Theorem Proving Graph Grammars: Strategies for Discharging Proof Obligations*

Luiz Carlos Lemos Junior, Simone André da Costa Cavalheiro, and Luciana Foss

Universidade Federal de Pelotas, Centro de Desenvolvimento Tecnológico
Rua Gomes Carneiro, 1, 96010-610, Pelotas - RS, Brazil
{lclemos,simone.costa,lfoss}@inf.ufpel.edu.br

Abstract. One way of developing reliable systems is through the use of Formal Methods. A Graph Grammar specification is visual and based in a simple mechanism of rewriting rules. On the other hand, verification through theorem proving allows the proof of properties for systems with huge (and infinite) state space. There is a previously proposed approach that has allowed the application of theorem proving technique to graph grammars. One of the disadvantages of such an approach (and theorem proving in general) is the specific mathematical knowledge required from the user for concluding the proofs. This paper proposes proof strategies in order to help the developer in the verification process through theorem proving, when adopting graph grammar as specification language.

1 Introduction

In the present scenario we find a wide variety of software and hardware systems that are increasingly complex. In this situation, it is important to adopt strategies for increasing reliability. A way of achieve such a goal is using formal specification and verification. A formal specification is carried by a mathematical model, with well-defined syntax and semantics and formal verification can guarantee system properties.

There are several specification languages, among them, graph grammars (GG) [1] stand out, which are visual, based on rewriting rules and capable of describing complex behaviours. In graph grammars, states are modelled as graphs and state changes are described by graph rules. Likewise, there are a number of verification techniques, and one of them is Theorem Proving [2]. In this technique both, the system and the desired properties are described using mathematical descriptions and logic. The verification strategy consists of finding a proof from axioms and intermediary lemmas of the system. This technique is particularly interesting [3] for systems with big or infinite state space, since it does not require the construction of (any fragment of) the state space.

Previous work [4,5,6] has allowed the verification of systems specified in graph grammars through theorem proving. This technique proposed a relational and logical approach to GG, providing the coding of graphs and rules with relations. The relations that define a grammar determine axioms that can be used to develop proofs. The

* The authors gratefully acknowledge financial support received from CNPq and FAPERGS, specially under Grants, ARD 11/0764-9, PRONEM 11/2016-2 and PRONEX 10/0043-0.

J. Iyoda and L. de Moura (Eds.): SBMF 2013, LNCS 8195, pp. 147–162, 2013.

rule application is described by an event such as an inference rule (when a set of variables satisfies guard conditions, the rule is applied). This approach was translated into Event-B structures [7], allowing the use of theorem provers compatible with this language (available in the Rodin platform [8]) for demonstrations of properties. The proof process is semi-automatic, requiring user interaction.

When modelling a system in Event-B, Rodin makes a syntactic (static check) and a dynamic verification. In these verifications the tool generates proof obligations to ensure invariants are preserved, guard conditions and actions are well defined, formulas are meaningful, among others. These obligations are stated in order to ensure system consistency. Some are completed automatically, others need user intervention. The knowledge of both the system and the tool required for completing the proofs hinders the use of this proposal.

This work presents the proof obligations generated by a GG specification in Rodin, as well as establishes the provers that discharge them. Also it proposes proof strategies to assist the developers when discharging semi-automatic proof obligations generated by the specification of atomic properties in the model. Next sections are organised as follows. Section 2 introduces the graph grammar formalism. Section 3 presents the mapping of graph grammars into Event-B structures. Section 4 presents the proof obligations generated by a GG in Event-B, indicating the respective provers to discharge them. Section 5 describes strategies for discharging proof obligations generated by the specification of atomic properties. Section 6 concludes and discusses future works.

2 Graph Grammars

Graph Grammar is a specification language suitable for representing complex situations, because it is simple and visual. A graph is defined by two sets and two functions. A graph morphism is defined by two partial functions. The identifiers used in next definitions (prefixed with `inv_`, `grd_`, `axm_` and `act_`) are those used in the Event-B model (meaning respectively, invariant, guard condition, axiom and action) in Section 3.

