Towards a Rule-level Verification Framework for Property-Preserving Graph Transformations

Hanh Nhi Tran, Christian Percebois

Towards a Rule-level Verification Framework for Property-Preserving Graph Transformations

Hanh Nhi Tran
Lab-Sticc
ENSTA-Bretagne
2 rue François Verny, 29806 Brest, France
hanh_nhi.tran@ensta-bretagne.fr

Christian Percebois
IRIT
University of Toulouse
118 Route de Narbonne, 31062 Toulouse, France
percebois@irit.fr

Abstract—**We report in this paper a method for proving that a graph transformation is property-preserving. Our approach uses a relational representation for graph grammar and a logical representation for graph properties with first-order logic formulas. The presented work consists in identifying the general conditions for a graph grammar to preserve graph properties, in particular structural properties. We aim to implement all the relevant notions of graph grammar in the Isabelle/HOL proof assistant in order to allow a (semi) automatic verification of graph transformation with a reasonable complexity. Given an input graph and a set of graph transformation rules, we can use mathematical induction strategies to verify statically if the transformation preserves a particular property of the initial graph. The main highlight of our approach is that such a verification is done without calculating the resulting graph and thus without using a transformation engine.**

Keywords- verification of graph transformations, property-preserving graph grammar, theorem proving, Isabelle/HOL.

I. INTRODUCTION

One of the most challenging issues of verification of graph transformation is to show that the transformation preserves the semantics of the source models, with respect to certain properties. To this end, two popular strategies can be used: *model checking* and *theorem-proving*. Works on model-checking are based on the exhaustive exploration of all reachable states of a graph transformation system with respect to a start graph. Consequently they are often restricted to finite systems. Generally, using the model-checking approach, to verify the correctness of a transformation system, one has to generate the potentially enormous state space that the transformation rules encode and analyze explicitly the system's behaviors.

The theorem-proving approach uses the logic inference to ensure the correctness of the transformations. Although it requires advanced proof skills, its solution is more general and can be applied to analyze infinite systems.

Most of theorem-proving-based works aim at *offline verifications* of transformation, where only the definition of the transformation itself is used in the analysis and no concrete input models are needed. The offline technique is very powerful, since all results are independent from the input models; nevertheless, it is not always practically effective and scalable due to the combinatorial explosion caused by the non-determinism of graph transformation applications.

To avoid the costly verification of offline techniques, we investigate a more pragmatic method based on the work of da Costa and Ribeiro [2][3]. Their approach gives a logical characterization of graph rewriting and associated correctness problems based on first-order (FO) definable transductions [1]. They use relational structures for encoding graph grammars and first-order formulas to define rule applications as graph grammar transformations. In this context, proving a property on graph transformation systems can be done by mathematical induction on rules' specification without necessity of a transformation engine. More precisely, this verification method reasons the definition of rules and avoids building or specifying the full state space of the system as well the explicit analysis of the system's behaviors. Although this approach requires a concrete input model for verifying a transformation, we think it is well-suited to enable a practical use of theorem proving techniques to prove graph transformations' properties.

In [2], da Costa and Ribeiro proposed an encoding of graphs and rules into relations to enable the use of logic formulas for expressing properties of a graph grammar's reachable states. Then they proposed a manual verification based on the mathematical induction technique to verify properties of systems specified in graph grammars. In [3] they propose a translation of graph grammar specifications into event-B structures, such that it is possible to use the event-B provers to demonstrate properties of a graph grammar. The verifications in these papers were illustrated with some specific structural properties. For each property to be verified, they recursively defined a function that examines the concerned transformation rule to indicate if a reachable graph has the property in question. However, in these functions, the constraints that ensure the property-preservation of a rule are implicitly checked and particularly defined for each property. The verification of graph transformations presented in [2] and [3] thus did not reveal the general characteristics of property-preserving rules.

From da Costa and Ribeiro's work, we intend to develop a framework based on inductive theorem proving for verifying invariants of graphs in graph transformation systems. An invariant property of an initial graph is preserved in a graph transformation system if the system's transformation rules preserve the property. In our

framework, as in [2][3], verifications are done by analyzing the system's transformation rules without performing the transformation to obtain the output graph. But we aim to support developers more in the verification process by providing a general reusable verification protocol that works for different properties. For this purpose, we clarified what it means for a transformation rule to preserve a property and then, at the first stage, proposed a common template to verify structural invariants. This is a novel contribution compared to the work in [2][3]. A more practical contribution of our works lies in encoding the graph transformation in the proof assistant Isabelle/HOL [15] and proposing a verification protocol based on the inductive strategy to prove structural invariant properties.

In Section 2 we present our research question and give an overview of our approach. Section 3 discusses the property preservation of transformation rules. Then we describe in Section 4 the representation and verification of graph transformation systems in our framework. Some discussions about related works are given in Section 5. Section 6 resumes our contributions and sketches out some future works.

II. VERIFICATION PROBLEM AND APPROACH

A. Problem statement

Our objective is to provide a rule-level static verification of invariant properties of systems specified in graph transformations. More specifically, given an initial graph G_0 satisfying a property P, a transformation rule r and a match m of the rule r in G_0, without carrying out the transformation, we want to determine whether the output graph created by applying the rule r on the graph G_0 satisfies the property P.

At the current stage of our work, we focus on structural properties, i.e. the system's properties that can be represented as constraints on graph structures (e.g. connectedness of graphs). We also handle properties which can be related on attribute values computed by rule applications when considering attribute graph grammars [6]. We only deal with the property preservation in each individual application of injective transformation rules, not in a sequence of various rules' applications.

