Electronic Communications of the EASST
Volume 30 (2010)

International Colloquium on Graph and Model Transformation - On the occasion of the 65th birthday of Hartmut Ehrig
(GraMoT 2010)

Towards Theorem Proving Graph Grammars using Event-B

Leila Ribeiro, Fernando Luís Dotti, Simone André da Costa and Fabiane Cristine Dillenburg

16 pages

Guest Editors: Claudia Ermel, Hartmut Ehrig, Fernando Orejas, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Towards Theorem Proving Graph Grammars using Event-B*

Leila Ribeiro[1], **Fernando Luís Dotti**[2], **Simone André da Costa**[3] **and Fabiane Cristine Dillenburg**[4]

[1] leila@inf.ufrgs.br
[4] fabiane.dillenburg@inf.ufrgs.br
Instituto de Informática
Universidade Federal do Rio Grande do Sul, Brazil

[2] fernando.dotti@pucrs.br
Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul, Brazil

[3] simone.costa@ufpel.edu.br
Institudo de Física e Matermática
Universidade Federal de Pelotas, Brazil

Abstract: Graph grammars may be used as specification technique for different kinds of systems, specially in situations in which states are complex structures that can be adequately modeled as graphs (possibly with an attribute data part) and in which the behavior involves a large amount of parallelism and can be described as reactions to stimuli that can be observed in the state of the system. The verification of properties of such systems is a difficult task due to many aspects: the systems in many situations involve an infinite number of states; states themselves are complex and large; there are a number of different computation possibilities due to the fact that rule applications may occur in parallel. There are already some approaches to verification of graph grammars based on model checking, but in these cases only finite state systems can be analyzed. Other approaches propose over- and/or under-approximations of the state-space, but in this case it is not possible to check arbitrary properties. In this work, we propose to use the Event-B formal method and its theorem proving tools to analyze graph grammars. We show that a graph grammar can be translated into an Event-B specification preserving its semantics, such that one can use several theorem provers available for Event-B to analyze the reachable states of the original graph grammar. The translation is based on a relational definition of graph grammars, that was shown to be equivalent to the Single-Pushout approach to graph grammars.

Keywords: Graph Grammars, Theorem Proving, Event-B

* Partially supported by CNPq/Brazil.

1 Introduction

Graph grammars [Ehr79, Roz97] are a formal description technique suitable for the specification of distributed and reactive systems. The basic idea of this formalism is to model the states of a system as graphs and describe the possible state changes as rules (where the left- and right-hand sides are graphs). The operational behavior of the system is expressed via applications of these rules to graphs depicting the current states of the system. Graph grammars are appealing as specification formalism because they are formal and based on simple, but powerful, concepts to describe behavior. At the same time they also have a nice graphical layout that helps even non-theoreticians to understand a specification. At the same time they also have a nice graphical layout that helps even non-theoreticians to understand a specification.

The verification of graph grammar models through model-checking is currently supported by various approaches. 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 [CGJ+01]. Baldan and König proposed [BK02] 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 [BCMR07] of the system. However, as [DHR+07] emphasizes, these approaches that derive the model as approximations can result in inconclusive error/verification reports.

Besides model checking, theorem proving [RV01, CW96] is another well-established approach used to analyze systems. Theorem proving 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 verification 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 construct proofs over infinite domains. The use of this technique may require interaction with a human; however, via this interactive process 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 complementary. Most of the existing approaches use model checkers to analyze properties of computations, that is, properties over the sequences of steps a system may engage in. Properties about reachable states are handled, if at all possible, only in restricted ways. In this work, our main aim is to provide a means to prove properties of reachable graphs using the theorem proving technique.

In previous work [CR09a] we proposed a relational approach to graph grammars, providing an encoding of graphs and rules into relations. This enabled the use of logic formulas to express properties of reachable states of a graph grammar. This encoding was shown to be equivalent to the Single-Pushout approach to graph grammars, and was inspired by Courcelle's research about logic and graphs [Cou97].

Courcelle investigates in various papers [Cou94, Cou97, Cou04] the representation of graphs and hypergraphs by relational structures as well as the expressiveness of their properties by logical languages. In [Cou94] 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). Since theorem provers, in general, works efficiently with specifications in relational style, we extended the relational representation of graphs to graph grammar models and use such representation for the formal analysis of reactive systems through the theorem proving technique. Other authors have investigated the analysis of GTSs based on relational logic or set theory. Baresi and Spoletini [BS06] explore the formal language Alloy to find instances and counterexamples for models and GTSs. With Alloy, they only analyze the system for a finite scope, whose size is user-defined. Strecker [Str08], aiming to verify structural properties of GTSs, proposes a formalization of graph transformations in a set-theoretic model. His goal is to obtain a language for writing graph transformation programs and reasoning about them. Nevertheless, the language has only two statements, one to apply a rule repeatedly to a graph, and another to apply several rules in a specific order to a graph. Until now, the work just presents a glimpse of how to reason about graph transformations.

