Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 240 (2009) 43–60
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Formal Verification of Graph Grammars using Mathematical Induction

Simone André da Costa[1]

Departamento de Informática
Universidade Federal de Pelotas
Pelotas, Brazil

Leila Ribeiro[2]

Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil

Abstract

Graph grammars are a formal description technique suitable for the specification of distributed and reactive systems. Model-checking of graph grammars is currently supported by various approaches. However, in many situations the use of this technique can be very time and space consuming, hindering the verification of properties of many systems. This work proposes a relational and logical approach to graph grammars that allows formal verification of systems using mathematical induction. We use relational structures to define graph grammars and first-order logic to model graph transformations. This approach allows proving properties of systems with infinite state-spaces.

Keywords: Graph grammars, mathematical induction, formal verification.

1 Introduction

Reactive systems are usually characterized by several autonomous components that run in parallel and interact with each other, for example, via messages. The verification of such systems is much more complex than sequential ones, since the interactions of independent components affect the behaviour of the whole system. To ensure that a reactive system works as expected, in addition to knowing that each component provides the required functionality, we also have to know how each component reacts to outside influences and how each component influences its outside.

[1] Email: scosta@inf.ufrgs.br
[2] Email: leila@inf.ufrgs.br

doi:10.1016/j.entcs.2009.05.044

Graph grammars are a formal language suitable for the specification of reactive systems [11,18]. The basic idea of this formalism is to model the states of a system as graphs and describe the possible state changes as rules (whose left and right-hand sides are graphs). The behaviour of the system is expressed via applications of these rules to graphs, describing the current states of the system.

One way of analyzing graph grammar models is through model-checking. In [10] a translation of a specific class of graph grammars, Object Based Graph Grammars (OBGG), to PROMELA was defined, which allows verification using the SPIN model checker [14]. [15] presents an approach to verify a timed extension of graph grammars, allowing the automatic verification of real-time systems using the UPPAAL model checker [4]. Different approaches for model-checking other kinds of graph grammars can be found in [17].

Although model checking is an important analysis method, it has as disadvantage the need to build the complete state space, which can lead to the state explosion problem. Much progress has been made to deal with this difficulty, and a lot of techniques have increased the size of the systems that could be verified [5]. Baldan and König proposed [2] approximating the behavior of (infinite-state) graph transformation systems by a chain of finite under- or over- approximations, at a specific level of accuracy of the full unfolding [1] of the system. However, as [12] emphasizes, these approaches that derive the model as approximations can result in inconclusive error reports or inconclusive verification reports.

Besides model checking, theorem proving [19] is another well-established approach used to analyze systems for desired properties. Theorem proving [6] is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. A logical description defines the system, establishing a set of axioms and inference rules. The process consists in finding a proof of the required property from the axioms or intermediary lemmas of the system. In contrast to model checking, theorem proving can deal directly with infinite state spaces and it relies on techniques such as structural induction to prove over infinite domains. The use of this technique may require interaction with a human; however, the user often gains very useful perceptions into the system or the property being proved.

Each verification technique has arguments for and against its use, but we can say that model-checking and theorem proving are very complementary. Most of the existing approaches use model checkers to analyze properties of computations. Properties about reachable states are handled, if at all possible, only in very restricted ways. Our work aims to provide a means to prove structural properties of reachable graphs using the theorem proving technique. In order to accomplish this goal, we propose a logical approach to graph grammars that allows the application of the mathematical induction technique to analyze systems with infinite state-spaces. We have defined graph grammars using relational structures and used first-order logic to model rule applications. The approach proposed here is inspired by Courcelle's research about logic and graphs [8].

Courcelle investigates in various papers [7,8,9] the representation of graphs and hypergraphs by relational structures as well as the expressiveness of its properties by logical languages. In [7] the description of graph properties and the transformation of graphs in monadic second-order logic is proposed. However, these works are not particularly interested in effectively verifying the properties of graph transformation systems (GTSs). On the other hand, other authors have investigated the analysis of GTSs based on relational logic or set theory. Baresi and Spoletini [3] explore the formal language Alloy to find instances and counterexamples for models and GTSs. In fact, with Alloy, they only analyze the system for a finite scope, whose size is user-defined. Strecker [21], aiming to verify structural properties of GTSs, proposes a formalization of graph transformations in a set-theoretic model. Nevertheless, this work does not use a logical approach and it just presents a glimpse on how to reason about graph transformations.

The rest of the paper is organized as follows. In Section 2, we introduce graph grammars according to the SPO-approach [20]. In Section 3 we present our representation of graph grammars by relational structures and in Section 4 we use first-order formulas to define rule applications as graph grammar transformations. Finally, in Section 5 we use our approach to verify properties of systems specified in graph grammars through mathematical induction. Final remarks are given in Section 6.

2 Graph Grammar

Graph-based formal description techniques often present a friendly means of carry information in a compact and understandable way, and so can be easily followed by non-specialists on formal specification methods. Graph grammars generalize Chomsky grammars from strings to graphs: it specifies a system in terms of states, described by graphs, and state changes, described by rules having graphs at the left- and right-hand sides.

Definition 2.1 [Graph, Graph morphism] A **graph** $G = (Vert_G, Edge_G, src_G, trg_G)$ consists of a set of vertices $Vert_G$, a set of edges $Edge_G$, a source and a target function $src_G, trg_G : Edge_G \to Vert_G$. We consider $Vert_G \cap Edge_G = \emptyset$. A **(partial) graph morphism** $g : G \to H$ from a graph G to a graph H is a tuple $g = (g_{Vert}, g_{Edge})$ consisting of two partial functions $g_{Vert} : Vert_G \to Vert_H$ and $g_{Edge} : Edge_G \to Edge_H$ which are weakly homomorphic, i.e., $g_{Vert} \circ src_G \geq src_H \circ g_{Edge}$ and $g_{Vert} \circ trg_G \geq trg_H \circ g_{Edge}$.[3] A morphism g is called total/ injective if both components are total/ injective, respectively.

The weak commutativity used above means that everything that is preserved (mapped) by the morphism must be compatible. The term "weak" is used because the compatibility is just required on preserved items, not on all items. A typed graph is a graph equipped with a morphism t^G to a fixed graph of types.

Definition 2.2 [Typed Graph, Typed Graph Morphism] A **typed graph** G^T is a tuple $G^T = (G, t^G, T)$, where G and T are graphs and $t^G : G \to T$ is a total graph

[3] $\geq$ is the usual relation between partial functions meaning "more defined than".

morphism called typing morphism. A **typed graph morphism** between graphs G^T and H^T with type graph T is a morphism $g : G \to H$ such that $t^G \geq t^H \circ g$ (that is, g may only map elements of the same type).

A rule specifies a possible behaviour of the system. It consists of a left-hand side, describing items that must be present in a state to enable the rule application and a right-hand side, expressing items that will be present after the rule application. We require that rules do not collapse vertices or edges (are injective) and do not delete vertices.

Definition 2.3 [Rule] Let T be a graph. A **rule** with respect to T is an injective typed graph morphism $\alpha : L^T \to R^T$ from a typed graph L^T to a typed graph R^T, such that $\alpha_{Vert} : Vert_L \to Vert_R$ is a total function on the set of vertices.

A graph grammar is composed of a *type graph*, characterizing the types of vertices and edges allowed in a system, an *initial graph*, representing the initial state of a system and a *set of rules*, describing the possible state changes that can occur in a system.

Definition 2.4 [Graph Grammar] A **(typed) graph grammar** is a tuple $GG = (T, G0, R)$, such that, T is a type graph (the type of the grammar), $G0$ is a graph typed over T (the initial graph of the grammar) and R is a set of rules with respect to type T .