B. Approach

To make sure that a rule preserves a property in the transformed graph, we have to examine how the transformation affects the initial graph without generating the output graph. To do so, first we studied the satisfaction relation between the component graphs of a rule r and the invariant property P. Then we analyzed the effects of different types of satisfaction relation on the output graph to identify under which conditions transformation rules preserve structural properties. Based on the identified patterns of property-preserving transformation rules, we proposed a general procedure to verify structural invariants by using an inductive proof strategy.

Afterwards, we encoded the relevant relational structures of graph grammars and the logic formulas that help to express properties in the proof assistant Isabelle/HOL. For validation, two following structural invariants were formalized over the encoded theories and verified by the defined procedure:

- *Property 1* (*OutGoingEdge*): Every vertex of a reachable graph has at least one outgoing edge.
- *Property 2* (*GraphPath(x,y)*): In a reachable graph, if x is a vertex having one outgoing edge and y is a vertex having one incoming edge , then there is a path between x and y.

C. Relational Approach to Graph Grammars

In the following, we summarize the approach of da Costa and Ribeiro [2][3], the base of our work, which uses a relational and logical approach for the description of graph grammars: graphs and graph morphisms are described as relational structures, i.e. tuples formed by a set and relations on this set.

A relational graph G is a tuple composed of a set, the domain of the structure, representing all vertices and edges of G and by two finite relations: the unary relation $vert_G$ defining the set of vertices of G and the ternary relation inc_G representing the incidence relation between vertices and edges of G. In the same way, a relational graph morphism g from a relational graph G to a relational graph H is defined by a set of two binary relations g_V and g_E which map respectively vertices and edges of G and H.

A relational rule $\alpha = \{\alpha_V, \alpha_E\}$ is an injective relational graph morphism from a left side relational graph L, to a right side relational graph R. The mappings of a rule are restricted to some conditions over relations between vertices and edges, in particular to check injective matches for rules.

A relational graph grammar is composed by an initial relational graph, representing the initial state of a system and a set of relational rules, describing the possible state changes that can occur in a system.

Given a relation rule α and a relational graph G, we say that the rule α is applicable in the graph G if there is a match m, which is an image of the left side graph L of the rule α in the graph G. A relational match m of the rule α in G is defined by a total relational graph morphism $m = \{m_V, m_E\}$ from L to G, such that m is injective.

The operational behavior of a graph grammar is defined in terms of rule applications. Inspired from [1], FO formulas associated to FO definable transductions are then enough to support rule applications as graph grammar transformations [2]. In this approach, applying the transduction over a source graph grammar gives another graph grammar similar to the one obtained when applying a rule to the initial state of the source grammar. In particular, rules of the source graph grammar remain unchanged. Then, given a relational rule and a relational match, the transduction maps a relational graph grammar GG to a new relational graph grammar GG' where its initial state corresponds to the result of the rule application at the given match in the initial state of GG.

In this context, proofs are established by induction over a data type reachable graph G recursively defined. This data type has one constructor for the initial graph G_0 and another one which applies a rule of the graph grammar to a given reachable graph G. The proof consists in two steps: first, the base case must show that the property holds for the initial graph G_0; second, at the inductive step, the property must hold for every rule of the graph grammar applicable to a reachable graph G, if the property initially holds for G.

III. PROPERTY PRESERVATION

We now discuss the satisfying condition for a transformation to preserve a property. In general, when an initial graph satisfies the property *P*, an applicable transformation rule preserves the property, if this rule respects the property. That is, all the elements that are *transformed* or *introduced* by the rule don't violate the property *P*; and all the *non transformed* elements of the initial graph which are necessary for satisfying *P* are kept intact. To enable a (semi) automatic verification of invariants based on analyzing transformation rules, we must formalize the above condition. For that purpose, first we analyze the possible relations between the components of a graph grammar and then represent the identified conditions formally.

A. Property-Preserving Conditions for a Transformation Rule

Given a rule *r*: $L \rightarrow R$, an initial graph G_0, a match *m*: $L \rightarrow G_0$ and a property *P* satisfied by G_0, the following points need to be clarified:

(a) What is the satisfaction relation between the rule's graphs (i.e. L and R) and the property P necessary for the rule to be property-preserving?

We remark that the property *P* may be validated on the left side graph, on the right side graph or on both side graphs of the rule *r*. The validation of *P* on the left side graph *L* is not necessary because *L* is a subgraph of the initial graph G_0 thanks to the match *m* and G_0 satisfies the property *P* by hypothesis. In contrast, the validation of *P* on the right side graph *R* is needed to ensure that new and transformed elements in *R* continue to respect the property *P*. We illustrate this statement by the examples in Fig. 1 and Fig. 2 on the preservation of the properties *OutGoingEdge* and *GraphPath*.

In these examples, the left side graph *L* does not satisfy the property to be verified. The right side graph *R* contains the transformed elements i.e. the vertices *n1*, *n2* in both figures and the new elements i.e. the vertex *n3*, the edges *(n2, n3)*, *(n3, n1)* in Fig. 1 and the edges *(n1, n3)*, *(n3, n2)* in Fig. 2.

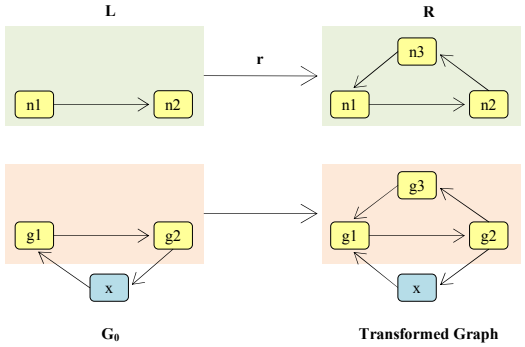

Figure 1. Preservation of the property *OutGoingEdge* after the application of the rule *r* on G_0. The *r*'s right graph *R* satisfies *OutGoingEdge*.