Definition 1 (Graph and graph morphism). *A **graph** G is a tuple $(vertG, edgeG, sourceG, targetG)$, where $vertG$ is a set of vertices, $edgeG$ is a set of edges, and $sourceG, targetG\colon edgeG \to vertG$ are total functions, defining source and target of each edge, respectively. Given two graphs $G = (vertG, edgeG, sourceG, targetG)$ and $H = (vertH, edgeH, sourceH, targetH)$, a **(partial) graph morphism** $f\colon G \nrightarrow H$ is a tuple $(f\_V\colon vertG \nrightarrow vertH, f\_E\colon edgeG \nrightarrow edgeH)$ such that f commutes with source and target functions:*

$$\texttt{grd\_srctgt:}\ \forall e \in dom(f\_E) \cdot f\_V(sourceG(e)) = sourceH(f\_E(e)) \text{ and } \forall e \in dom(f\_E) \cdot f\_V(targetG(e)) = targetH(f\_E(e))$$

A graph morphism is said to be total or injective if both of its components are total or injective functions, respectively.

A typed graph is defined by two graphs connected by a total graph morphism (typing morphism). A typed graph morphism is a graph morphism that satisfies a compatibility condition, which establishes that the mapping of components must preserve types.

Definition 2 (Typed Graph, Typed Graph Morphism). *A **typed graph** is given by a tuple $G^T = (G, tG, T)$ where G and T are graphs and $tG = (tG\_V, tG\_E)$ is a typing morphism from G over T, i.e., $tG\colon G \to T$ is a* total graph morphism *(*`inv_tG_V` *and* `inv_tG_E`*). A **(typed) graph morphism from** G^T **to** H^T is defined by a morphism $g = (g\_V, g\_E)$ from G to H, s.t. the typed morphism compatibility condition is satisfied:*

$$\texttt{grd\_vertices:}\ \forall v \in dom(g\_V) \cdot tG\_V(v) = tH\_V(g\_V(v)) \ \text{ and}$$
$$\texttt{grd\_edges:}\ \ \forall e \in dom(g\_E) \cdot tG\_E(e) = tH\_E(g\_E(e))$$

A rule is composed of two typed graphs and a morphism between them, which describes a possible behaviour of the system.

Definition 3 (Rule). *A **rule** typed over T is a typed graph morphism $\alpha = (\alpha\_V, \alpha\_E)\colon L^T \rightarrowtail R^T$, where: L^T and R^T are graphs typed over T; α is injective (*`axm_alphaV` *and* `axm_alphaE`*); $\alpha\_V\colon vertL \rightarrowtail vertR$ is a total function (*`axm_alphaV`*).*

Definition 4 (Graph Grammar). *A **(typed) graph grammar** is a tuple $GG = (T, G0, R)$, where T is a graph, called type graph; $G0$ is a graph typed over T, called initial graph; and, R is a set of rules typed over T.*

The occurrence of the left-hand side (LHS) of a rule in a state graph is called match.

Definition 5 (Match). *Let $r = (\alpha\colon L^T \rightarrowtail R^T)$ be a rule, with L^T and R^T typed graphs over T. Let $G^T = (G, tG, T)$ be a typed graph with $tG = (tG\_V, tG\_E)$. A **match** m **of rule** r **in** G^T is defined by a total typed graph morphism $m = (m\_V, m\_E)\colon L^T \to G^T$, such that $m\_E\colon edgeL \rightarrowtail edgeG$ is injective.*

The behaviour of a GG is given by rule applications. A rule is applied only if there is a match of the rule in the state graph. When a rule is applied a new state is generated.

Definition 6 (Rule Application). *Let $r = (\alpha : L^T \rightarrowtail R^T)$ be a rule and $m = (m\_V, m\_E)$ be a match of r in a typed graph G^T. A **rule application** $G^T \overset{r,m}{\Rightarrow} H^T$, or the application of r to G^T at m, generates a typed graph $H^T = (H, tH, T)$, with $H = (vertH, edgeH, sourceH, targetH)$, as follows:*

- $vertH = vertG \uplus (vertR - rng(\alpha\_V))$ (`act_vert`);
- $edgeH = (edgeG - rng(m\_E)) \uplus edgeR$ (`act_edge`);
- *for all $e \in edgeH$ (*`act_src` *and* `act_tgt`*)*

$$sourceH(e) = \begin{cases} sourceG(e) & \text{if } e \in edgeG \\ \overline{m}(sourceR(e)) & \text{otherwise} \end{cases}$$

$$targetH(e) = \begin{cases} targetG(e) & \text{if } e \in edgeG \\ \overline{m}(targetR(e)) & \text{otherwise} \end{cases}$$

where $\overline{m}\colon vertR \to vertH$ is defined by:

$$\overline{m}(v) = \begin{cases} m\_V(v') & \text{if } v \in rng(\alpha\_V) \text{ and } v = \alpha\_V(v') \\ v & \text{otherwise} \end{cases}$$