Given a rule α and a state G, we say that this rule is applicable in this state if there is a match m, that is, an image of the left-hand side of the rule in the state. The operational behaviour of a graph grammar is defined in terms of rule applications.

Definition 2.5 [Match, Rule Application] Given a rule $\alpha : L^T \to R^T$ with respect to a type graph T, a **match** of a rule α in a typed graph G^T is a total typed graph morphism $m : L^T \to G^T$ which is injective on edges. A **rule application** $G^T \overset{(\alpha,m)}{\Longrightarrow} H^T$, or the application of α to a typed graph G^T at match m, generates a typed graph $H^T = (H, t^H, T)$, with $H = (Vert_H, Edge_H, src_H, trg_H)$, as follows.

Resulting graph H**.** The set of vertices and edges are defined by

$$\begin{aligned} Vert_H &= Vert_G \uplus (Vert_R - \alpha_{Vert}(Vert_L)) \\ Edge_H &= (Edge_G - m_{Edge}(Edge_L)) \uplus Edge_R \end{aligned}$$

and the source and target functions are given by

$$\begin{aligned} e \in (Edge_G - m_{Edge}(Edge_L)) &\Rightarrow src_H(e) = src_G(e), trg_H(e) = trg_G(e) \\ e \in Edge_R &\Rightarrow src_H(e) = m(src_R(e)), trg_H(e) = m(trg_R(e)) \end{aligned}$$

where $m : Vert_R \to Vert_H$ is defined by

$$\begin{aligned} m(\alpha_{Vert}(v)) &= m_{Vert}(v) \ \text{ if } \ v \in Vert_L \\ m(v) &= v \ \text{ otherwise} \end{aligned}$$

Typing morphism. The morphism $t^H = (t^H_{Vert}, t^H_{Edge})$ from H to T is specified as

$$\begin{aligned}
v \in Vert_G &\Rightarrow t^H_{Vert}(v) = t^G_{Vert}(v)\\
v \in (Vert_R - \alpha_{Vert}(Vert_L)) &\Rightarrow t^H_{Vert}(v) = t^R_{Vert}(v)\\
e \in (Edge_G - m_{Edge}(Edge_L)) &\Rightarrow t^H_{Edge}(e) = t^G_{Edge}(e)\\
e \in Edge_R &\Rightarrow t^H_{Edge}(e) = t^R_{Edge}(e)
\end{aligned}$$

Intuitively, the application of α to G at the match m first removes from G the image of the edges in L. Then, graph G is extended by adding the new nodes in R (i.e., the nodes in $Vert_R - \alpha_{Vert}(Vert_L)$) and the edges of R. This construction can be described by a pushout in a suitable category of typed graphs.

2.1 Working Example: The Token Ring Protocol

We illustrate the use of graph grammars specifying the token-ring protocol. This protocol is used to control the access of various stations to a shared transmission medium in a ring topology network [22]. According to the protocol, a special bit pattern, called the token, is transmitted from station to station in only one direction. When a station wants to send some content through the network, it waits for the token, holds it, and sends the message (data frame) to the ring. The message circulates the ring and all stations may copy its contents. When the message completes the cycle, it is received by the originating station, which then removes the message from the ring and sends the token to the next station, restarting the cycle. If only one token exists, only one station may be transmitting at a given time. Here we will model a token-ring protocol in an environment in which new stations may be added at any time.

Figure 1 illustrates the graph grammar for the example. The type graph T defines a single type of node Node, and five types of edges Msg (Message), Tok (Token), Nxt (Next), Act (Active Station) and Stb (Standby Station). Node represents a network station and an edge Msg defines a frame of data. The stations are connected by edges of type Nxt. Tok (the token) represents a special signal which enables the station to start the transmission. Every station is either an active station (Act), meaning that the station is transmitting a message on the network, or a standby station (Stb). There can be only one active station on a ring at a time. The initial graph G0 defines a ring with three nodes named N01, N02, N03. Initially the Token (Tok01) is at station N01 and no station is transmitting information on the network (all the stations have a Stb edge).

The behaviour of the protocol is modeled by the rules. A standby node with a Tok edge may retain this edge and send a message, becoming an active station (rule $\alpha1$), or pass the token to the next node (rule $\alpha2$). When a message is received by a standby node, rule $\alpha3$ can be applied and Msg is passed to the next node. If the receiving node is an active station, then rule $\alpha4$ can be applied, removing Msg from the ring and sending the token (Tok edge) to the next station. Rule $\alpha5$ is applied to insert a new station into the ring. This model has an infinite state-space and generates infinite computations.

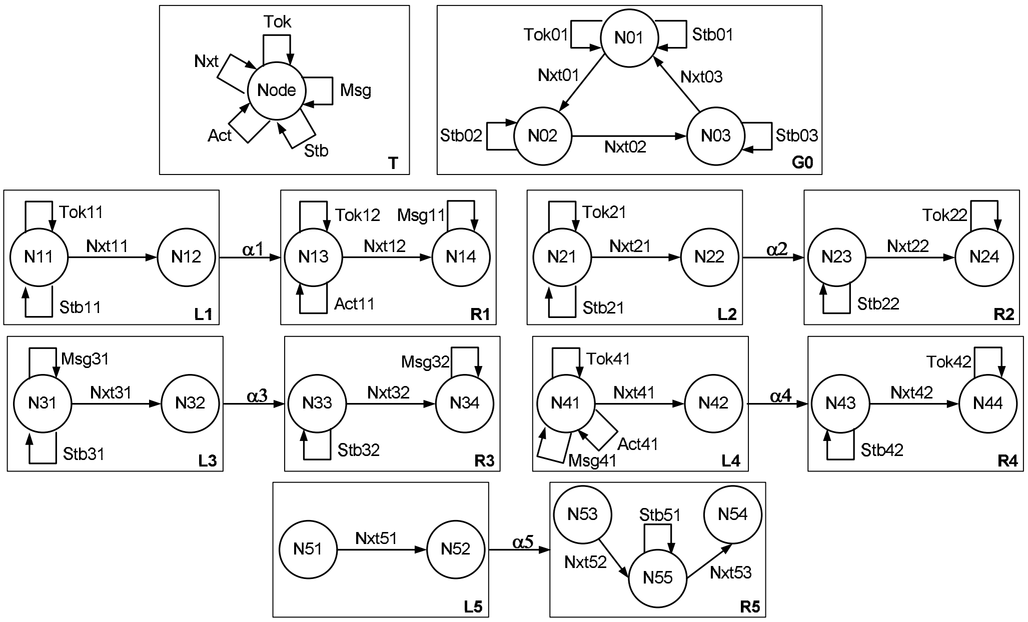

Fig. 1. Type Graph, Initial Graph and Rules

3 Representation of Graph Grammars by Relational Structures

Aiming to define a theory that allows the formulation of properties and the development of proofs for systems specified as graph grammars, we propose a representation of graph grammars by relational structures (i.e., by structures with relations only). A relational structure [8] is a tuple formed by a set and by a family of relations over this set.

Definition 3.1 [Relational Structures] Let $\mathcal{R}$ be a finite set of relation symbols, where each $R \in \mathcal{R}$ has an associated positive integer called its arity, denoted by $\rho(R)$. An **$\mathcal{R}$-structure** is a tuple $S = \langle D_S, (R_S)_{R\in\mathcal{R}}\rangle$ such that D_S is a possible empty set called the domain of S and each R_S is a $\rho(R)$-ary relation on D_S, i.e., a subset of $D_S^{\rho(R)}$. $R(d_1,\ldots,d_n)$ holds in S if and only if $(d_1,\ldots,d_n) \in R_S$, where $d_1,\ldots,d_n \in D_S$. The class of $\mathcal{R}$-structures is denoted by $STR(\mathcal{R})$.