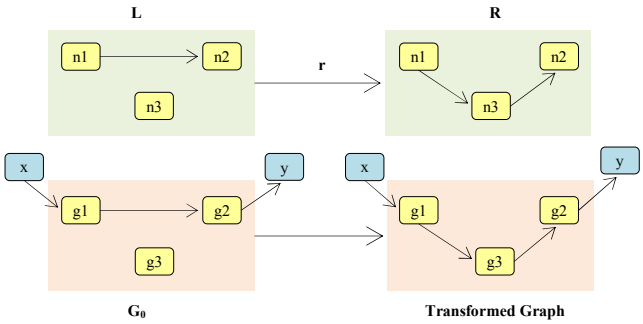

Figure 2. Preservation of the property *GraphPath(x,y)* after the application of the rule *r* on the graph G_0. The *r*'s right graph *R* satisfies *GraphPath(n1,n2)*.

For each rule in these examples, the right side graph *R* satisfies the property *P*. We can see that with such definitions, the rules in both the examples are property-preserving.

(b)Is the validation of the property P on the rule's right graph R essential for the rule to be property-preserving?

Sometimes, the property *P* is not validated on the right graph *R* of the rule *r*. The resulting graph of the rule's application, however, satisfies the property *P*.

In such cases, the elements of the initial graph G_0 who decide or affect the property *P* are kept intact in the rule's application. This can be happened if

(i) the rule *r* does not involve in the definition of the property *P*; or

(ii) the property *P* relates to an infinite number of elements of graph, or to elements that are outside the finite right graph *R*. A reduced reasoning on *R* thus may not be enough for concluding about the preservation of the property *P* after the rule's application. To illustrate the case *(ii)*, let consider the examples in Fig. 3 and Fig. 4.

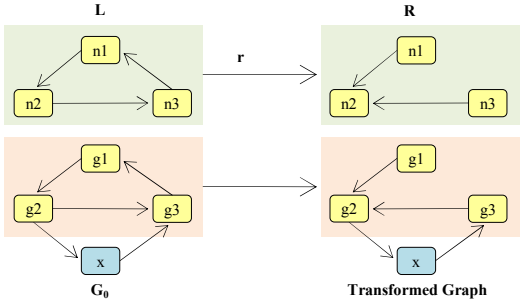

Figure 3. Preservation of the property *OutGoingEdge* after the application of the rule *r* on *G*. The *r*'s right graph *R* does not satisfy *OutGoingEdge*.

In Fig. 3, the vertex *g2* on the graph G_0 has two outgoing edges. These two edges *(g2, g3)* and *(g2, x)* participate to the satisfaction of the property *OutGoingEdge* on G_0. In the *r*'s right graph, *n2* is the only vertex violating the property *OutGoingEdge*. Luckily, the image of *n2* in G_0 is *g2* and the application of *r* on G_0 deletes the edge *(g2, g3)* but preserves the vertex *x* and its related edges, i.e. the edges *(g2, x)* and *(x, g3)*. Thus, the resulting graph satisfies the property *OutGoingEdge*.

A similar situation can be observed at Fig. 4 in the example of the preservation of the property *GraphPath(x,y)*. Here the right graph *R* deletes the path from *n1* to *n3* which is composed of two edges*(n1, n2)* and *(n2, n3)* in the left graph *L* but creates two new edges *(n1, n4)* and *(n5, n3)*. Although now the property *GraphPath(n1, n3)* is not satisfied locally in the graph *R*, after the rule's application on the graph G_0, the satisfaction of *GraphPath(x, y)* on the resulting graph is possible if there is a path between *g4* and *g5* in the initial graph G_0 as illustrated in Fig. 4. We can see that *g4* and *g5* are the images of *n4* and *n5* by the match *m* from *L* to G_0. The vertices *n4* and *n5* are preserved by the rule *r*; consequently, their images will be also preserved in the rule's application. That means the vertices *g4* and *g5* as well as the path between them are preserved in the resulting graph. We can deduce that in the right graph *R*, besides the explicit paths from *n1* to *n4* and from *n5* to *n3*, an implicit path between *n4* and *n5* is needed so that the rule *r* can preserves the property *GraphPath(n1, n3)* and thus does not violate the property *GraphPath(x, y)*. The existence of such an implicit path cannot be verified locally in the graph *R*, but in the context of the initial graph G_0.

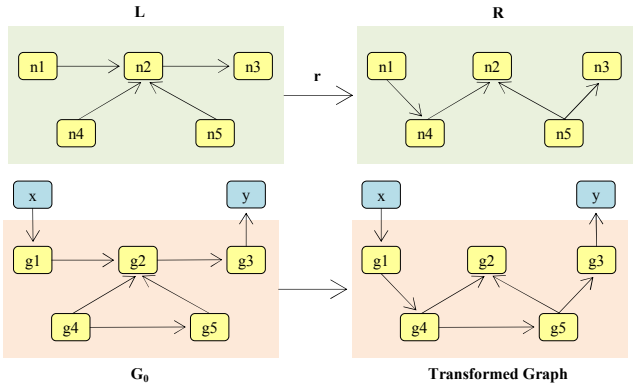

Figure 4. Preservation of the property *GraphPath(x,y)* after the application of the rule *r* on G_0. The *r*'s right graph *R* does not satisfy*GraphPath(n1,n3)*.

To conclude, the conditions for a rule *r* to preserve a property *P* can be summarized in two cases:

(1) The right graph R of the rule r satisfies the property P.