In this paper we use Event-B to analyze properties of graph grammars. Event-B [AH07] is a state-based formal method closely related to Classical B [Abr05]. It has been successfully used in several applications, and there is tool support for both model specification and analysis. There are a series of powerful theorem provers that can be used to analyze event-B specifications[ABHV06, DEP]. Due to the similarity between event-B models and graph grammar specifications, specially concerning the rule-based behavior, in this paper we propose to translate graph grammar specifications in event-B structures, such that it is possible to use the event-B provers to demonstrate properties of a graph grammar. This translation is based on the relational definition of graph grammars.

The paper is organized as follows. Section 2 presents the relational approach of graph grammars. Section 3 briefly introduces the event B formalism. Section 4 shows how a graph grammar can be translated into an Event-B program. Section 5 contains some final remarks.

2 Relational Approach to Graph Grammars

Graph Grammars are a generalization of Chomsky grammars from strings to graphs suitable for the specification of distributed, asynchronous and concurrent systems. The basic notions behind this formalism are: states are represented by graphs and possible state changes are modeled by rules, where the left- and right-hand sides are graphs.

We use a relational and logical approach for the description of Graph Grammars: graphs and graph morphisms are described as relational structures [CR09a, CR10], that is, they are defined as tuples formed by a set and by a family of relations over this set. Proofs about the well-definedness of these definitions were detailed in [CR09b].

Definition 1 (Relational Structures) Let $\mathscr{R}$ be a finite set of relation symbols, where each $R \in \mathscr{R}$ has an associated positive integer called its arity, denoted by $\rho(R)$. An **$\mathscr{R}$-structure** is a tuple $S = \langle D_S, (R_S)_{R \in \mathscr{R}} \rangle$ 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 $d_1, \ldots, d_n \in D_S$.

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: 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 $|G|$. The *uniqueness edge condition* assures that the same edge doesn't connect different vertices.

Definition 2 (Relational Graph) Let $\mathscr{R}_{gr} = \{vert, inc\}$ be a set of relations, where *vert* is unary and *inc* is ternary. A **relational graph** is a $\mathscr{R}_{gr}$-structure $|G| = \langle D_G, (R_G)_{R\in\mathscr{R}_{gr}}\rangle$, where:

- $D_G = V_G \cup E_G$ is the union of sets of possible vertices and edges of $|G|$, respectively (we always assume that $V_G \cap E_G = \varnothing$);
- $vert_G \subseteq V_G$, with $vert_G(x)$ iff x is a vertex of $|G|$;
- $inc_G \subseteq E_G \times V_G \times V_G$, with $inc_G(x,y,z)$ iff x is a directed edge that links vertex y to vertex z in $|G|$.

such that the following condition is satisfied:

- **Uniqueness Edge Condition.** $\forall x,y,z,y',z'$,
  $[inc_G(x,y,z) \wedge inc_G(x,y',z') \Rightarrow y = y' \wedge z = z']$.

A relational graph morphism $|g|$ from a relational graph $|G|$ to a relational graph $|H|$ is obtained through two binary relations: one to relate vertices (g_V) and other to relate edges (g_E). The *type consistency conditions* state 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 *(morphism) commutativity condition* assures that the mapping of edges preserves the mapping of source and target vertices.

Definition 3 (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 relational 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:

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

A relational typing morphism is a relational graph morphism that has the role of typing all elements of a graph $|G|$ over a graph $|T|$.

Definition 4 (Relational Typing Morphism) Let $|G|$ and $|T|$ be relational graphs. A **relational typing morphism from** $|G|$ **over** $|T|$ is defined by a total relational graph morphism $|t^G| = \{t_V^G, t_E^G\}$ from $|G|$ to $|T|$.

A relational typed graph is defined by two relational graphs together with a relational typing morphism. A relational typed graph morphism between graphs typed over the same graph is defined by a relational graph morphism. A *(typed morphism) compatibility condition* assures that the mappings of vertices and edges preserve types.

Definition 5 (Relational Typed Graph, Relational Typed Graph Morphism) A **relational typed graph** is given by a tuple $|G^T| = \langle |G|, |t^G|, |T| \rangle$ where $|G|$ and $|T|$ are relational graphs and $|t^G| = \{t_V^G, t_E^G\}$ is a relational typing morphism from $|G|$ over $|T|$. A **relational (typed) graph morphism from** $|G^T|$ **to** $|H^T|$ is defined by a relational graph morphism $|g| = \{g_V, g_E\}$ from $|G|$ to $|H|$, such that the typed morphism compatibility condition is satisfied:

- **(Typed Morphism) Compatibility Condition.** $\forall x, x', y$,
  $[g_V(x,x') \wedge t_V^G(x,y) \Rightarrow t_V^H(x',y)]$; and
  $[g_E(x,x') \wedge t_E^G(x,y) \Rightarrow t_E^H(x',y)]$.

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

Definition 6 (Relational Rule) A **relational rule** 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 typed graphs;
- $|\alpha| = \{\alpha_V, \alpha_E\}$ is an injective relational typed graph morphism from $|L^T|$ to $|R^T|$, such that α_V is a total function on the set of vertices.

A relational graph grammar is composed by a *relational type graph*, characterizing the types of vertices and edges allowed in a system, 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.