We start by defining a relational structure to model graphs, and establishing a relational representation for graph morphisms, typed graphs and rules, which will later be used to build the relational structure associated to a graph grammar. Due to space limitations, proofs about the well-definedness of the relational representations were omitted. A relational structure representing a 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: a unary relation, $vert_G$, defining the set of vertices of G and a ternary relation inc_G representing the incidence relation between vertices and edges of G.

Definition 3.2 [Relational Structure Representing a Graph] Let $\mathcal{R}_{gr} = \{vert, inc\}$

be a set of relations, where $vert$ is unary and inc is ternary. Given a graph $G = (Vert_G, Edge_G, src_G, trg_G)$, a **relational structure representing** G is a $\mathcal{R}_{gr}$-structure $|G| = \langle D_G, (R_G)_{R \in \mathcal{R}_{gr}} \rangle$, where:

- $D_G = V_G \cup E_G$ is the union of sets of vertices and edges of G, respectively: $Vert_G = V_G$ and $Edge_G = E_G$. We also require that $V_G \cap E_G = \emptyset$;
- $vert_G \subseteq V_G$, with $x \in vert_G$ iff $x \in Vert_G$;
- $inc_G \subseteq E_G \times V_G \times V_G$, with $(x, y, z) \in inc_G$ iff $x \in Edge_G \wedge src_G(x) = y \wedge\ trg_G(x) = z$;

Example 3.3 The typed graph $G0$ depicted in Figure 1 can be defined by the relational structure $|G0| = \langle Vert_{G0} \cup Edge_{G0}, \{vert_{G0}, inc_{G0}\} \rangle$, where $vert_{G0} = \{N01, N02, N03\}$, $inc_{G0} = \{(Tok01,\ N01,\ N01), (Stb01,\ N01,\ N01), (Nxt01,\ N01,\ N02), (Stb02, N02, N02), (Nxt02,\ N02,\ N03), (Stb03, N03, N03), (Nxt03, N03, N01)\}$.

The relational representation of a graph morphism g from a graph G to a graph H is obtained through two binary relations: one to relate vertices (g_V) and other to relate edges (g_E). Since these relations just map vertices and edges names, we have to impose some restrictions to ensure that they represent a morphism. The *existence condition* states that if two vertices are related by g_V then the first one must be a vertex of G and the second one a vertex of H, and if two edges are related by g_E, then the first one must be an edge of G and the second one an edge of H. The *commutativity condition* assures that the mapping of edges preserves the mapping of source and target vertices.

Definition 3.4 [Relational Graph Morphism] Let $|G| = \langle V_G \cup E_G, \{vert_G, inc_G\} \rangle$ and $|H| = \langle V_H \cup E_H, \{vert_H, inc_H\} \rangle$ be $\mathcal{R}_{gr}$-structures representing graphs. A **relational graph morphism** g **from** $|G|$ **to** $|H|$ is defined by a set $g = \{g_V, g_E\}$ of binary relations where:

- $g_V \subseteq V_G \times V_H$ is a partial function that relates vertices of $|G|$ to vertices of $|H|$;
- $g_E \subseteq E_G \times E_H$ is a partial function that relates edges of $|G|$ to edges of $|H|$;

such that the following conditions are satisfied:

- **Existence Conditions.** $\forall x, x'\ [\ g_V(x, x')] \Rightarrow vert_G(x) \wedge vert_H(x')$; and $\forall x, x'\ [\ g_E(x, x')] \Rightarrow \exists y, y', z, z'[inc_G(x, y, z) \wedge inc_H(x', y', z')\ ]$;
- **Commutativity Condition.** $\forall x, y, z, x', y', z'$, $[g_E(x, x') \wedge inc_G(x, y, z) \wedge inc_H(x', y', z') \Rightarrow\ g_V(y, y')\ \wedge\ g_V(z, z')]$.

g is called total/ injective if relations g_V and g_E are total/ injective functions, respectively.

A typing morphism is a graph morphism that has the role of typing all elements of a graph G over a graph T. Thus, its relational definition is the same as graph morphisms, with the restriction that both relations must represent total functions.

Definition 3.5 [Relational Typing Morphism] Let $|G|$ and $|T|$ be $\mathcal{R}_{gr}$-structures representing graphs. A **relational typing morphism** t_G **from** $|G|$ **over** $|T|$ is defined by a total relational graph morphism $t_G = \{t_{G_V}, t_{G_E}\}$ from $|G|$ to $|T|$.

Example 3.6 The relational typing morphism from $|G0|$ over $|T|$, both illustrated in Figure 1, is defined by $t_{G0} = \{t_{G0_V}, t_{G0_E}\}$, with $t_{G0_V} = \{(N01, Node), (N02, Node), (N03, Node)\}$ and $t_{G0_E} = \{(Tok01, Tok), (Stb01, Stb), (Stb02, Stb), (Stb03, Stb), (Nxt01, Nxt), (Nxt02, Nxt), (Nxt03, Nxt)\}$.

The relational representation of a typed graph $G^T = (G, t^G, T)$ is defined by two $\mathcal{R}_{gr}$-structures representing G and T and by a relational typing morphism, which must satisfy a condition that guarantees that it defines the same typing morphism t^G.

Definition 3.7 [Relational Representation of a Typed Graph] Given a typed graph $G^T = (G, t^G, T)$ with $t^G = (t^G_{Vert}, t^G_{Edge})$, **a relational representation of** G^T is given by a tuple $|G^T| = \langle |G|, t_G, |T| \rangle$ where:

- $|G|$ and $|T|$ are $\mathcal{R}_{gr}$-structures representing G and T respectively;
- $t_G = \{t_{G_V}, t_{G_E}\}$ is a relational typing morphism from $|G|$ over $|T|$ that satisfies the conditions: $(x, y) \in t_{G_V}$ iff $t^G_{Vert}(x) = y$ and $(x, y) \in t_{G_E}$ iff $t^G_{Edge}(x) = y$.

A relational graph morphism is also the basis of the relational definition of a rule from a graph L to a graph R. Since both graphs mapped by a rule are typed over the same graph T, a *compatibility condition* assures that the mappings of vertices and edges preserve types. Besides, rules must not collapse vertices or edges, nor delete vertices.

Definition 3.8 [Relational Rule] Let $|L|$, $|R|$ and $|T|$ be $\mathcal{R}_{gr}$-structures representing graphs and $t_L = \{t_{L_V}, t_{L_E}\}$ and $t_R = \{t_{R_V}, t_{R_E}\}$ be relational typing morphisms from $|L|$ and $|R|$ over $|T|$, respectively. A **relational rule** $|\alpha|$ **from** $|L|$ **to** $|R|$ is defined by an injective relational graph morphism $|\alpha| = \{\alpha_V, \alpha_E\}$, such that the following conditions are satisfied:

- α_V is total;
- **Compatibility Condition.** The mapping of vertices and edges preserves types:
$$\forall x, x', y\ [\alpha_V(x, x') \wedge t_{L_V}(x, y) \Rightarrow t_{R_V}(x', y)],$$
$$\forall x, x', y\ [\alpha_E(x, x') \wedge t_{L_E}(x, y) \Rightarrow t_{R_E}(x', y)].$$

Example 3.9 The relational rule $\alpha 1$ illustrated in Figure 1 is defined by $|\alpha_1| = \{\alpha_{1_V}, \alpha_{1_E}\}$, where $\alpha_{1_V} = \{(N11, N13), (N12, N14)\}$ and $\alpha_{1_E} = \{(Tok11, Tok12), (Nxt11, Nxt12)\}$. The relational typing morphisms from $L1$ and $R1$ over T are respectively given by $t_{L1_V} = \{(N11, Node), (N12, Node)\}$, $t_{L1_E} = \{(Tok11, Tok), (Stb11, Stb), (Nxt11, Nxt)\}$ and $t_{R1_V} = \{(N13, Node), (N14, Node)\}$, $t_{R1_E} = \{(Tok12, Tok), (Act11, Act), (Nxt12, Nxt), (Msg11, Msg)\}$.