(2) The right graph R of the rule r satisfies the property P in integrating the preserved elements of the initial G_0 which decide the property P.

B. Formalizing the Property-Preserving Condition

The formalization of the above conditions can be done if we know how to describe the property *P*.

In this initial phase, we attempt to identify different classes of properties. We are mainly concerned by structural considerations about vertices and edges of the transformed graph, particularly connectedness and reachability expressed by transitive closures which require most times inductive proofs.

Table 1 presents the notation used to formalize the conditions for a transformation to preserve a graph property.

TABLE 1. USED NOTATIONS

Notation	***Meaning***
P	*P* is the property to be verified
G_0	G_0 is the initial graph
$vert_G(x)$	*x* is a vertex of the graph *G*
$inc_G(a,x,y)$	*a* is an edge of the graph *G* connecting the vertices *x* and *y*
r	*r* is a rule defining the mapping α from a left side graph *L* to a right side graph *R*
m	*m* is a match defining the image of the rule *r*'s left side graph *L* in the graph G_0
$\alpha_V(x,y)$	the vertex $y \in R$ is the image of the vertex $x \in L$ by the rule *r*'s vertices mapping
$\alpha_E(a,b)$	the edge $b \in R$ is the image of the edge $a \in L$ by the rule *r*'s edges mapping
$m_V(x,y)$	the vertex $y \in G_0$ is the image of the vertex $x \in L$ by the match *m*'s vertices mapping
$m_E(a,b)$	the edge $a \in G_0$ is the image of the edge $b \in L$ by the match *m*'s edges mapping
G	*G* is the transformed graph obtained from the application of the rule *r* on the graph G_0 with the match *m*

We now describe and study three contexts about rule transformations respectively defined on a vertex only, on edges of a vertex and between two vertices.

1) Property P is defined on the attribute(s) of a vertex

Such a property involves for example conditions on the value of a vertex's attribute or its existence.

Given a vertex *x* of a graph *G*, the properties in this class can be represented by a predicate of type $P_G(x)$. These properties are defined independently of other graph's vertices and can be verified separately and locally.

A rule *r* preserves a property *P* on a transformed graph *G* if one of the following cases is validated:

(1A) $nr \in R$ is a ***transformed vertex*** from a vertex $nl \in L$ by the rule *r*; **and** the image $ng \in G_0$ of *nl* by the match *m* satisfies *P;* **and** *nr* satisfies *P* in the context of *R*.

(1B) $nr \in R$ is a ***new vertex*** introduced by the rule *r;* **and** *nr* satisfies P^1 in the context of *R*.

More formally we denote:

$$preserving\ (G_0,r,P) \equiv \forall nr \in R\ \{$$
$$[(\exists \alpha_V(nl,nr) \wedge vert_L(nl)) \wedge (\exists m_V(nl,ng) \wedge vert_{G0}(ng) \wedge P_{G0}(ng)) \rightarrow P_R(nr)] \quad \textbf{(1A)}$$
$$\vee [\neg(\exists \alpha_V(nl,nr) \wedge vert_L(nl)) \rightarrow P_R(nr)] \quad \textbf{(1B)}\ \}$$

2) Property P is defined on the edge(s) related to a vertex

This class of property rests on the features of edges related to a vertex. The property *OutGoingEdge* presented in Section 2 belongs to this category. The specification of a property in this class can imply the existence of another vertex linked to the examined edge, but the characteristics of this second vertex are not important for the verification of the property.

[1] This situation means that the property *P* is required for every new vertex.

Given an outgoing edge e of a vertex x of a graph G, we can represent this type of property with the predicate $P_G(e(x,\perp))$ [2].

As shown in Section 3, for this class of property, in some cases, verifying locally the satisfaction of the property on the right side graph of the rule is not enough to conclude about the property-preserving feature of the rule. Therefore, a rule r preserves a property P for a transformed graph G if one of the following cases is validated:

(2A) $nr \in R$ is a ***transformed vertex*** from a vertex $nl \in L$ by the rule r; **and** the image $ng \in G_0$ of nl by the match m satisfies P; **and** nr satisfies P in the context of R.

(2B) $nr \in R$ is a ***transformed vertex*** from a vertex $nl \in L$ by the rule r, **and** the image $ng \in G_0$ of nl by the match m satisfies P; **but** nr does not satisfy P in the context of R. In this case, the image of nr in the resulting graph G may satisfy P thanks to the original edges of G_0 which remain in G. The supplementary conditions then are: there exist an edge $eg'(ng, \perp) \in G_0$ which is not an image of m (i.e. eg' is preserved in G) **and** eg' enables the satisfaction of P on G_0.

(2C) $nr \in R$ is a ***new vertex*** introduced by the rule r; **and** nr satisfies P in the context of R.

More formally we denote:

$$
\begin{aligned}
&preserving\ (G_0,r,P) \equiv \forall nr \in R\ \{\\
&[(\exists \alpha_V(nl,nr) \wedge vert_L(nl)) \wedge (\exists m_V(nl,ng) \wedge vert_{G0}(ng) \wedge inc_{G0}(eg,ng,\perp)\\
&\wedge P_{G0}(eg(ng,\perp)) \rightarrow (\exists er \wedge inc_R(er,nr,\perp) \wedge (P_R(er(nr,\perp) \quad \textbf{(2A)}\\
&\qquad \vee (\neg (\exists er \wedge inc_R(er,nr,\perp) \wedge (P_R(er(nr,\perp)\\
&\qquad \wedge (\exists eg' \wedge inc_{G0}(eg',ng,\perp))\\
&\qquad \wedge (\neg(\exists m_E(el,eg')) \wedge (P_{G0}(eg'(ng,\perp)))] \quad \textbf{(2B)}\\
&\vee\\
&[\neg(\exists \alpha_V(nl,nr) \wedge vert_L(nl)) \rightarrow (\exists er \wedge inc_R(er,nr,\perp) \wedge (P_R(er(nr,\perp)]\\
&\textbf{(2C)}\ \}
\end{aligned}
$$

3) Property P is defined on two (or more) vertices

This is a class of global properties which rely on many vertices of a graph, e.g. the existence of a path/cycle between two vertices. The property *GraphPath* presented in Section 2 belongs to this category. Given two vertices x and y of a graph G, we can represent this type of property with the predicate $P_G(x, y)$.