– *for all* $v \in vertH$ *and all* $e \in edgeH$, $tH = (tH\_V, tH\_E)$ *is defined by (*`act_tV` *and* `act_tE`*)*

$$tH\_V(v) = \begin{cases} tG\_V(v) \text{ if } v \in vertG \\ tR\_V(v) \text{ otherwise} \end{cases}$$

$$tH\_E(e) = \begin{cases} tG\_E(v) \text{ if } e \in edgeG \\ tR\_E(v) \text{ otherwise} \end{cases}$$

Next we use the GG language to specify the Token Ring protocol [9]. In this protocol, a special signal, called a token, is passed from station to station in only one direction. A message can be transmitted only by stations that hold the token (active stations). The transmission circulates for all the ring and finishes when the message returns to the original station. Then the station that started the transmission becomes standby and the signal token is passed for the next station, restarting the cycle. In our example there is only one token, so a single station can transmit at a given time. We also allows the addition of new stations into the network at any time.

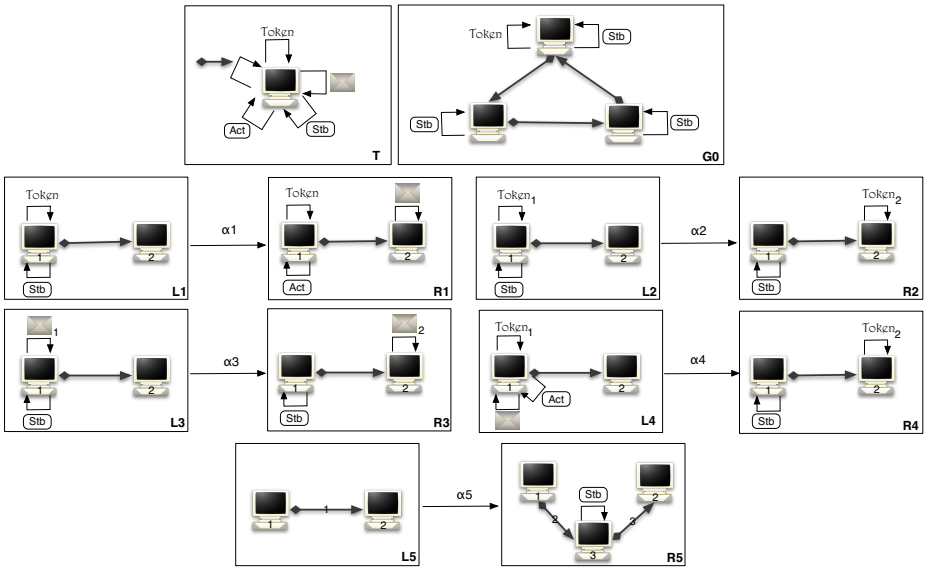


Fig. 1. Token Ring GG

Figure1 illustrates the graph grammar for the example. The type graph T defines a single type of node (Node), and five types of edges (Message), Token (Token), (Next), Act (Active Station) and Stb (Standby Station). represents a network station and defines a portion of data. The Token is a special signal which enables stations to transmit. connects each station. One station with edge Act transmits some information through the network. One station with edge Stb is standby, and can

receive a message. The initial graph $G0$ was set with three nodes, and none of the stations are transmitting (there is no edge of type (Act)).

The behaviour of this protocol is given by the rules. In this representation, the morphism is not explicitly represented, but we consider that the items with the same name and type are mapped. In the rule $\alpha 1$ a standby station that holds the token becomes active and sends a message to the next station. It is also possible that a standby station holding the token directly passes it to the next station (rule $\alpha 2$). When a message arrives at a standby node, it goes directly to the next station (rule $\alpha 3$). In rule $\alpha 4$ the message is received by the transmitting station, which returns to the standby mode and pass the token to the next station. In this specification, at each instant of time new nodes can be added into the ring, by rule $\alpha 5$. This model generates an infinite state-space.