Given a rule $\alpha : L^T \to R^T$, its relational representation is given by the relational representation of typed graphs L^T and R^T, together with a relational rule which must define the same morphism given.

Definition 3.10 [Relational Representation of a Rule] Given a rule $\alpha : L^T \rightarrow R^T$, $\alpha = (\alpha_{Vert}, \alpha_{Edge})$, **a relational representation of** α is given by a tuple $\langle |L^T|, |\alpha|, |R^T| \rangle$ where:

- $|L^T| = \langle |L|, t_L, |T| \rangle$ and $|R^T| = \langle |R|, t_R, |T| \rangle$ are relational representations of typed graphs L^T and R^T, respectively;
- $|\alpha| = \{\alpha_V, \alpha_E\}$ is a relational rule from $|L|$ to $|R|$ that satisfies the following conditions: $(x, y) \in \alpha_V$ iff $\alpha_{Vert}(x) = y$ and $(x, y) \in \alpha_E$ iff $\alpha_{Edge}(x) = y$.

Given a graph grammar $GG = (T, G0, R)$, we define a relational structure $|GG|$ associated to it as a tuple composed of a set and a collection of relations. The set describes the domain of the structure. The relations define the type graph, the initial graph and the rules. The type graph is defined by relations of a $\mathcal{R}_{gr}$-structure representing T. The initial graph $G0$, the left- and right-hand sides of rules are specified by relations of $\mathcal{R}_{gr}$-structures representing graphs, which are typed over T by relational typing morphisms. Relational rules map the graphs of left-hand side and right-hand side of rules.

Definition 3.11 [Relational Structure Associated to a Graph Grammar] Let $\mathcal{R}_{GG} = \{vert_T,\ inc_T,\ vert_{G0},\ inc_{G0},\ t_{GO_V},\ t_{G0_E}, (vert_{Li},\ inc_{Li},\ t_{Li_V},\ t_{Li_E}, vert_{Ri},\ inc_{Ri},\ t_{Ri_V},\ t_{Ri_E},\ \alpha_{i_V},\ \alpha_{i_E})_{i \in \{1,\ldots,n\}}\}$ be a set of relation symbols. Given a graph grammar $GG = (T, G0, R)$ where $|R| = n$, **the $\mathcal{R}_{\mathbf{GG}}$-structure associated to** GG, denoted by $|GG|$, is the tuple $\langle D_{GG}, (r_{GG})_{r \in \mathcal{R}_{GG}} \rangle$[4] where

- $D_{GG} = V_{GG} \cup E_{GG}$ is the set of vertices and edges of the graph grammar, where: $V_{GG} \cap E_{GG} = \emptyset$, $V_{GG} = V_T \cup V_{G0} \cup (V_{Li} \cup V_{Ri})_{i \in \{1,\ldots,n\}}$ and $E_{GG} = E_T \cup E_{G0} \cup (E_{Li} \cup E_{Ri})_{i \in \{1,\ldots,n\}}$.
- $vert_T$ and inc_T model the **type graph**. They are the relations of a $\mathcal{R}_{gr}$-structure $|T| = \langle V_T \cup E_T, \{vert_T, inc_T\} \rangle$ representing graph T.
- $vert_{G0}$, inc_{G0}, t_{G0_V} and t_{G0_E} model the **initial graph typed over** T, i.e., they are the relations that compose the relational representation of $G0^T$.
- Each collection $(vert_{Li},\ inc_{Li},\ t_{Li_V},\ t_{Li_E},\ vert_{Ri}, inc_{Ri},\ t_{Ri_V},\ t_{Ri_E},\ \alpha_{i_V},\ \alpha_{i_E})$ defines a **rule**:
  - · $vert_{Li}$, inc_{Li}, t_{Li_V} and t_{Li_E} model the **left-hand side** of the rule, i. e., they are the relations of the relational representation of Li^T.
  - · $vert_{Ri}, inc_{Ri}$, t_{Ri_V} and t_{Ri_E} model the **right-hand side** of the rule, i. e., they are the relations of the relational representation of Ri^T.
  - · α_{i_V} and α_{i_E} are relations of the set $|\alpha_i|$ which defines a **relational rule** from $|Li|$ to $|Ri|$, such that the tuple $\langle |Li^T|, |\alpha_i|, |Ri^T| \rangle$ is a relational representation of rule $\alpha_i : Li^T \rightarrow Ri^T$.

[4] In order to simplify the reading we omit the subscript GG in relations.

4 Rule Applications as First-Order Definable Transductions

In this section, inspired in the definition of monadic second-order definable transduction, introduced in [8], we show how to define rule applications as graph grammar transformations. This approach will allow a graph grammar theory to be defined, which will be later used to verify properties of distributed and reactive systems.

A monadic second-order definable transduction [8] replaces for graphs the notion of finite automaton used for transformations of words or trees. It is defined through a tuple $(\varphi, \psi, (\theta_q)_{q\in\mathcal{Q}})$ of monadic second-order formulas [13] that specifies a $\mathcal{Q}$-structure T based on an $\mathcal{R}$-structure S. The first formula of the tuple, φ, establishes a condition to be satisfied in order to make the transduction possible. The following formula ψ define the domain of the relation T. Finally, for each relation $q \in \mathcal{Q}$, a formula θ defines the elements of the T domain that belong to the relation. In the original definition, it is possible to make k copies of the original structure S before redefining the relations q, to obtain the new structure T. Next, we propound the definition of first-order definable transductions (via first-order formulas) without copies of the original structure, which is enough to represent rule applications as graph-grammar transformations.

Definition 4.1 [First-Order Definable Transduction] Let $\mathcal{R}$ and $\mathcal{Q}$ be two finite ranked sets of relation symbols. Let $\mathcal{W}$ be a finite set of set variables (parameters) and $FO(\mathcal{R},\mathcal{W})$ be the set of first-order formulas over $\mathcal{R}$, with free variables in $\mathcal{W}$. A $(\mathcal{Q},\mathcal{R})$**-definition scheme** is a tuple $\Delta = (\varphi, \psi, (\theta_q)_{q\in\mathcal{Q}})$, where $\varphi \in FO(\mathcal{R},\mathcal{W})$, $\psi \in FO(\mathcal{R},\mathcal{W} \cup \{x_1\})$ and $\theta_q \in FO(\mathcal{R},\mathcal{W} \cup \{x_1,\dots,x_{\rho(q)}\})$.

These formulas are intended to define a structure T in $STR(\mathcal{Q})$ from a structure S in $STR(\mathcal{R})$ in the following way: let $S \in STR(\mathcal{R})$ and γ be a $\mathcal{W}$-assignment in S, **a** $\mathcal{Q}$**-structure** T with domain $D_T \subseteq D_S$ **is defined in** (S,γ) **by** Δ if:

(i) $(S,\gamma) \models \varphi$. Formula φ establishes a condition to be fulfilled so that the translation is possible. I.e., T is defined only if φ holds true in S for some γ.

(ii) $D_T = \{d \in D_S \mid (S,\gamma,d) \models \psi\}$. Assuming that (i) is satisfied, formula ψ defines the domain of T as the set of elements in the S domain that satisfy ψ for γ.

(iii) for each $q \in \mathcal{Q}$, $q_T = \{(d_1,\dots,d_t) \in D_T^t \mid (S,\gamma,d_1,\dots,d_t) \models \theta_q\}$, where $t = \rho(q)$. Formulas θ_q define the relation q_T for each $q \in \mathcal{Q}$.