To verify such a property, we need to examine the two concerned vertices. As in the previous case, the local verification of the property P on the right side graph of the rule sometimes may be not enough to conclude about the property-preserving feature of the rule. The conditions for a rule r to preserve a property P for a transformed graph G are:

(3A) **For each** pair of vertices $nr1$ and $nr2 \in R$ **such that** $nr1$ and $nr2$ are the images of $nl1$ and $nl2 \in L$ by the mapping α_1 and α_2, **then if** $ng1$ and $ng2 \in G_0$ are the images of $nl1$ and $nl2$ by a match m **and if** $ng1$ and $ng2$ together satisfy P, **then if** $nr1$ *and* $nr2$ together satisfy P **then** r preserves P.

(3B) When $nr1$ and $nr2$ don't satisfy P in the context of R, their images in G can satisfy P thanks to the original preserved relations between their related vertices in $G0$. In this case, **when** there is a set of edges $eg' \in G_0$ connecting $ng1$ and $ng2$ **and when** none of eg' is an image of m, r_i preserves P **if** eg' enables $ng1$ and $ng2$ satisfies P.

(3C) As previously, when $nr1$ and $nr2$ are new vertices introduced by the rule r, **then if** $nr1$ and $nr2$ together satisfy P **then** r preserves P.

More formally:

$$
\begin{aligned}
&preserving\ (G_0,r,P) \equiv \forall nr1,nr2 \in R\ \{\\
&[(\exists \alpha 1_V(nl1,nr1) \wedge vert_L(nl1)) \wedge \exists \alpha 2_V(nl2,nr2) \wedge vert_L(nl2))\\
&\wedge (\exists m1_V(nl1,ng1) \wedge vert_{G0}(ng1)\\
&\wedge (\exists m2_V(nl2,ng2) \wedge vert_{G0}(ng2) \wedge P(ng1,ng2)\\
&\quad \rightarrow \quad (P_R(nr1,nr2) \quad \textbf{(3A)}\\
&\qquad \vee (\neg P_R(nr1,nr2) \wedge (\exists eg' \wedge setinc_{G0}(eg',ng1,ng2)\\
&\qquad\quad \wedge (\neg(\exists m_E(el,eg')) \wedge (P_{G0}(ng1,ng2)] \quad \textbf{(3B)}\\
&\vee\\
&[\neg(\exists \alpha_V(nl,nr) \wedge vert_L(nl)) \rightarrow (\exists er \wedge inc_R(er,nr,\perp) \wedge (P_R(er(nr,\perp)]\\
&\textbf{(3C)}\ \}
\end{aligned}
$$

IV. Representing and Verifying Transformations

We now present briefly the implementation of a framework that allows representing and reasoning about graph transformations. Our system is encoded in the Isabelle system [15]. The definitions presented below are extracted and simplified from the Isabelle sources.

A. Representing Graph Transformation

A graph is defined as a record consisting of a set of vertices and a set of edges (pairs of vertices):

```
record  graph =
  vertices        :: "vertex set
  edges           :: "edge set
```

A rule, a graph morphism, is defined by a left side graph (domain), a right side graph (codomain) and two sets of vertices and edges mappings. The mappings are represented as a set of relations between two vertices or two edges:

```
record  rule =
  leftGraph       :: "graph"
  rightGraph      :: "graph"
  verticesMapping :: "(vertex*vertex) set"
  edgesMapping    :: "(edge*edge) set"
```

In our system, a match is encoded as a rule.

```
types match = rule
```

We also implemented some basic formulas that help to express and verify structural properties. For example, the following functions *isVertex* and *isEdge* encode respectively the formulas $vert_G(x)$ and $inc_G(a,x,y)$ used in Section 3:

```
definition isVertex :: "vertex ⇒ graph  ⇒ bool" where
"isVertex x g == x ∈ (vertices g)"

definition isEdge ::"edge ⇒ vertex ⇒ vertex ⇒ graph ⇒bool"
where "isEdge a x y g == a∈(edges g)∧((edgeRel a)=(x, y))"
```

[2] A similar case for incoming edge is omitted to simplify the analysis.

We now have all ingredients for representing graph grammars. For example, Fig. 5 shows an initial graph *g0*, a rule *rule1* and a match *match1*:

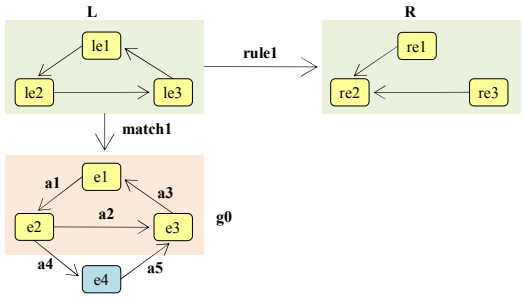

Figure 5. A graph grammar with *g0* satisfies *GraphPath(e1,e3)*.

