents several advantages from the point of view of the transformation developer: (i) it notably reduces his work, (ii) it facilitates grammar and meta-model evolution, as a change in the constraints of the latter has less impact on the rules, as many application conditions can be automatically derived, (iii) it eliminates the risk of not adding appropriate conditions that would cause rule applications to violate the meta-model constraints, and (iv) it eliminates the risk of adding too restrictive conditions that would forbid applying the rule, even when its application would not break any constraint (i.e. a condition that is not the weakest). In fact, the OCL condition of the rules in Fig. 2 is not the weakest, as we will show in Section 4. Moreover, this has also a clear advantage at run-time, as tools do not need to implement a roll-back mechanism if some rule application leads to an inconsistent state, and do not even need to check the meta-model constraints at each intermediate state.

Furthermore, combined with techniques for model finding (e.g. Cabot et al. 2014), our method enables the analysis of a plethora of correctness properties for the specified transformations. As we will see, several verification and testing procedures are easier to apply once the post-conditions have been advanced which facilitates a more homogeneous analysis and a better tool integration of those procedures with current modelling editors. Besides, we propose the new notion of *transformation diagnosis*, defined as the process of: (i) finding a problem in a transformation, (ii) explaining to the engineer what the problem is, and (iii) proposing some solution. Hence, for example, we are not only able to detect if a rule is not strongly executable, but can explain why (giving a model that makes the rule fail) and propose a solution (giving the weakest pre-condition that makes the rule strongly executable).

1.4. Contributions and structure of this paper

This paper continues the work in Cabot et al. (2010a), where the method was developed and applied to generate rule pre-conditions given some meta-model constraints. In this paper, we make a systematic analysis on the applicability of the method, and show techniques