Since T is associated in a unique way with S,γ and Δ whenever it is defined (whenever $(S,\gamma) \models \varphi$) we can use the functional notation $def_\Delta(S,\gamma)$ for T. A **transduction defined by** Δ is the relation $def_\Delta := \{(S,T) \mid T = def_\Delta(S,\gamma)$ for some $\mathcal{W}$-assignment γ in $S\} \subseteq STR(\mathcal{R}) \times STR(\mathcal{Q})$. $f \subseteq STR(\mathcal{R}) \times STR(\mathcal{Q})$ **is a FO-definable transduction**, if it is equal to def_Δ, for some $(\mathcal{Q},\mathcal{R})$-definition scheme Δ. In the case where $\mathcal{W} = \emptyset$ we say that f is definable without parameters.

A rule application may be described by a FO-definable transduction on relational structures associated to graph grammars. The result of the transduction over a graph grammar is another graph grammar whose initial state corresponds to the result of the application of a rule α_i at a match m to the initial state of the original grammar. The other components of the grammar remain unchanged (i.e., the resulting grammar has the same type graph and rules of the original one). In order to define rule application as a FO-definable transduction, we first introduce the relational representation of a match. A relational match of a rule α (from L to R) in a graph G is a total relational graph morphism (from L to G) which is injective on edges and preserves types (satisfies the match compatibility condition). The relational representation of a match $m : L^T \to G^T$ is then defined by two relational structures representing typed graphs L^T and G^T together with a relational match that must define the same morphism m given.

Definition 4.2 [Relational Match] Let $|\alpha|$ from $|L|$ to $|R|$ be a relational rule, where $t_L = \{t_{L_V}, t_{L_E}\}$ and $t_R = \{t_{R_V}, t_{R_E}\}$ are the relational typing morphisms from $|L|$ and $|R|$ over $|T|$, respectively. Let $|G|$ be an $\mathcal{R}_{gr}$-structure representing a graph G typed over $|T|$ by the relational typing morphism $t_G = \{t_{G_V}, t_{G_E}\}$. A **relational match** $|m|$ **of a rule** $|\alpha|$ **in** $|G|$ is defined by a total relational graph morphism $|m| = \{m_V, m_E\}$ from $|L|$ to $|G|$, such that the following conditions are satisfied:

- m_E is injective;
- **Match Compatibility Condition.** The mapping of vertices and edges preserve types: $\forall x, x', y\ [m_V(x, x') \wedge t_{L_V}(x, y) \Rightarrow t_{G_V}(x', y)]$,
  $\forall x, x', y\ [m_E(x, x') \wedge t_{L_E}(x, y) \Rightarrow t_{G_E}(x', y)]$.

Definition 4.3 [Relational Representation of a Match] Given a match $m : L^T \to G^T$, $m = (m_{Vert}, m_{Edge})$, **a relational representation of** m is given by a tuple $\langle |L^T|, |m|, |G^T| \rangle$ where:

- $|L^T| = \langle |L|, t_L, |T| \rangle$ and $|G^T| = \langle |G|, t_G, |T| \rangle$ are relational representations of typed graphs L^T and G^T, respectively;
- $|m| = \{m_V, m_E\}$ is a relational match from $|L|$ to $|G|$ that satisfies the following conditions: $(x, y) \in m_V$ iff $m_{Vert}(x) = y$ and $(x, y) \in m_E$ iff $m_{Edge}(x) = y$.

Now, a rule application is represented by a definable transduction (i.e., by a tuple of first-order formulas) that defines a $\mathcal{R}_{GG}$-structure $|GG|'$ (i.e., a graph grammar) based on another $\mathcal{R}_{GG}$-structure $|GG|$. Before applying the transduction, we must first fix a relational rule $|\alpha_i|$ of $|GG|$ and a relational match $|m|$ of $|\alpha i|$ in $|G0|$ (initial graph of $|GG|$). Then, the $\mathcal{R}_{GG}$-definition scheme $\Delta = (\varphi, \psi, (\theta_q)_{q \in \mathcal{R}_{GG}})$ defines the relational structure $|GG|'$ from $|GG|$, which corresponds to the same grammar, excepted that $|G0|'$ (initial state of $|GG|'$) represents the result of the application of $|\alpha i|$ at match $|m|$ in $|G0|$. In Δ, φ ensures that $|m|$ effectively defines a match, ψ defines the domain of the resulting grammar (the same of original grammar) and each formula θ_q, $q \in \mathcal{R}_{GG}$, defines the elements that will be present in relations $q_{GG'}$, $q \in \mathcal{R}_{GG}$ of the resulting grammar. In fact, the collection (θ_q) defines the structure associated to graph grammar $|GG|'$. Since the type graph and the rules remain unchanged, the formulas that define these components are constructed in the obvious way (they are defined by relations of the original grammar). Formulas $\theta_{vert_{G0}}$, $\theta_{inc_{G0}}$, $\theta_{t_{G0_V}}$, $\theta_{t_{G0_E}}$ that define the resulting graph of the rule application are specified according to Definition 2.5. The following table presents the intuitive meaning and the equivalent notation of the formulas used in θ specifications.

Formula	**Intuitive Meaning**	**Equivalent Notation**
$vert_{G_{GG}}(x)$	x is a vertex of graph G in GG.	-
$inc_{G_{GG}}(x,y,z)$	x is an edge of graph G with source vertex y and target vertex z in GG.	-
$t_{G_{V\,GG}}(x,y)$	x is a vertex of graph G of type y in GG.	-
$t_{G_{E\,GG}}(x,y)$	x is an edge of graph G of type y in GG.	-
$\alpha_{i_{V\,GG}}(x,y)$	x is a vertex of graph Li mapped to vertex y of Ri by rule α_i in GG.	-
$\alpha_{i_{E\,GG}}(x,y)$	x is an edge of graph Li mapped to edge y of Ri by rule α_i in GG.	-
$vert_{Ri_{GG}}(x) \wedge \nexists y\Big(\alpha_{i_{V\,GG}}(y,x)\Big)$	x is a vertex of graph Ri that is not image of the rule α_i in GG.	$nvert_{Ri_{GG}}(x)$
$inc_{G0_{GG}}(x,y,z) \wedge \nexists w\Big(m_E(w,x)\Big)$	x is an edge of graph $G0$ with source y and target z in GG that is not image of the match.	$ninc_{G0_{GG}}(x,y,z)$
$\exists r,s\Big[inc_{Ri_{GG}}(x,r,s) \wedge \overline{n}(r,y) \wedge \wedge \overline{n}(s,z)\Big]$	x is an edge of graph Ri with source and target vertices given by binary relation $\overline{n}$.	$ninc_{Ri_{GG}}(x,y,z)$
$\begin{cases} \exists v \Big(\alpha_{i_{V\,GG}}(v,r) \wedge \wedge m_V(v,y)\Big) & \text{if } r \neq y \\ \nexists v\ \alpha_{i_{V\,GG}}(v,r) & \text{if } r = y \end{cases}$	Vertex r is related to some different vertex y if it is image of the rule applied to some vertex v. In this case r is related with the image of the match applied to v. Vertex r is related to itself if it is not image of the rule.	$\overline{n}(r,y)$
$vert_{G0_{GG}}(x) \wedge t_{G0_{V\,GG}}(x,t)$	x is a vertex of graph $G0$ of type t in GG.	$nvert_{G0_{GG}}(x,t)$
$\exists y,z\Big(inc_{G0_{GG}}(x,y,z)\Big) \wedge \wedge \nexists w\Big(m_E(w,x)\Big) \wedge t_{G0_{E\,GG}}(x,t)$	x is an edge of graph $G0$ of type t in GG that is not image of the match.	$nt_{G0_{E\,GG}}(x,t)$