This graph grammar is represented in our system as follows:

```
definition g0 :: "graph" where
"g0 == (|   vertices = {e1,e2,e3,e4},
            edges = {a1,a2,a3,a4,a5}|)"
```

where the edges are created by the pairs of source and target vertices, for example: `"a1 == createEdge 1 (e1,e2)"`

The graphs *left* and *right* are created in the same way. The graph morphisms *rule1* and *match1* are then defined:

```
definition rule1 :: "rule" where
"rule1 == (|    leftGraph = left, rightGraph = right,
                verticesMapping={(le1,re1),(le2,re2),(le3,re3},
                edgesMapping = {(la1,ra1),(la2,ra2)} |)"

definition match1 :: "match" where
"match1 == (|   leftGraph = left, rightGraph = g0,
                verticesMapping = {(le1,e1),(le2,e2),(le3,e3)},
                edgesMapping = {(la1,a1),(la2,a2),(la3,a3)} |)"
```

B. *Verifying Graph Transformation*

The relational and logical approach previously detailed allows the use of mathematical induction technique to verify properties of systems specified in graph grammars. We proposed a general procedure to use inductive proof in Isabelle to verify invariants of graph transformations.

Step 1: Encoding the property to be verified

In this first step, we can use the formulas encoded before in the framework to formulate the property to be checked on a graph. For instance, the properties *OutGoingEdge* is defined as follows:

```
definition hasOutGoingEdge :: "vertex ⇒ graph ⇒ bool" where
"hasOutGoingEdge x g == (∃a∈(edges g).(fst(edgeRel a)) = x)"
```

Step 2: Defining the data type of reachable graphs

The data type reachable graph of a graph grammar is defined with two constructors, one for the initial graph *g0* and another one for the operator that applies the rule *r* at match *m* to a reachable graph. In the case of the property *OutGoingEdge*, the datatype of reachable graphs is defined:

```
datatype reachableGraph = InitialGraph graph
                        |TransformedGraph rule match graph
```

Step 3: Defining a recursive function that is applied to a reachable graph to verify a property of the graph

This function follows the inductive proof strategy to verify the property: first (base case), the property is verified for the initial graph *g0* and then, at the inductive step, the property is verified for a graph obtained by the application of rule *r* to a reachable graph *g* with a match *m*, considering that the property is valid for *g*.

To verify if a graph satisfies the property *OutGoingEdge* we used the function *checkOutGoingEdge*:

```
primrec checkOutGoingEdge :: "reachableGraph ⇒ bool" where
"checkOutGoingEdge (InitialGraph g) = isOutGoingEdgeOfGraph g"|
"checkOutGoingEdge (TransformedGraph r m g) =
            (isOutGoingEdgeOfGraph g)∧(isOutGoingEdgeOfRule r m))"
```

Step 4: Defining the functions that indicates if a reachable graph has a property

The recursive function defined in the previous step needs two auxiliary functions:

(i) if a given graph satisfies a property: e.g. given a graph *g*, the function *isOutGoingEdgeOfGraph g* determines if the property *OutGoingEdge* is satisfied on *g*.

```
definition isOutGoingEdgeOfGraph :: "graph ⇒ bool" where
"isOutGoingEdgeOfGraph g  = (∀x.((isVertex x g ) →
                                    (hasOutGoingEdge x g)))"
```

(ii) if a rule r preserves a property on the transformed graph from the application of rule r with the match m on the initial graph: for example the function *isOutGoingEdgeOfRule r m* determines if the property *OutGoingEdge* is satisfied on the resulting graph of *r*'s application to the initial graph (i.e. the *m*'s right side graph).

To define such a function, we followed the general conditions identified in Section 3 to verify if a rule is property-preserving. This template can be observed in the code of the function *isOutGoingEdgeOfRule*:

```
definition isOutGoingEdgeOfRule :: "rule ⇒ match ⇒ bool" where
"isOutGoingEdgeOfRule r m = (∀nr.(isVertex nr (rightGraph r )
→ (if(isnotMappedVertex r nr) then (caseNewVertex nr r)
   else                    (caseTransformedVertex nr r m))))"
```

in this example, the functions *caseNewVertex* and *caseTransformedVertex* are defined as follows:

```
definition caseNewVertex::"vertex ⇒ rule ⇒ bool" where
"caseNewVertex nr r == (if (hasOutGoingEdge nr (rightGraph r))
                                    then True else False) "
definition caseTransformedVertex::"vertex ⇒ rule ⇒ match ⇒
bool" where
"caseTransformedVertex nr r m ==(∃nl ∈ (vertices (leftGraph r)).
   ∃ng ∈ (vertices (rightGraph m)).
   ((isVertexMapping r nl nr)∧(isVertexMapping m nl ng) ∧
   (hasOutGoingEdge ng (rightGraph m)) →
         (if (hasOutGoingEdge nr (rightGraph r)) then True
          else (testPropertyOnInitialGraph ng m))))"
```

In the code of *caseTransformedVertex*, the condition 2B (cf. Section 3) is implemented with the verification *hasOutGoingEdge nr (rightGraph r)*, i.e. the property is satisfied directly in the rule *r*'s right graph. The condition 2C to check the property in the context of the integration of the rule *r*'s right graph and the initial graph is implemented by the function *testPropertyOnInitialGraph ng m*.