3 Graph Grammars in Event-B

An Event-B model [10] consists of a context (static part) and a machine (dynamic part). In the context are defined sets, constants and axioms. In the machine are defined variables, invariants and events. A model is called correct if all set of proof obligations generated from the model is discharged. An extensive tool support through the Rodin Platform [8] makes Event-B especially attractive.

The Event-B model and its behaviour is similar to a graph grammar. There is a concept of state (set of variables in Event-B and a graph in GG) and a transition is considered an atomic operation in the current state (an event that updates the variables in Event-B and a rule application in a graph grammar). Each stage should preserve the properties of the state. In Event-B these properties are treated as invariants, and in graph grammars, they are related to the graph structure. A graph grammar with n rules, α_1 to α_n and $i \in \{1, ..., n\}$, can be modelled in Event-B as follows:

1. Context
   (a) The *sets* of the model are $vertT$, $edgeT$, $vertLi$, $edgeLi$, $vertRi$ and $edgeRi$ (the sets of vertices and edges of all graphs);
   (b) The *constants* represent the vertices and edges of the type graph and rules and also the names of typing functions tLi_V, tLi_E, tRi_V and tRi_E, source and target functions, $sourceT$, $targetT$, $sourceLi$, $targetLi$, $sourceRi$ and $targetRi$, and rules $alphaiV$ and $alphaiE$;
   (c) The *axioms* define explicitly all sets and functions of the model.
2. Machine
   (a) The *model variables* are specified by $vertG$, $edgeG$, $sourceG$, $targetG$, tG_V and tG_E (current state of the system);
   (b) The *invariants* define the types of variables;
   (c) The *initialisation action* sets the initial values for the variables $vertG$, $edgeG$, $sourceG$, $targetG$, tG_V and tG_E (specifying the initial graph $G0$);
   (d) The *set of events* defines the rule applications. Guard conditions guarantee the occurrence of a match (conditions for the rule to be applied).

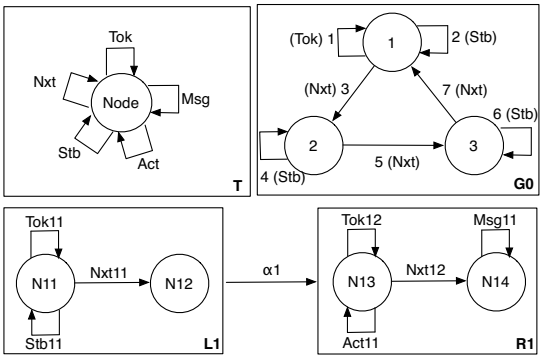


Fig. 2. Alternative Representations for Type Graph T, Initial Graph $G0$ and Rule $\alpha 1$

Static Part: The static part of graph grammars is specified in the context. Figure 2 presents an alternative representation for graph T and rule $\alpha 1$ of Figure 1.

The event-B specification is show in Figure 3. For specifying the type graph T, we define as sets, $vertT$ and $edgeT$; as constants, the names of the vertices ($Node$) and the edges (Nxt, Tok, Msg, Stb, Act), as well the names of the source and target functions ($sourceT$, $targetT$); in the axioms, we define these sets explicitly (e.g., axm_vertT is defined by $partition(vertT, \{Node\})$ meaning that $vertT = \{Node\}$).