Definition 7 (Relational Graph Grammar) Let $\mathscr{R}_{GG} = \{vert_T,\ inc_T,\ vert_{G0}, inc_{G0},\ t_V^{G0},\ t_E^{G0},\ (vert_{Li},\ inc_{Li},\ t_V^{Li},\ t_E^{Li}, vert_{Ri}, inc_{Ri},\ t_V^{Ri},\ t_E^{Ri},\ \alpha_{i_V},\ \alpha_{i_E})_{i\in\{1,\ldots,n\}}\}$ be a set of relation symbols. A **relational graph grammar** is a $\mathscr{R}_{GG}$-structure $|GG| = \langle D_{GG}, (r)_{r\in\mathscr{R}_{GG}} \rangle$ where

- $D_{GG} = V_{GG} \cup E_{GG}$ is the set of vertices and edges of the graph grammar, where: $V_{GG} \cap E_{GG} = \varnothing$, $V_{GG} = V_T \cup V_{G0} \cup (V_{Li} \cup V_{Ri})_{i\in\{1,\ldots,n\}}$ and $E_{GG} = E_T \cup E_{G0} \cup (E_{Li} \cup E_{Ri})_{i\in\{1,\ldots,n\}}$.
- $|T| = \langle V_T \cup E_T, \{vert_T, inc_T\} \rangle$ defines a relational graph **(the type of the grammar)**.
- $|G0^T| = \langle |G0|, |t^{G0}|, |T| \rangle$, with $|G0| = \langle V_{G0} \cup E_{G0}, \{vert_{G0}, inc_{G0}\} \rangle$ and $|t^{G0}| = \{t_V^{G0}, t_E^{G0}\}$, defines a relational typed graph **(the initial graph of the grammar)**.

- Each collection $(vert_{Li},\ inc_{Li},\ t_V^{Li},\ t_E^{Li},\ vert_{Ri}, inc_{Ri},\ t_V^{Ri},\ t_E^{Ri},\ \alpha_{i_V},\ \alpha_{i_E})$ defines a **rule**:
  - $|Li^T| = \langle |Li|, |t^{Li}|, |T| \rangle$, with $|Li| = \langle V_{Li} \cup E_{Li}, \{vert_{Li}, inc_{Li}\} \rangle$ and $|t^{Li}| = \{t_V^{Li}, t_E^{Li}\}$, defines a relational typed graph **(the left-hand side of the rule)**.
  - $|Ri^T| = \langle |Ri|, |t^{Ri}|, |T| \rangle$, with $|Ri| = \langle V_{Ri} \cup E_{Ri}, \{vert_{Ri}, inc_{Ri}\} \rangle$ and $|t^{Ri}| = \{t_V^{Ri}, t_E^{Ri}\}$, defines a relational typed graph **(the right-hand side of the rule)**.
  - $\langle |Li^T|, |\alpha_i|, |Ri^T| \rangle$, with $|\alpha_i| = \{\alpha_{i_V}, \alpha_{i_E}\}$, defines a relational rule.

Given a relational rule and a state, we say that this rule is applicable in this state if there is a match, 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 applications of the rules to some state graph.

Definition 8 (Relational Match) Let $\langle |L^T|, |\alpha|, |R^T| \rangle$ be a relational rule, with $|L^T| = \langle |L|, \{t_V^L, t_E^L\}, |T| \rangle$ and $|R^T| = \langle |R|, \{t_V^R, t_E^R\}, |T| \rangle$. Let $|G^T| = \langle |G|, |t^G|, |T| \rangle$ be a relational typed graph with $t^G = \{t_V^G, t_E^G\}$. A **relational match** $|m|$ **of the given rule in** $|G^T|$ is defined by a total relational typed graph morphism $|m| = \{m_V, m_E\}$ from $|L^T|$ to $|G^T|$, such that the following conditions are satisfied:

- m_E is injective;
- **Match Compatibility Condition.** $\forall x, x', y$
  $[m_V(x,x') \wedge t_V^L(x,y) \Rightarrow t_V^G(x',y)]$,
  $[m_E(x,x') \wedge t_E^L(x,y) \Rightarrow t_E^G(x',y)]$.

Since we restrict our approach to injective rules that can not delete vertices and matches that can no identify edges, the application of a given rule to a match in a state essentially removes from the state all elements that are in the left-hand side of the rule that are not mapped to the right-hand side, and creates in the state all new elements of the right-hand side of the rule. The rest of the state remains unchanged.

Given a rule $\langle |Li^T|, |\alpha_i|, |Ri^T| \rangle$ of a graph grammar and a corresponding match $|m| = \{m_V, m_E\}$ in the initial state of the graph grammar, formulas $\theta_{vert_{G'}}$, $\theta_{inc_{G'}}$, $\theta_{t_V^{G'}}$, $\theta_{t_E^{G'}}$ described below define the graph resulting of the rule application. The elements that satisfy the stated formulas θ_{rel} are those that define the relations rel of the resulting typed graph $|G'^T|$. Table 1 presents the explanations for the notation used in θ specifications.

$$
\begin{aligned}
\theta_{vert_{G'}}(x) &= vert_{G0}(x) \vee nvert_{Ri}(x) \\
\theta_{inc_{G'}}(x,y,z) &= ninc_{G0}(x,y,z) \vee ninc_{Ri}(x,y,z). \\
\theta_{t_V^{G'}}(x,t) &= nvert_{G0}(x,t) \vee \left[nvert_{Ri}(x) \wedge t_V^{Ri}(x,t) \right]. \\
\theta_{t_E^{G'}}(x,t) &= nt_E^{G0}(x,t) \vee t_E^{Ri}(x,t).
\end{aligned}
$$

This construction is described by a first-order definable transduction (i.e., by a tuple of first-order formulas) on relational structures associated to graph grammars. Details can be found in [CR09a].