Definition 4.4 [Rule Application as FO-Definable Transduction] Let $|GG|$ be a relational structure associated to a graph grammar with an additional requirement: the sets of edges and vertices of graphs $|T|$, $|G0|$, $|Li|$ and $|Ri|$ are disjoint. Given a relational rule $|\alpha_i| = \{\alpha_{i_V}, \alpha_{i_E}\}$ from $|Li|$ to $|Ri|$ and a relational match $|m| = \{m_V, m_E\}$ of $|\alpha_i|$ in $|G0|$, the **transduction that maps a graph grammar** $|GG|$ **to a graph grammar** $|GG|'$, where $|G0|'$ (initial state of $|GG|'$) corresponds to the result of the application of rule $|\alpha_i|$ at match $|m| = \{m_V, m_E\}$ in $|G0|$ (initial state of $|GG|$), **is defined by** $\Delta = (\varphi, \psi, (\theta_q)_{q \in \mathcal{R}_{GG}})$, with $\mathcal{W} = \emptyset$, where:

φ expresses that $|m| = \{m_V, m_E\}$ defines a relational match of $|\alpha_i|$ in $|G0|$.

ψ is the Boolean constant true (same domain).

θ_{vert_T}, θ_{inc_T} are, respectively, the formulas $vert_{T_{GG}}(x)$ and $inc_{T_{GG}}(x,y,z)$ (same type graph).

$\theta_{vert_{G0}}$ is the formula $vert_{G0_{GG}}(x) \ \vee \ nvert_{Ri_{GG}}(x)$ (see next table).

$\theta_{inc_{G0}}(x,y,z)$ is the formula $ninc_{G0_{GG}}(x,y,z) \ \vee \ ninc_{Ri_{GG}}(x,y,z)$.

$\theta_{t_{G0_V}}(x,t)$ is the formula $nvert_{G0_{GG}}(x,t) \ \vee \ \Big[nvert_{Ri_{GG}}(x) \ \wedge \ t_{Ri_{V\,GG}}(x,t)\Big]$.

$\theta_{tG0_E}(x,t)$ is the formula $nt_{G0_{EGG}}(x,t) \;\vee\; t_{Ri_{EGG}}(x,t)$.

$\theta_{vert_{Li}}$, $\theta_{inc_{Li}}$, $\theta_{t_{Li_V}}$, $\theta_{t_{Li_E}}$, $\theta_{vert_{Ri}}$, $\theta_{inc_{Ri}}$, $\theta_{t_{Ri_V}}$, $\theta_{t_{Ri_E}}$, $\theta_{\alpha_{i_V}}$, $\theta_{\alpha_{i_E}}$ are respectively the formulas $vert_{Li_{GG}}(x)$, $inc_{Li_{GG}}(x,y,z)$, $t_{Li_{V\,GG}}(x,y)$, $t_{Li_{E\,GG}}(x,y)$, $vert_{Ri_{GG}}(x)$, $inc_{Ri_{GG}}(x,y,z)$, $t_{Ri_{V\,GG}}(x,y)$, $t_{Ri_{E\,GG}}(x,y)$, $\alpha_{i_{V\,GG}}(x,y)$ and $\alpha_{i_{E\,GG}}(x,y)$, for $i = 1\,..\,n$ (same rules).

Proposition 4.5 *The rule application as a FO-definable transduction is well-defined.*

Proof (Sketch) Let $|GG|'$ be the result of the transduction applied to graph grammar $|GG|$ corresponding to the application of relational rule $|\alpha_i|$ at relational match $|m|$. Considering that the given rule $|\alpha_i|$ and the given match $|m|$ are the relational representations of $\alpha_i : Li^T \to Ri^T$ and $m : Li^T \to G0^T$, respectively, and considering $H^T = (H, t^H, T)$ to be the typed graph obtained through the application of α_i to graph $G0^T$ at match m (according to Definition 2.5) we have to show that:

(i) $vert_{T_{GG'}}$ and $inc_{T_{GG'}}$ are the relations of a $\mathcal{R}_{gr}$-structure $|T|' = \langle V'_T \cup E'_T, \{vert_{T_{GG'}}, inc_{T_{GG'}}\}\rangle$ representing graph $T = (Vert_T, Edge_T, src_T, trg_T)$.

(ii) $vert_{G0_{GG'}}$ and $inc_{G0_{GG'}}$ are the relations of a $\mathcal{R}_{gr}$-structure $|G0|' = \langle V'_{G0} \cup E'_{G0}, \{vert_{G0_{GG'}}, inc_{G0_{GG'}}\}\rangle$ representing graph $H = (Vert_H, Edge_H, src_H, trg_H)$.

(iii) $t_{GO_{V\,GG'}}$ and $t_{G0_{E\,GG'}}$ are from the set $t_{G0_{GG'}}$ such that the tuple $\langle |G0|', t_{G0_{GG'}}, |T|'\rangle$ is a relational representation of the typed graph $H^T = (H, t^H, T)$.

The graph grammar that results from the application of rule $|\alpha_i|$ at match $|m|$ in $|G0|$ ($|GG|$ initial state) has its initial graph defined by relations $vert_{T_{GG'}}$, $inc_{T_{GG'}}$, $vert_{G0_{GG'}}$, $inc_{G0_{GG'}}$, $t_{G0_{V\,GG'}}$ and $t_{G0_{E\,GG'}}$, whose elements are those of $|GG|'$ domain (same $|GG|$ domain) that satisfy, respectively, the formulas θ_{vert_T}, θ_{inc_T}, $\theta_{vert_{G0}}$, $\theta_{inc_{G0}}$, $\theta_{t_{G0_V}}$ and $\theta_{t_{G0_E}}$. Such θ formulas are defined by $|GG|$ relations, which compose, according to Definition 3.11, relational representations of graphs, typed graphs or rules. Following Definitions 3.2, 3.7 and 3.10 these relations will define the respective graphs, typed graphs and rules of the original grammar. We can also notice that the formulas θ that define the resulting graph are specified according to graph H^T described in Definition 2.5. Consequently, the elements that satisfy the formulas will define the relational representation of H^T. $\square$

5 Verifying Properties

The logical approach previously detailed allows the use of mathematical induction technique to verify properties of systems specified in graph grammars. Inspired by the standard procedure of Isabelle [16], we lay the foundation for future creation of a graph grammar theory that must define a data type named reachable graph and a standard library, which may be used to formulate properties and develop proofs.

The **data type reachable graph** (*reach_gr*) of a graph grammar may be defined with two constructors, one for the initial graph $G0$ and another one for the operator $ap(\alpha i, m)$ that applies the rule αi at match m to a reachable graph (obtaining a graph $G0'$ according to the transduction defined in Section 4). The **standard library** must provide a collection of (recursive) functions that can be used to enunciate and prove desirable properties. For instance, we define two functions: one to determine the types of edges of a reachable graph and another to indicate if a reachable graph has a ring topology. Let $|GG|$ be the relational structure associated to a graph grammar.[5]

Types of Edges of a Reachable Graph. The types of edges of a reachable graph are recursively defined by:

$$tip_E\ G0 = \{(x,t) \mid t_{G0_E}(x,t)\} \tag{1}$$
$$tip_E\ ap(\alpha i, m)\ g = \{(x,t) \mid t_{Ri_E}(x,t) \vee [(x,t) \in tip_E\ g \wedge \nexists w\ m_{E_{\alpha i}}(w,x)]\} \tag{2}$$

That is, if we consider the initial graph (1), typing is given by the relation t_{G0_E} of the relational structure. If we consider a graph obtained from applying rule αi at match $m = \{m_{V_{\alpha i}}, m_{E_{\alpha i}}\}$ to graph g (2), the type of an edge is either the type of edges of right-hand side of the rule or a type of edge of graph g (in the latter case, the edge can not be image of the match).