CONTEXT ctx_T
SETS
`vertT, edgeT` // Type graph T
`vertL1, edgeL1` // Graph $L1$
`vertR1, edgeR1` // Graph $R1$
CONSTANTS
`Node Nxt Tok Msg Stb Act sourceT targetT` // Constants of type graph T
`N11 N12 Tok11 Stb11 Nxt11 sourceL1 targetL1 tL1_V, tL1_E` // Constants of graph $L1$
`N13 N14 Tok12 Act11 Nxt12 Msg11 sourceR1 targetR1 tR1_V tR1_E` // Constants of graph $R1$
`alpha1V, alpha1E` // Morphism components
AXIOMS
`axm_vertT :` $partition(vertT, \{Node\})$ // Type graph T
`axm_edgeT :` $partition(edgeT, \{Nxt\}, \{Tok\}, \{Msg\}, \{Stb\}, \{Act\})$ // Type graph T
`axm_srcTtype :` $sourceT \in edgeT \rightarrow vertT$ // Type graph T
`axn_srcTdef :` $partition(sourceT, \{Nxt \mapsto Node\}, \{Tok \mapsto Node\}, \{Msg \mapsto Node\}, \{Stb \mapsto Node\}, \{Act \mapsto Node\})$ // Type graph T
`axm_tgtTtype :` $targetT \in edgeT \rightarrow vertT$ // Type graph T
`axn_tgtTdef :` $partition(targetT, \{Nxt \mapsto Node\}, \{Tok \mapsto Node\}, \{Msg \mapsto Node\}, \{Stb \mapsto Node\}, \{Act \mapsto Node\})$ // Type graph T
`:` ...
`axm_tR1_V :` $tR1\_V \in vertR1 \rightarrow vertT$ // Typing morphism graph $R1$
`axm_tR1_V_def :` $partition(tR1\_V, \{N13 \mapsto Node\}, \{N14 \mapsto Node\})$ // Typing morphism graph $R1$
`axm_tR1_E :` $tR1\_E \in edgeR1 \rightarrow edgeT$ // Typing morphism graph $R1$
`axm_tR1_E_def :` $partition(tR1\_E, \{Tok12 \mapsto Tok\}, \{Act11 \mapsto Act\}, \{Nxt12 \mapsto Nxt\}, \{Msg11 \mapsto Msg\})$ // Typing morphism graph $R1$
`axm_alpha1V :` $alpha1V \in vertL1 \nrightarrow vertR1$ // Vertex Morphism Component from graph $L1$ to $R1$
`axm_alpha1V_def :` $partition(alpha1V, \{N11 \mapsto N13\}, \{N12 \mapsto N14\})$
`axm_alpha1E :` $alpha1E \in edgeL1 \nrightarrow edgeR1$ // Edge Morphism Component from graph $L1$ to $R1$
`axm_alpha1E_def :` $partition(alpha1E, \{Tok11 \mapsto Tok12\}, \{Nxt11 \mapsto Nxt12\})$

Fig. 3. (Part of) GG Specification in Event-B

To define a rule in Event-B, it is necessary to specify the two typed graphs and the morphism. Graphs $L1$ and $R1$ are specified analogously T. The typing morphism names $tR1\_V$ and $tR1\_E$ are declared as constants while their definitions are explicitly defined in the axioms (axm_tR1_V, $axm\_tR1\_V\_def$, axm_tR1_E, $axm\_tR1\_E\_def$). The morphism components are also declared as constants ($alpha1V$, $alpha1E$) and explicitly specified in the axioms ($axm\_alpha1V$, $axm\_alpha1V\_def$, $axm\_alpha1E$, $axm\_alpha1V\_def$).

Dynamic Part: The dynamic part of a graph grammar is specified in the machine. A set of variables define the state graph and a set of invariants determine its types. Both are illustrated in Figure 4.

MACHINE mch_trAll
SEES ctx_GG
VARIABLES
`vertG, edgeG, sourceG, targetG, tG_V, tG_E`
INVARIANTS
`inv_vertG` : $vertG \in \mathbb{P}(\mathbb{N})$
`inv_edgeG` : $edgeG \in \mathbb{P}(\mathbb{N})$
`inv_sourceG` : $sourceG \in edgeG \to vertG$
`inv_targetG` : $targetG \in edgeG \to vertG$
`inv_tG_V` : $tG\_V \in vertG \to vertT$
`inv_tG_E` : $tG\_E \in edgeG \to edgeT$
EVENTS
Initialisation
`act_vertG` : $vertG := \{1, 2, 3\}$
`act_edgeG` : $edgeG := \{1, 2, 3, 4, 5, 6, 7\}$
`act_srcG` : $sourceG := \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 2, 5 \mapsto 2, 6 \mapsto 3, 7 \mapsto 3\}$
`act_tgtG` : $targetG := \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 2, 5 \mapsto 3, 6 \mapsto 3, 7 \mapsto 1\}$
`act_tG_V` : $tG\_V := \{1 \mapsto Node, 2 \mapsto Node, 3 \mapsto Node\}$
`act_tG_E` : $tG\_E := \{1 \mapsto Tok, 2 \mapsto Stb, 3 \mapsto Nxt, 4 \mapsto Stb, 5 \mapsto Nxt, 6 \mapsto Stb, 7 \mapsto Nxt\}$
Event $alpha1 \widehat{=}$
any
$mV\ \ mE\ \ newEmsg\ \ newEact$
where
`grd_mV` : $mV \in vertL1 \to vertG$
`grd_mE` : $mE \in edgeL1 \rightarrowtail edgeG$
`grd_newEmsg` : $newEmsg \in \mathbb{N} \setminus edgeG$
`grd_newEact` : $newEact \in \mathbb{N} \setminus edgeG$
`grd_E1E2` : $newEmsg \neq newEact$
`grd_vertices` : $\forall v \cdot v \in vertL1 \Rightarrow tL1\_V(v) = tG\_V(mV(v))$
`grd_edges` : $\forall e \cdot e \in edgeL1 \Rightarrow tL1\_E(e) = tG\_E(mE(e))$
`grd_srctgt` : $\forall e \cdot e \in edgeL1 \Rightarrow mV(sourceL1(e)) = sourceG(mE(e)) \land mV(targetL1(e)) = targetG(mE(e))$
then
`act_E` : $edgeG := (edgeG \setminus \{mE(Stb11)\}) \cup \{newEmsg, newEact\}$
`act_src` : $sourceG := (\{mE(Stb11)\} \lhd sourceG) \cup \{newEact \mapsto mV(N11), newEmsg \mapsto mV(N12)\}$
`act_tgt` : $targetG := (\{mE(Stb11)\} \lhd targetG) \cup \{newEact \mapsto mV(N11), newEmsg \mapsto mV(N12)\}$
`act_tE` : $tG\_E := (\{mE(Stb11)\} \lhd tG\_E) \cup \{newEact \mapsto Act, newEmsg \mapsto Msg\}$