Step 5: Proving that the property is preserved in a transformation

The final step is to make proofs of the property

preservation of the transformation rule *r*. For this purpose, we introduced the lemmas for the base case and the inductive case. Proofs of these lemmas are obtained interactively by using simple rewriting with the required functions and concrete data of the graph grammar. For example, we proved that the graph *g0* satisfies *OutGoingEdge*:

```
lemma proofOutGoingEdgeOfGraph : "isOutGoingEdgeOfGraph g0"
apply (simp add : isOutGoingEdgeOfGraph_def)
apply (simp add : hasOutGoingEdge_def)
apply (simp add : g0_def)
done
```

Similarly, we proved that the application of the *rule1* with the *match1* on the graph *g0* preserves the property *OutGoingEdg*e:

```
lemma proofOutGoingEdgeOfRule : "isOutGoingEdgeOfRule rule1 match1"
apply (simp add : isOutGoingEdgeOfRule_def)
apply (simp add : caseNewVertex_defcaseTransformedVertex_def)
apply (simp add : hasOutGoingEdge_def)
apply (simp add : testPropertyOnInitialGraph_def)
apply (simp add : rule1_def left_def right_def match1_def g0_def)
done
```

Now the proof of the preservation of the graph grammar in Fig. 5 for the property *OutGoingEdge* is obtained with the following theorems:

```
theorem preserving_OutGoingEdgeOfGraph :
                              " checkOutGoingEdge (InitialGraph g0)"
apply (simp add : checkOutGoingEdge_def proofOutGoingEdgeOfGraph)
done

theorem preserving_OutGoingEdgeOfRule :
"proofOutGoingEdgeOfGraph g ⟹
            checkOutGoingEdge(TransformedGraph rule1 match1 g)"
apply (simp add : checkOutGoingEdge_def proofOutGoingEdgeOfRule)
done
```

We applied the same procedure to prove the property *GraphPath* with the graph grammar presented in Fig. 4. The same templates were used for most of the codes; the changes were made for the functions representing the new property. Concretely, the function *hasGraphPath*, which determines if there is a path between the two vertices *x* and *y* of the graph *g*, replaced the function *hasOutGoingEdge*:

```
definition hasGraphPath:: "graph ⇒vertex ⇒vertex ⇒ bool"
where
"hasGraphPath g x y = ((isVertex x g) ∧ (isVertex y g)
          ∧ ((x,y) ∈ (transitiveClosure (edges g)))"
```

The functions *isGraphPathOfGraph* and *isGraphPathOfRule* substituted for the functions *isOutGoingEdgeOfGraph* and *isOutGoingEdgeOfRule* respectively.

```
definition isGraphPathOfGraph :: "graph ⇒ vertex  ⇒ vertex ⇒bool"
where
"isGraphPathOfGraph g x y = (((isVertex x g) ∧ (isVertex y g)) →
                                        (hasGraphPath g x y))"
```

In *isGraphPathOfRule*, for the special case 3B presented in Section 3, where the rule *r*'s right side graph does not satisfy the property *GraphPath*, we used the following function *rulePath* to check the property in the combined context of the initial graph and the rule:

```
definition rulePath :: "rule ⇒ match ⇒ vertex ⇒ vertex ⇒ bool"
where
"rulePath r m x y ==((∃z z'∈(vertices (rightGraph r)).
((hasGraphPath (rightGraph r) x z) ∧ (implicitGraphPath r m z z')
∧(hasGraphPath (rightGraph r) z' y)))"
```

The same proof strategy was used to prove the property *GraphPath*. This property was also checked in the context of weighted graphs with an additional verification for the weight of a path which is presented in the next section.

C. Attribute computations

The weight of a path in a weighted graph is the sum of the weights of the traversed edges. Attribute values along edges of weighted graphs can be covered in order to verify some computations done by a simple attribute graph grammar. As an example, let's consider two functions *applyComputations* and *valuePath*. The first one simulates the application of the same rule computation on a sequence of weighted edges. The second one corresponds to a recursive computation of values along weights of edges of the same attributed graph.

In this context, starting from a sequence built by *replicate m 0* for the *applyComputations* function, one can establish a proof that the n^{th} attribute computation is equal to the n^{th} term of *valuePath*. Our attribute computation adds the value of the attribute stored in a current path's node to the one given in the traversed edge. This is done repeatedly on each edge of the path, supposing the same value added at each iteration. On the other side, value of a path is recursively defined as a sum function which adds all the path attributes and returns the result. In Isabelle, this verification about attribute computations during rewriting can be emulated by the following theorem:

```
theorem path : "forall a m.Suc n < m
→(nth (applyComputations (replicate m 0) a v) n)= a + valuePath n v"
apply (induct n)
apply clarsimp apply (case_tac m) apply simp+
apply clarsimp apply (case_tac m) apply simp
apply clarsimp
done
```

The ingredients *applyComputations* and *valuePath* are defined as follows:

```
fun applyComputations :: "nat list ⇒ nat ⇒ nat ⇒ nat list"
where
"applyComputations [ ] a v = [ ]"  |
"applyComputations (x#xs) a v =
                              (x+a)#applyComputations xs (a+v) v"

fun valuePath :: "nat ⇒ nat ⇒ nat" where
"valuePath n v = (case n of
                        0 ⇒ 0| Suc m ⇒ v +(valuePath m v))"
```

As our framework for attribute computations during rewriting is based on λ-terms [16], rules for manipulating and updating attributes during the rewriting process are similar to the ones encountered in functional programming frameworks [17]. Resting on this example, it appears that the end-user can write functions simulating the behavior of such computations and so prove some properties about them.

V. Related Work

There are many significant works on formal verification of transformation as in [4][7][8][10][11][9][13]. In this section, we do not have intention to discuss all of these works, but just compare our approach with those that consider conditions under which a transformation rule is property-preserving. The common framework is when both

the initial graph satisfies the property and all the transformation rules are also property preserving. In this case, the graph transformation system satisfies the property.