Ring Topology in a Reachable Graph. Initially, we define the transitive closure of edges of type t in a graph G, denoted by $TC^t_{inc_G}$, by:

$$\forall a,x,y,z\ [inc_G(a,x,y) \wedge t_{G_E}(a,t) \rightarrow (x,y) \in TC^t_{inc_G}] \wedge \\ [(x,y) \in TC^t_{inc_G} \wedge (y,z) \in TC^t_{inc_G} \rightarrow (x,z) \in TC^t_{inc_G}]$$

Then, we recursively define the function that indicates if a reachable graph has a ring topology of edges of type t:

$$\text{Ring}_t\ G0 \equiv \forall x\ [vert_{G0}(x) \rightarrow (x,x) \in TC^t_{inc_{G0}}]\ \wedge \tag{1'}$$
$$\wedge \forall a,b,x,y,z\ [inc_{G0}(a,x,y) \wedge t_{G0_E}(a,t) \wedge inc_{G0}(b,x,z) \wedge \\ \wedge\, t_{G0_E}(b,t) \rightarrow a = b]\ \wedge \tag{2'}$$
$$\wedge \forall x,z\ [vert_{G0}(x) \wedge vert_{G0}(z) \rightarrow (x,z) \in TC^t_{inc_{G0}}] \tag{3'}$$

$$\text{Ring}_t\ ap(\alpha i, m)\ g \equiv \text{Ring}_t\ g\ \wedge \tag{4'}$$
$$\wedge \forall a,x,y,z,w\ [inc_{Li}(a,x,y) \wedge t_{Li_E}(a,t) \wedge \alpha_{i_V}(x,z) \wedge \\ \wedge\, \alpha_{i_V}(y,w) \rightarrow (z,w) \in TC^t_{inc_{Ri}}]\ \wedge \tag{5'}$$
$$\wedge \forall a,b,x,y,z\ [inc_{Ri}(a,x,y) \wedge t_{Ri_E}(a,t) \wedge inc_{Ri}(b,x,z) \wedge \\ \wedge\, t_{Ri_E}(b,t) \rightarrow a = b] \tag{6'}$$

That is, $G0$ has a ring topology if the following conditions are satisfied:

(1') There is a cycle, i.e., every vertex of $G0$ has a path with origin and destination in itself;

[5] Again, in what follows, we omit the subscript GG in relations, assuming that it is clear from context which grammar is under consideration.

(2') There is no bifurcation of edges of type t in $G0$, i.e., if there are two edges of type t with origin at the same vertex, these edges are equal. This property guarantees that the paths of edges of type t in $G0$ are unique;

(3') The graph is connected, i.e., from every vertex in $G0$ there is a path to all other vertices.

And, to have a graph with a ring topology resulting from the application of a rule $\alpha_i = \{\alpha_{i_V}, \alpha_{i_E}\}$ to a reachable graph, it must be guaranteed that:

(4') The reachable graph has a ring structure;

(5') For every edge a of type t going from x to y in Li there is a corresponding path in Ri starting at the image α_{i_V} of x and ending at the image α_{i_V} of y;

(6') There is no bifurcation of edges of type t in Ri. This guarantees that the paths of edges of type t in Ri are unique.

Other functions could also be included in the library, such as, functions to define types of vertices of a reachable graph, cardinality of edges, cardinality of vertices and many others. Having established the theory, we describe the **proof strategy** used to demonstrate properties for a system specified in graph grammar. First, we must define the relational structure associated to the grammar (according to Definition 3.11). The relations of this structure define axioms that are used in the proofs. For example, considering $|GG| = \langle D_{GG}, (R)_{R\in\mathcal{R}_{GG}}\rangle$ the relational structure associated to the grammar, we have $R(x_1, \ldots, x_n) \equiv true$ iff $(x_1, \ldots, x_n) \in R$. Then we may state a goal to be proven using logic formulas. Properties about reachable states may be proven by induction, since this data type is recursively defined. The proof must be performed in the following way: first (base case), the property is verified for the initial graph ($G0$) and then, at the inductive step, the property is verified for every rule of the grammar applicable to a reachable graph g (i.e., for $ap(\alpha i, m)\ g$), considering that the property is valid for g. This process may be semi-automated: it may proceed until a separate property or lemma is required, then we must establish the property or prove the lemma, and then the proof of the original goal can continue.

Now, we give two **examples** of proofs of properties for the Token Ring protocol: one about types of edges and another about the structure of reachable graphs.

Proposition 5.1 *Any reachable graph has (only) one edge of the type Tok.*

According to the definition of tip_E, previously established in the library, the property to be proven can be enunciated by the formula:

$$\exists! x\ [(x, Tok) \in tip_E\ reach\_gr].$$

Proof. Basis: Here, the property is verified for the initial graph $G0$.

$$\exists! x\ [(x, Tok) \in tip_E\ G0] \overset{(1)}{\equiv} \exists! x\ [t_{G0_E}(x, Tok)] \equiv true.$$

The last equivalences may be verified automatically. Since the relational structure that defines the grammar has a single pair with the second component

Tok belonging to the relation t_{G0_E} (see Example 3.6), the logical expression must be evaluated to *true*.

Hypothesis ⇒ Inductive Step: Assuming that $\exists! x[(x, Tok) \in tip_E reach\_gr]$, the proof reduces to five cases, depending on the rule that is applicable:

(i) $\exists! x\ [(x, Tok) \in tip_E\ ap(\alpha 1, m)\ reach\_gr)] \overset{(2)}{\equiv}$
$\exists! x\ [t_{R1_E}(x, Tok) \vee [(x, Tok) \in tip_E\ reach\_gr \wedge \nexists w\ m_{E_{\alpha 1}}(w, x)]]$.

Now it is necessary to inform if the edge x of type Tok of the reachable graph is an image of the match or not, when rule αi is applied. This can be done stating:

$$\forall x\ (x, Tok) \in tip_E\ reach\_gr, \quad \exists w\ m_{E_{\alpha i}}(w, x) \Leftrightarrow \exists w\ t_{Li_E}(w, Tok) \tag{3}$$

According to (3), the edge of type Tok of the reachable graph will be an image of the match if and only if the left-hand side of the applied rule contains an edge of the type Tok. Then:

$\exists! x\ [t_{R1_E}(x, Tok) \vee [(x, Tok) \in tip_E\ reach\_gr \wedge \nexists w\ m_{E_{\alpha 1}}(w, x)]] \overset{(3)}{\equiv}$
$\exists! x\ [t_{R1_E}(x, Tok) \vee [(x, Tok) \in tip_E\ reach\_gr \wedge \nexists w\ t_{L1_E}(w, Tok)]] \equiv true.$

There is a (single) pair at the relation t_{R1_E} that has the second component Tok (see Example 3.9). Besides it is assumed by hypothesis that $(x, Tok) \in tip_E\ reach\_gr$. Since expression $\nexists w\ m_{E_{\alpha 1}}(w, x)$ is evaluated to false (there is a pair in relation t_{L1_E} that has the second component Tok), the complete formula may be automatically evaluated to true.