Fig. 4. State Graph and Events in Event-B

The initial graph and rule applications are specified by events in Event-B. Figure 2 also shows an alternative representation for the initial graph $G0$ of the token ring example. Vertices and edges are named with natural numbers with its types described into the brackets. The initialisation event, depicted in Figure 4, defines $G0$. It is responsible for initialising the value of each state variable.

Other events determine the behaviour of the system, specifying the rule applications. Figure 4 shows the specification for rule $\alpha 1$. Guard conditions guarantee the occurrence of a match, and the actions specify the value of the modified variables. If there are values for the variables mV, mE, $newEmsg$, $newEact$ satisfying the guard conditions, then the rule is applied. Guard conditions grd_mV, grd_mE, $grd\_vertices$, grd_edges and grd_srctgt guarantee that the pair (mV, mE) defines a match of the rule in the state graph. The conditions $grd\_newEmsg$, $grd\_newEact$ and grd_E1E2 ensure that $newEmsg$ and $newEact$ are new free edges names. The actions act_E, act_src, act_tgt and act_tE modify the state graph. In this case, an Stb edge is deleted and two new edges are created, one of type Act and other one of type Msg.

4 Proof Obligations Generated from a GG Specification in Rodin

When specifying a system in Event-B, the Rodin platform executes a static (syntactic) and a dynamic verification. In these verifications proof obligations are generated, which must be demonstrated in order to ensure (part of) the correctness of the model. These properties can be discharged using provers that comes with the tool or external ones, which can be installed in the form of plugins. Some of the proof obligations are proved automatically, while other ones depend on the user interaction.

Proof obligations generated by a GG specification in Rodin are basically of two kinds, to ensure well-definedness conditions (labelled with WD) or to preserve invariants (labelled with INV). In the Event-B specification, the set of variables define the state graph and the invariants specify their types. Proof obligations are generated to guarantee the preservation of their types by the initialisation event and by the events that specify the rules. The corresponding proof obligation are generated whenever a variable is modified by an event, in order to guarantee its type preservation. Other proof obligations aim to ensure that guards conditions (conditions for applying a rule) and actions (responsible for updating the values of some variables) are well-defined.

The main provers available for Rodin are NewPP, PP (predicate prover) and ML (mono-lemma). The NewPP prover has three forces. In the configuration “restricted” (nPP R), all selected hypotheses and the goal are passed to New PP. In the configuration “after lasso” (nPP with a lasso), a lasso operation is applied to the selected hypotheses and the goal and the result is passed to New PP. The lasso operation selects any unselected hypothesis that have a common symbol with the goal or a hypothesis that is currently selected. In the configuration “unrestricted” (nPP), all the available hypotheses are passed to New PP. This prover is embedded in the tool and its input language is first-order logic with the predicate $\in$. First, all function and predicate symbols that are different from $\in$ and not related to arithmetic are translated away. Then New PP translates the proof obligation to conjunctive normal form and applies a combination of unit resolution and the Davis Putnam algorithm. The PP prover, available in the Atelier-B as an external prover, also has three forces (P0, P1, PP). In the configuration “P0”, all selected hypotheses and the goal are passed to PP. In the configuration “P1”, one lasso operation is applied to the selected hypotheses and the goal and the result is passed to PP. In the configuration “PP”, all the available hypotheses are passed to PP. The input sequent is translated to classical B and fed to the PP prover of Atelier B. PP works in a