Table 1: Formulas used in θ specifications

Notation	Formula	Intuitive Meaning
$vert_{G0}(x)$	$vert_{G0}(x)$	x is a vertex of graph $\lvert G0\rvert$.
$t_V^{Ri}(x,y)$	$t_V^{Ri}(x,y)$	x is a vertex of $\lvert Ri\rvert$ of type y.
$t_E^{Ri}(x,y)$	$t_E^{Ri}(x,y)$	x is an edge of graph $\lvert Ri\rvert$ of type y.
$nvert_{Ri}(x)$	$vert_{Ri}(x) \wedge \nexists y \left(\alpha_{i_V}(y,x)\right)$	x is a vertex of graph $\lvert Ri\rvert$ created by rule $\lvert\alpha_i\rvert$.
$ninc_{G0}(x,y,z)$	$inc_{G0}(x,y,z) \wedge \nexists w \left(m_E(w,x)\right)$	x is an edge of graph $\lvert G0\rvert$ that is not image of the match.
$\overline{n}(r,y)$	$\begin{cases} \exists v \left(\alpha_{i_V}(v,r) \wedge m_V(v,y)\right) \text{ if } r \neq y \\ \nexists v \ \alpha_{i_V}(v,r) \text{ if } r = y \end{cases}$	$\overline{n}$ relates vertices r and y if (i) $r = y$ and r is created by rule $\lvert\alpha_i\rvert$, or (ii) there is a vertex v preserved by the rule whose images in Ri and $G0$ are r and y, resp.
$ninc_{Ri}(x,y,z)$	$\exists r,s \left[inc_{Ri}(x,r,s) \wedge \overline{n}(r,y) \wedge \overline{n}(s,z)\right]$	x is an edge created by rule $\lvert\alpha_i\rvert$ (connecting existing or newly created vertices.
$nvert_{G0}(x,t)$	$vert_{G0}(x) \wedge t_V^{G0}(x,t)$	x is a vertex of $\lvert G0\rvert$ of type t.
$nvert_{Ri}(x,t)$	$vert_{Ri}(x) \wedge t_V^{Ri}(x,t)$	x is a vertex of $\lvert Ri\rvert$ of type t.
$nt_E^{G0}(x,t)$	$\exists y,z \left(inc_{G0}(x,y,z)\right) \wedge \ \nexists w \left(m_E(w,x)\right) \ \wedge \ \wedge \ t_E^{G0}(x,t)$	x is an edge of graph $\lvert G0\rvert$ of type t that is not image of the match.

3 Event-B

Event-B [AH07] is a state-based formalism closely related to Classical B [Abr05] and Action Systems [BS89].

Definition 9 (Event-B Model, Event) An Event-B Model is defined by a tuple $EBModel = (c, s, P, v, I, R_I, E)$ where c are constants and s are sets known in the model; v are the model variables[1]; $P(c,s)$ is a collection of axioms constraining c and s; $I(c,s,v)$ is a model invariant limiting the possible states of v s.t. $\exists c,s,v \cdot P(c,s) \wedge I(c,s,v)$ - i.e. P and I characterise a non-empty set of model states; $R_I(c,s,v')$ is an initialisation action computing initial values for the model variables; and E is a set of model *events*.

Given states v, v' an event is a tuple $e = (H,S)$ where $H(c,s,v)$ is the guard and $S(c,s,v,v')$ is the before-after predicate that defines a relation between current and next states. We also denote an event guard by $H(v)$, the before-after predicate by $S(v,v')$ and the initialization action by $R_I(v')$.

An event-B model is assembled from two parts, a *context* which defines the triple (c,s,P) and a *machine* which defines the other elements (v, I, R_I, E).

Model correctness is demonstrated by generating and discharging a collection of proof obligations. The model *consistency* condition states that whenever an event on an initialisation action is attempted, there exists a suitable new state v' such that the model invariant is maintained - $I(v')$. This is usually stated as two separate proof obligations: a feasibility $(I(v) \wedge H(v) \Rightarrow \exists v' \cdot S(v,v'))$ and an invariant satisfaction obligation $(I(v) \wedge H(v) \wedge S(v,v') \Rightarrow I(v'))$. The behaviour of an Event-B model is the transition system defined as follows.