In [18], OCL constraints can be assigned to model elements and rewriting steps of metamodel-based graph transformation systems. Using such constraints specifies the conditions of the matching process (preconditions) and the ones of the required result (postconditions). The main objective is to validate constraints during the model transformation and not to identify global properties of a transformation.

da Silva and Ribeiro describe [2] and code an event-B implementation [3] of graph transformation systems based on a mathematical deduction-like proof system. However, no conditions are specified about rule applicability and the authors want to evaluate and improve their approach with case studies. They also have in mind to validate a stepwise development based on the well-know refinement process in B. This process may help to analyze and reason on graph grammars.

The verification problem is also addressed by Zarrin Langari and Richard Treffer [14]. The authors can verify invariants that are expressed by the CTL modality AG and atomic propositions by adding proposition graphs to transformation rule graphs. These graphs use regular expression graphs in which edges may be labeled with the Kleene star operator. Owing to proposition graphs, the designer can compactly express feature connectivity patterns required during the transformation. The main result of the paper states a satisfaction condition theorem for a transformation rule which preserve a property *P* only if its left side graph does not weakly satisfies *P* or its right side satisfies *P*.

In a graph grammar, properties that are inherently guaranteed to be preserved are related to the graph structure. This observation we advocated is also shared by Martin Strecker [5] who notes that simply looking at the left and right sides of a transformation rule does not hold with respect to property-preserving. The source graph is then split into a finite interior region and an arbitrarily large exterior one. Reasoning about these regions can be reduced to a relation containment formula proved using Boolean set operations. However, transformations towards a Boolean satisfiability problem have an exponential complexity because the proof has to check of any combination of vertices and edges in the source graph.

VI. CONCLUSION

Our work aims to help designers by giving them the ability to prove their graph transformations' properties. In this context, mathematical induction is a simple and efficient tool for reasoning about invariants of rule applications in graph transformation systems. Without considering the rule engine, we isolate and formalize logical conditions about rules and the input graph itself which preserve some characteristics about the transformed graph.

This formalization provides a general reusable protocol that can work statically on graph grammars to verify any graph invariants if those properties can be expressed in an appropriate logic. Currently, using first-order logic, we can verify structural properties; we're considering employ more expressive logics, e.g. monadic second-order logic or temporal logics, to address more complex properties.

The concepts of our framework have been easily embodied in the Isabelle/HOL proof assistant and validated on rules adding or removing vertices and edges, but also on an example of attribute computations during the rewriting process. It is hopep that the reasonable complexity of our approach may result in a practical use of theorem proving for end-users.

It's essential to avoid a huge of possible combinations when proving a property. However, as proving graph transformations correctness entails complex traversal strategies of a graph, we claim that a challenge consists in devising new logics and proof techniques dedicated to express and to perform semi-automatically the proofs of correctness and properties of the considered rewriting systems. The emphasis here is on formalizing this logic in an interactive proof assistant, thus allowing machine-supported verification of graph transformations.

ACKNOWLEDGMENT

Part of this research has been supported by the CLIMT (*Categorical and Logical Methods in Model Transformation*) project (ANR-11-BS02-016). We would like to thank Martin Strecker for his useful discussions, especially for his help in programming with Isabelle/HOL.

REFERENCES

[1] B. Courcelle. Monadic second-order definable graph transductions: A survey. Theoretical Computer Science 126 (1994), pp. 53–75.

[2] S. A. da Costa and L. Ribeiro. Formal verification of graph grammars using mathematical induction. Electronic Notes in Theoretical Computer Science 240 (2009), pp. 43-60.

[3] L. Ribeiro, F. L. Dotti, S. A. da Costa, and F. C. Dillenburg. Towards theorem proving graph grammars using Event-B. GraMot 2010, 2010.

[4] M. Strecker. Modeling and verifying graph transformations in proof assistants. Electronic Notes in Theoretical Computer Science 203 (2007), pp.135-148.

[5] M. Strecker. Locality in reasoning about graph transformations. AGTIVE 2011, 2011.

[6] H. N. Tran, C. Percebois, A. Abou Dib, L. Féraud, and S. Soloviev. Attribute Computations in the DPoPb Graph Transformation Engine. GRABATS 2010, 2010.

[7] L. Baresi, P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. ICGT06. LNCS 4178, pp. 306–320, 2006.

[8] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. ICSE '06, pages72-81, 2006.

[9] Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, Keisuke Nakano. Graph-Transformation Verification using Monadic Second-Order Logic. PPDP 2011, 2011.

[10] Asztalos, M., Lengyel, L., & Levendovszky, T. Towards Automated, Formal Verification of Model Transformations. ICST 2010, 2010.

[11] Narayanan, a, & Karsai, G. Towards Verifying Model Transformations. *Electronic Notes in Theoretical Computer Science*, *211*, 191-200., 2008.

[12] Küster, J. M. (2006). Definition and validation of model transformations. *Software & Systems Modeling*, *5*(3), 233-259.

[13] Hermann, F., Mathias, H., Barbara, K., Specification and Verification of Model Transformations. GraMoT2010, 2010.

[14] Zarrin Langari and Richard Trefler. 2009. Application of Graph Transformation in Verification of Dynamic Systems. IFM '09, 2009

[15] T. Nipkow, L. C. Paulson and M. Wenzel. Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer, 2002.

[16] H. Barendregt, E. Barendsen. Introduction to Lambda Calculus, 1994.

[17] B. Boisvert, L. Féraud and S.Soloviev. Typed lambda-terms in categorical attributed graph transformation. AMMSE'11, 2011.

[18] László Lengyel, Tihamér Levendovszky, Hassan Charaf. Constraint Validation in Model Compilers. Journal of Object Technology, vol. 5, no. 4, Mai-June 2006, pp. 107-127