$(ii-v)$ The proofs for rules $\alpha 2$, $\alpha 3$, $\alpha 4$ and $\alpha 5$ are analogous. It is important to notice that, since the property that informs if an edge of type Tok is the image of a match has already been stated, the verification for these rules may proceed automatically. □

Proposition 5.2 *Any reachable graph has a ring topology of edges of type Nxt.*

Considering that the transitive closure of edges and the function that identifies a ring topology are previously defined in the library, the property to be proven can be enunciated as:

$$\mathrm{Ring}_{Nxt}\ reach\_gr \equiv true.$$

Proof. Basis:

$$\mathrm{Ring}_{Nxt}\ G0 \overset{\text{def.}}{\equiv} \forall x\ [vert_{G0}(x) \rightarrow (x, x) \in TC^{Nxt}_{incG0}]\ \wedge \tag{1'}$$
$$\wedge \forall a, b, x, y, z\ [inc_{G0}(a, x, y) \wedge t_{G0_E}(a, Nxt) \wedge inc_{G0}(b, x, z) \wedge$$
$$\wedge\, t_{G0_E}(b, Nxt) \rightarrow a = b]\ \wedge \tag{2'}$$
$$\wedge \forall x, z\ [vert_{G0}(x) \wedge vert_{G0}(z) \rightarrow (x, z) \in TC^{Nxt}_{incG0}] \equiv \tag{3'}$$
$$\equiv true$$

Considering that the result of the operation TC^{Nxt}_{incG0} is the set $\{(N01, N02),$ $(N02, N03), (N03, N01), (N01, N03), (N02, N01), (N03, N02), (N01, N01), (N02,$ $N02), (N03, N03)\}$, (1') and (3') are satisfied. (2') is also satisfied because there are no two edges of the type Nxt in $G0$ starting at the same vertex (see Examples

3.3 and 3.6).

Hypothesis $\Rightarrow$ Inductive Step: $\text{Ring}_{Nxt}\ reach\_gr \equiv true \Rightarrow$

$$(i)\ \text{Ring}_{Nxt}\ ap(\alpha 1, m)\ reach\_gr \overset{\text{def.}}{\equiv} \text{Ring}_{Nxt}\ reach\_gr\ \wedge \tag{4'}$$

$$\wedge \forall a,x,y,z,w\ [inc_{L1}(a,x,y) \wedge t_{L1_E}(a,Nxt) \wedge \alpha 1_V(x,z)\ \wedge$$
$$\wedge\ \alpha 1_V(y,w) \rightarrow (z,w) \in TC^{Nxt}_{inc_{R1}}]\ \wedge \tag{5'}$$

$$\wedge \forall a,b,x,y,z\ [inc_{R1}(a,x,y) \wedge t_{R1_E}(a,Nxt) \wedge inc_{R1}(b,x,z) \wedge$$
$$\wedge\ t_{R1_E}(b,Nxt) \rightarrow a = b] \equiv \tag{6'}$$

$$\equiv true$$

This property may be verified automatically: (4') is valid by the induction hypothesis; (5') is valid by the result of the operation $TC^{Nxt}_{inc_{R1}}$; and (6') is valid because there aren't two edges of type Nxt starting at the same node in $R1$ (see Example 3.9).

$(ii - v)$ The proofs for rules $\alpha 2$, $\alpha 3$, $\alpha 4$ and $\alpha 5$ are analogous. □

6 Final Remarks

We have introduced a relational and logical approach to graph grammars to allow the analysis of asynchronous distributed systems with infinite state space. In order to represent this specification language, we have used a relational structure to characterize a graph grammar and defined rule applications as first-order definable transductions. The main aim of this work is to enable the use of the theorem proving technique to prove properties about graph grammars, especially properties about reachable states.

We plan to extend the approach proposed here to specific classes of graph grammars such as, for example, object-based graph grammars [10], appropriate for the specification of object-based systems, or timed object-based graph grammars, suitable to specify real time systems. In the first case, the set of vertices of the graph grammar must be partitioned in two subsets of objects and values (of abstract data types) and the set of edges should be replaced by a set of hyperedges that must be partitioned into sets of message and attribute edges. In the second case, we also have to add time stamps to the messages. This extension will allow the proof of properties usually not analyzed in model-checkers of object-based systems such as properties about the internal states of objects and their attributes. Another topic of future work is the implementation of the proposed approach using, for example, the Isabelle theorem prover [16].

References

[1] Baldan, P., A. Corradini, U. Montanari and L. Ribeiro, *Unfolding semantics of graph transformation*, Inf. Comput. **205** (2007), pp. 733–782.

[2] Baldan, P. and B. König, *Approximating the behaviour of graph transformation systems*, in: *Proceedings of ICGT '02 (International Conference on Graph Transformation)*, LNCS **2505** (2002), pp. 14–29.

[3] Baresi, L. and P. Spoletini, *On the use of Alloy to analyze graph transformation systems*, in: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro and G. Rozenberg, editors, *ICGT*, LNCS **4178** (2006), pp. 306–320.

[4] Behrmann, G., A. David and K. G. Larsen, *Tutorial on* UPPAAL, in: *Formal Methods for the Design of Real-Time Systems*, LNCS **3185**, Springer, 2004 pp. 200–236.

[5] Clarke, E. M., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Progress on the state explosion problem in model checking*, in: *Informatics - 10 Years Back. 10 Years Ahead.* (2001), pp. 176–194.

[6] Clarke, E. M. and J. M. Wing, *Formal methods: state of the art and future directions*, ACM Computing Surveys **28** (1996), pp. 626–643.

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

[8] Courcelle, B., *The expression of graph properties and graph transformations in monadic second-order logic.*, in: Rozenberg [20], pp. 313–400.

[9] Courcelle, B., *Recognizable sets of graphs, hypergraphs and relational structures: A survey.*, in: C. Calude, E. Calude and M. J. Dinneen, editors, *Developments in Language Theory*, LNCS **3340** (2004), pp. 1–11.

[10] Dotti, F. L., L. Foss, L. Ribeiro and O. M. Santos, *Verification of distributed object-based systems*, in: *6th International Conference on Formal Methods for Open Object-based Distributed Systems*, LNCS **2884** (2003), pp. 261–275.

[11] Dotti, F. L. and L. Ribeiro, *Specification of mobile code systems using graph grammars*, in: *Formal Methods for Open Object-Based Distributed Systems*, Kluwer, 2000 pp. 45–64.

[12] Dwyer, M. B., J. Hatcliff, R. Robby, C. S. Pasareanu and W. Visser, *Formal software analysis emerging trends in software model checking*, in: *FOSE '07: 2007 Future of Software Engineering* (2007), pp. 120–136.

[13] Gurevich, Y., *Monadic second-order theories*, in: J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, Springer, 1985 pp. 479–506.

[14] Holzmann, G. J., *The model checker Spin*, IEEE Transactions on Software Engineering **23** (1997), pp. 279–295.

[15] Michelon, L., S. A. Costa and L. Ribeiro, *Specification of real-time systems with graph grammars*, in: P. C. Masiero, editor, *Brazilian Symposium on Software Engineering*, 2006, pp. 97–112.

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

[17] Rensink, A., Á. Schmidt and D. Varró, *Model checking graph transformations: A comparison of two approaches*, in: *Proc. ICGT 2004: Second International Conference on Graph Transformation*, LNCS **3256** (2004), pp. 226–241.

[18] Ribeiro, L., F. Dotti and R. Bardohl, *A formal framework for the development of concurrent object-based systems*, in: *Formal Methods in Software and Systems Modeling*, LNCS **3393** (2005), pp. 385–401.

[19] Robinson, J. A. and A. Voronkov, editors, “Handbook of Automated Reasoning (in 2 volumes),” Elsevier and MIT Press, 2001.

[20] Rozenberg, G., editor, “Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations,” World Scientific, 1997.

[21] Strecker, M., *Modeling and verifying graph transformations in proof assistants*, Electronic Notes in Theoretical Computer Science **203** (2008), pp. 135–148.

[22] Tanenbaum, A., “Computer Networks,” Prentice Hall, 2002.