Definition 10 (Event-B Model Behaviour) Given $EBModel = (c, s, P, v, I, R_I, E)$, its behaviour is given by a transition system $BST = (BState, BS_0, \rightarrow)$ where: $BState = \{\langle v \rangle | v \text{ is a } state\} \cup Undef$, $BS_0 = Undef$, and $\rightarrow \subseteq BState \times BState$ is the transition relation given by the rules:

$$\boxed{\text{start}}\ \frac{R_I(v') \wedge I(v')}{Undef \rightarrow \langle v' \rangle}$$

$$\boxed{\text{transition}}\ \frac{\exists (H,S) \in E \cdot I(v) \wedge H(v) \wedge S(v,v') \wedge I(v')}{\langle v \rangle \rightarrow \langle v' \rangle}$$

According to rule *start* the model is initialized to a state satisfying $R_I \wedge I$ and then, as long as there is an enabled event (rule *transition*), the model may evolve by firing an enabled event and computing the next state according to the event's before-after predicate. Events are atomic. In case there is more than one enabled event at a certain state, the choice is non-deterministic. The semantics of an Event-B model is given in the form of proof semantics, based on Dijkstra's work on weakest preconditions [Dij76].

An extensive tool support through the Rodin Platform makes Event-B especially attractive [DEP]. An integrated Eclipse-based development environment is actively developed, and open to third-party extensions in the form of Eclipse plug-ins. The main verification technique is

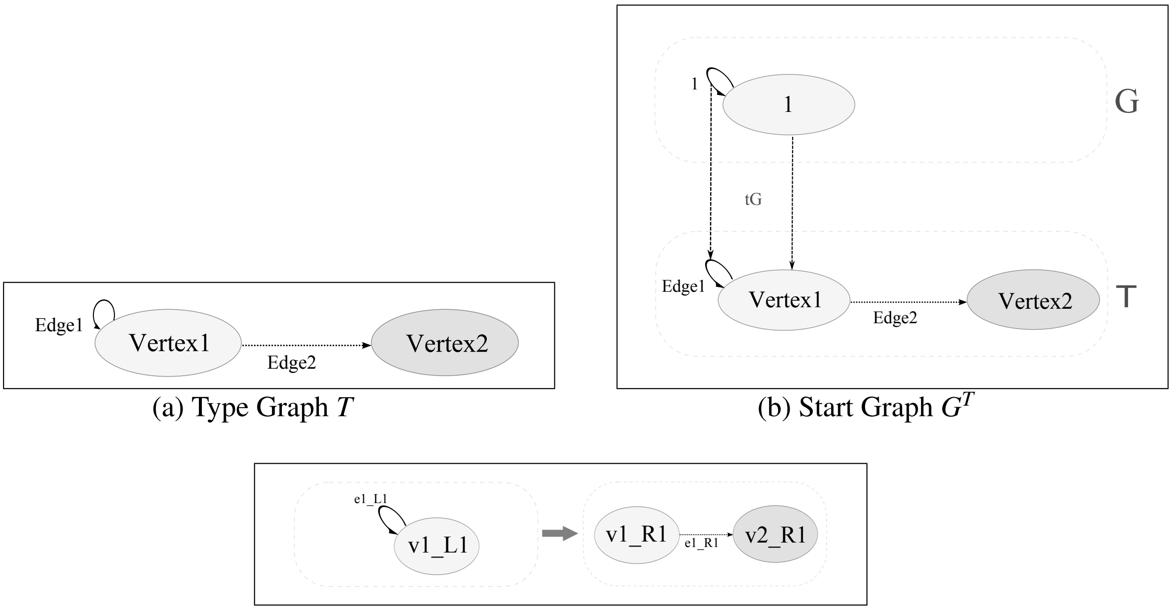
[1] For convenience, as in [Abr05], no distinction is made between a set of variables and a state of a system.

theorem proving supported by a collection of theorem provers, but there is also some support for model checking.

4 Verification of Graph Grammars using Event-B

The behavior of an event-B model is similar to a graph grammar: there is a notion of state (given by a set of variables in event-B, and by a graph in a graph grammar) and a step is defined by an atomic operation on the current state (an event that updates variables in event-B and a rule application in a graph grammar). Each step should preserve properties of the state. In event-B, these properties are stated as invariants. In a graph grammar, the properties that are inherently guaranteed to be preserved are related to the graph structure (only well-formed graphs can be generated).

Now, we present a way to model each structure of a graph grammar GG in event-B such that it is possible to use the event-B provers to demonstrate properties of a graph grammar. We will use an example to describe how graphs, typed graphs and rules can be defined in Event-B. The example is depicted in Figure1.



(a) Type Graph T (b) Start Graph G^T

(c) Rule $\alpha 1$

Figure 1: Example of Graph Grammar

Graphs: According to Def. 2 and Def. 7, sets V_{GG} and E_{GG} contain all possible vertices and edge names that may appear in graphs of this relational structure. We will define these sets as

$V_{GG} = vert_T \cup \mathbb{N}$, where $vert_T$ is the set of names used as vertex types in GG (we assume that $vert_T \cap \mathbb{N} = \varnothing$);

$E_{GG} = edge_T \cup \mathbb{N}$, where $edge_T$ is the set of names used as edge types in GG (we assume that $edge_T \cap \mathbb{N} = \varnothing$).

Moreover, we assume that $vert_T \cap edge_T = \varnothing$.

The type graph T is defined in an event-B context as described in Figure 2, where we define all vertex and edge types as constants. In the axioms, we define these sets explicitly (for example, axiom $axm1$ means that $vertT = \{Vertex1, Vertex2\}$). We also define the functions $sourceT$ and $targetT$ that respectively designate the source and target vertex of each edge. Text after a // is a comment. Here, instead of using the ternary inc relation we used a set of edges and two binary relations ($source$ and $target$) to define the edges of a graph. This is an equivalent formulation that is convenient to use in Event-B because it eases the proof of some proof obligations.

CONTEXT ctx_GG
SETS
 `vertT` // (Type Graph) Vertices
 `edgeT` // (Type Graph) Edges
CONSTANTS
 `Vertex1 Vertex2`
 `Edge1 Edge2`
 `sourceT` // (Type Graph) Source Function
 `targetT` // (Type Graph) Target Function
AXIOMS
 `axm1` : $partition(vertT, \{Vertex1\}, \{Vertex2\})$
 `axm2` : $partition(edgeT, \{Edge1\}, \{Edge2\})$
 `axm3` : $sourceT \in edgeT \rightarrow vertT$
 `axm4` : $partition(sourceT, \{Edge1 \mapsto Vertex1\}, \{Edge2 \mapsto Vertex1\})$
 `axm5` : $targetT \in edgeT \rightarrow vertT$
 `axm6` : $partition(targetT, \{Edge1 \mapsto Vertex1\}, \{Edge2 \mapsto Vertex2\})$
END

Figure 2: Event-B Type Graph

Instances of vertices and edges that appear in graphs representing states will be described by natural numbers. It is not necessary to have distinct numbers for vertices and edges: a graph may have a vertex with identity 1 as well as an edge with identity 1, these elements will be different because one will be mapped to a vertex type and the other to an edge type.

A graph typed over a type graph T is modeled by a set of variables describing its set of vertices, set of edges, source, target and typing functions. It is possible to state the compatibility conditions of types and source and target of edges (stated in Def. 3) as invariants. However, since we will always generate well-formed graphs (the start graph is well-formed and events implement the single-pushout construction), we will skip these invariants (each invariant that is used generates proof obligations and therefore it is advisable to use only the necessary ones). Figure 3 shows the definition of a graph G typed over T. Invariants are used to define the types of the variables (for example, tG_V is a total function from $vertG$ to $vertT$ and tG_E is a total function from $edgeG$ to $edgeT$).

MACHINE mch_GG
SEES ctx_GG
VARIABLES
 vertG // (Graph) Vertices
 edgeG // (Graph) Edges
 sourceG // (Graph) Source Function
 targetG // (Graph) Target Function
 tG_V // Typing of vertices
 tG_E // Typing of edges
INVARIANTS
 inv_vertG : $vertG \in \mathbb{P}(\mathbb{N})$
 inv_incG : $edgeG \in \mathbb{P}(\mathbb{N})$
 inv_sourceG : $sourceG \in edgeG \rightarrow vertG$
 inv_targetG : $targetG \in edgeG \rightarrow vertG$
 inv_tG_V : $tG\_V \in vertG \rightarrow vertT$
 inv_tG_E : $tG\_E \in edgeG \rightarrow edgeT$
EVENTS
Initialisation
 begin
  act1 : $vertG := \{10\}$
  act2 : $edgeG := \{20\}$
  act3 : $sourceG := \{20 \mapsto 10\}$
  act4 : $targetG := \{20 \mapsto 10\}$
  act5 : $tG\_V := \{10 \mapsto Vertex1\}$
  act6 : $tG\_E := \{20 \mapsto Edge1\}$
 end

Figure 3: Event-B Graph G

There is special event in an event-B model that is executed before any other. This is the initialization event. In our encoding, this event will be used to create the start graph of a graph grammar. This is done by assigning initial values to the variables that correspond to graph G (see Figure 3). Within an event, the order in which attributions occur in non-deterministic.

Rules: Left- and right-hand sides of rules are graphs, and thus will have representations as defined previously. Additionally, we have to define the partial morphism (α_V, α_E) that maps elements from the left- to the right-hand side of the rule. The Event-B enconding of ule $\alpha 1$ depicted in Figure 1 is shown in Figure 4. Since rules do not change during execution, their structures will be defined as constants.

The behavior of a rule is described by an event (for the example, by event $alpha1$ in Figure 5). Whenever there are concrete values for variables mV, mE, $newV$ and $newE$ that satisfies the guard conditions, the event may occur. Guard conditions $grd1$, $grd2$ and $grd5$ to $grd7$ assure that the pair (mV, mE) is actually a match from the left-hand side of the

SETS

    vertL1
    edgeL1
    vertR1
    edgeR1

CONSTANTS

    v1_L1   // vertex of LHS
    e1_L1   // edge of LHS
    v1_R1 v2_R1   // vertices of RHS
    e1_R1   // edge of RHS
    sourceL1
    targetL1
    sourceR1
    targetR1
    tL1_V   // (Rule 1) Typing vertices of LHS
    tL1_E   // (Rule 1) Typing edges of LHS
    tR1_V   // (Rule 1) Typing vertices of RHS
    tR1_E   // (Rule 1) Typing edges of RHS
    alpha1V   // (Rule 1) Rule morphism: mapping vertices
    alpha1E   // (Rule 1) Rule morphism: mapping edges

AXIOMS

// GRAPH L1:

axm7 : $partition(vertL1, \{v1\_L1\})$
axm8 : $partition(edgeL1, \{e1\_L1\})$
axm9 : $sourceL1 \in edgeL1 \rightarrow vertL1$
axm10 : $partition(sourceL1, \{e1\_L1 \mapsto v1\_L1\})$
axm11 : $targetL1 \in edgeL1 \rightarrow vertL1$
axm12 : $partition(targetL1, \{e1\_L1 \mapsto v1\_L1\})$
axm13 : $tL1\_V \in vertL1 \rightarrow vertT$
axm14 : $partition(tL1\_V, \{v1\_L1 \mapsto Vertex1\})$
axm15 : $tL1\_E \in edgeL1 \rightarrow edgeT$
axm16 : $partition(tL1\_E, \{e1\_L1 \mapsto Edge1\})$

// GRAPH R1:

axm17 : $partition(vertR1, \{v1\_R1\}, \{v2\_R1\})$
axm18 : $partition(edgeR1, \{e1\_R1\})$
axm19 : $sourceR1 \in edgeR1 \rightarrow vertR1$
axm20 : $partition(sourceR1, \{e1\_R1 \mapsto v1\_R1\})$
axm21 : $targetR1 \in edgeL1 \rightarrow vertR1$
axm22 : $partition(targetR1, \{e1\_R1 \mapsto v2\_R1\})$
axm23 : $tR1\_V \in vertR1 \rightarrow vertT$
axm24 : $partition(tR1\_V, \{v1\_R1 \mapsto Vertex1\}, \{v2\_R1 \mapsto Vertex2\})$
axm25 : $tR1\_E \in edgeR1 \rightarrow edgeT$
axm26 : $partition(tR1\_E, \{e1\_R1 \mapsto Edge2\})$

// Rule morphism alpha1:

axm27 : $alpha1V \in vertL1 \rightarrow vertR1$
axm28 : $partition(alpha1V, \{v1\_L1 \mapsto v1\_R1\})$
axm29 : $alpha1E \in edgeL1 \rightarrowtail edgeR1$
axm30 : $alpha1E = \varnothing$

END

Figure 4: Event-B Rule Strucure

rule to the state graph G (see Def. 8). Guard conditions $grd3$ and $grd4$ assure that $newV$ and $newE$ are new fresh elements (a new vertex and a new edge identifier, not belonging to graph G). The actions update the state graph (graph G) according to the rule. In this example one loop edge is deleted and a new vertex and a new edge are created. A vertex $newV$ with type $Vertex2$, and an edge $newE$ with type $Edge2$ are generated. The source of this new edge is the image of the only vertex in the left-hand side of the rule in G and the target is the newly created vertex. The relational operators[2] used in the definition of the actions implement the formulas that define rule application in Sect. 2. This is an encoding of rule $\alpha 1$, there is a concrete choice for identifiers of elements created by the rule ($newV$ and $newE$). For this reason and to obtain a more efficient encoding, we did not use explicitly the functions $sourceR1$, $targetR1$, $alpha1V$ and $alpha1E$ to define this event (but they were implicitly used to define the actions). For example, to obtain the set of vertices of the resulting graph we used the existing set of vertices $vertG$ and added a set containing $newV$, instead of taking a vertex of $R1$ that was not in the image of $alpha1V$ (there is a vertex of $R1$ that is not in the image of $alpha1V$, that is $v2\_R2$, and by giving it the name $newV$ in the generated graph we assure that this name did not occur already in $vertG$). Note that this choice of representation was dependent on the Event-B language, if we were to translate graph grammars to a different language, other encodings of the relational representation might be more suitable.

Proving Properties: Once the start graph and all rules are represented in the event-B model, the property to be proved can be stated as an invariant. For example, we could add the invariant $card(edgeG) \leq 2$, meaning that no reachable graph can have more than 2 edges. For the given example, this property is true, and this can be easily proven automatically by the Rodin platform.

5 Final Remarks

In this paper we have defined an event-B model that faithfully describes the behavior of a given graph grammar. To define this model, we used the relational definition of graph grammars, that was proven to be equivalent to the SPO approach. Now, it is possible to use the existing theorem provers for event-B to prove properties of graph grammars, for example, using the Rodin platform.

This is an initial work in using event-B to help proving properties of graph grammars. Besides implementation, case studies are necessary to evaluate and improve the proposed approach. Another interesting topic for further research is to investigate to which extent the theory of refinement, that is very well-developed in event-B, can be used to validate a stepwise development based on graph grammars.

[2] The relational operators used to define this event are: $\setminus$ (minus), $\cup$ (union), $\lhd\!\!\!-$ (domain subtraction).

EVENTS
Event $alpha1 \mathrel{\widehat{=}}$
  any
    mV
    mE
    $newV$
    $newE$
  where
    grd1 : $mV \in vertL1 \rightarrow vertG$ // total on vertices
    grd2 : $mE \in edgeL1 \rightarrowtail edgeG$ // total and injective on edges
    grd3 : $newV \in \mathbb{N} \setminus vertG$ // newV is a fresh vertex name
    grd4 : $newE \in \mathbb{N} \setminus edgeG$ // newE is a fresh edge name
    grd5 : $\forall v \cdot v \in vertL1 \Rightarrow tL1\_V(v) = tG\_V(mV(v))$
      // vertex type compatibility
    grd6 : $\forall e \cdot e \in edgeL1 \Rightarrow tL1\_E(e) = tG\_E(mE(e))$
      edge type compatibility
    grd7 : $\forall e \cdot e \in edgeL1 \Rightarrow mV(sourceL1(e)) = sourceG(mE(e)) \wedge mV(targetL1(e)) = targetG(mE(e))$
      source/target compatibility
  then
    act1 : $vertG := vertG \cup \{newV\}$
    act2 : $edgeG := (edgeG \setminus \{mE(e1\_L1)\}) \cup \{newE\}$
    act3 : $sourceG := (\{mE(e1\_L1)\} \lhd sourceG) \cup \{newE \mapsto mV(v1\_L1)\}$
    act4 : $targetG := (\{mE(e1\_L1)\} \lhd targetG) \cup \{newE \mapsto newV\}$
    act5 : $tG\_V := tG\_V \cup \{newV \mapsto Vertex2\}$
    act6 : $tG\_E := (\{mE(e1\_L1)\} \lhd tG\_E) \cup \{newE \mapsto Edge2\}$
  end
END

Figure 5: Event-B Rule Event

Bibliography

[ABHV06] J.-R. Abrial, M. J. Butler, S. Hallerstede, L. Voisin. An Open Extensible Tool Environment for Event-B. In Liu and He (eds.), *ICFEM*. Lecture Notes in Computer Science 4260, pp. 588–605. Springer, 2006.

[Abr05] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[AH07] J.-R. Abrial, S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.* 77(1-2):1–28, 2007.

[BCMR07] P. Baldan, A. Corradini, U. Montanari, L. Ribeiro. Unfolding semantics of graph transformation. *Inf. Comput.* 205(5):733–782, 2007.
doi:http://dx.doi.org/10.1016/j.ic.2006.11.004

[BK02] P. Baldan, B. König. Approximating the behaviour of graph transformation systems. In *Proceedings of ICGT '02 (International Conference on Graph Transformation)*. LNCS 2505, pp. 14–29. Springer, 2002.

[BS89] R.-J. Back, K. Sere. Stepwise Refinement of Action Systems. In Snepscheut (ed.), *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*. Pp. 115–138. Springer, London, UK, 1989.

[BS06] L. Baresi, P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In Corradini et al. (eds.), *ICGT*. LNCS 4178, pp. 306–320. Springer, 2006.

[CGJ+01] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead.* Pp. 176–194. Springer, London, UK, 2001.

[Cou94] B. Courcelle. Monadic Second-Order Definable Graph Transductions: A Survey. *Theoretical Computer Science* 126(1):53–75, 1994.

[Cou97] B. Courcelle. The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. Pp. 313–400 in [Roz97].

[Cou04] B. Courcelle. Recognizable Sets of Graphs, Hypergraphs and Relational Structures: A Survey. In Calude et al. (eds.), *Developments in Language Theory*. LNCS 3340, pp. 1–11. Springer, 2004.

[CR09a] S. A. da Costa, L. Ribeiro. Formal Verification of Graph Grammars using Mathematical Induction. *Electronic Notes Theoretical Computer Science* 240:43–60, 2009.
doi:http://dx.doi.org/10.1016/j.entcs.2009.05.044

[CR09b] S. A. da Costa, L. Ribeiro. Relational and Logical Approach to Graph Grammars. Technical report 359, Porto Alegre: Instituto de Informática/UFRGS, 2009.

[CR10] S. A. da Costa, L. Ribeiro. Formal Verification of Graph Grammars using Mathematical Induction. *Science of Computer Programming*, 2010. doi:http://dx.doi.org/10.1016/j.scico.2010.02.006

[CW96] E. M. Clarke, J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys* 28(4):626–643, 1996. doi:http://doi.acm.org/10.1145/242223.242257

[DEP] DEPLOY. Event-B and the Rodin Platform. http://www.event-b.org/ (last accessed 16 March 2010). Rodin Development is supported by European Union ICT Projects DEPLOY (2008 to 2012) and RODIN (2004 to 2007).

[DHR+07] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, W. Visser. Formal Software Analysis Emerging Trends in Software Model Checking. In *FOSE '07: 2007 Future of Software Engineering*. Pp. 120–136. IEEE Computer Society, 2007. doi:http://dx.doi.org/10.1109/FOSE.2007.6

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

[Ehr79] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology*. Lecture Notes in Computer Science 73, pp. 1–69. Springer-Verlag, Germany, 1979.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[RV01] J. A. Robinson, A. Voronkov (eds.). *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[Str08] M. Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *Electronic Notes in Theoretical Computer Science* 203(1):135–148, 2008. doi:http://dx.doi.org/10.1016/j.entcs.2008.03.039