manner similar to newPP but with support for equational and arithmetic reasoning. The ML prover is also available in the Atelier-B, but different from others (PP and NewPP). ML applies a mix of forward, backward and rewriting rules in order to discharge the goal (or detect a contradiction among hypotheses). For more details see [10,7].

Table 1 presents the main proof obligations generated when specifying a GG in Event-B. They can be easily discharged by running the available provers. Besides the identification of each proof obligation, follows a brief description of it, along with the prover that must be used to discharge it.

Proof obligations identified with INITIALISATION guarantee that variables that define the state-graph, when initialised, preserve their types. The initial value of the variables describes the initial graph of the graph grammar. For instance, in the token ring example, the obligation INITIALISATION /inv/_vertG/INV is used to ensure that $\{1, 2, 3\} \in \mathbb{P}(\mathbb{N})$ (see Figure 4). Similarly, proof obligations are generated in order to ensure the type preservation by the initialisation of the other variables ($edgeG$, $sourceG$, $targetG$, tG_V and tG_E). All of them can be discharged by running P0.

Proof obligations labelled with rule/grd ensure that the guard conditions for the respective event (or rule) are well-defined. For example, in the token ring, the obligation rule1/grd_vertices/WD ensures that guard condition `grd_vertices` (see Figure 4) is well-defined, that is, $v \in dom(tL1\_V)$, $v \in dom(mV)$ and $mV(v) \in dom(tG\_V)$, with $tL1\_V$, tG_V and mV preserving its types.

The result of a rule application can create edges, delete edges or create vertices in the state-graph, changing the values of the corresponding variables. Proof obligations are generated in order to guarantee that the variables with their values updated preserve its types. These obligations are prefixed with rule/inv. In the token ring, rule 1 delete one Stb edge and create one Act and one Msg edges (see Figure 4). In such case, variables $edgeG$, $sourceG$, $targetG$ and tG_E are modified, and then proof obligations are generated to assure that these variables, after updating, preserve their types. E.g., it must be guaranteed that $(edgeG \setminus \{mE(Stb11)\}) \cup \{newEmsg, newEact\} \in \mathbb{P}(\mathbb{N})$.

Obligations are also generated to ensure that actions (which define the result of a rule application) are well-defined. These obligations are prefixed with rule/act. When a rule deletes an e edge, then variables $edgeG$, $sourceG$, $targetG$ and tG_E are modified. In such case, the deleted edge of the state-graph is the image of e by the mE component of the match, i.e., $mE(e)$ is deleted from $edgeG$. In the same way, are excluded the elements (pairs) of the functions $sourceG$, $targetG$ and tG_E that have $mE(e)$ as first component. In order to the respective actions be well-defined, the edge e must belongs to the domain of mE and mE must preserve its type (these are the proof obligations rule/act_E/WD, rule/act_src/WD, rule/act_tgt/WD, rule/act_tE/WD). Besides that, when a rule adds an edge with source (or target) in a v vertex, that is preserved by the rule, the source (respect. target) of the added edge must be image of v by the mV component of the match. In this case, in order to variables $sourceG$ and $targetG$ be well-defined, v must belong to the domain of mV, with mV preserving its type (these are the proof obligations rule/act_src/WD and rule/act_tgt/WD). Proof obligations of this type are demonstrated automatically.

The proof obligations described above are those generated when specifying a GG in Event-B. Following this approach [5], any other property to be verified must be stated as

Table 1. GG Proof Obligations in Event-B

Identification	Description	Provers
INITIALISATION/inv_vertG/INV	Ensure that $vertG$ preserves its type in $G0$ (at initialisation).	ML or P0
INITIALISATION/inv_edgeG/INV	Ensure that $edgeG$ preserves its type in $G0$.	ML or P0
INITIALISATION/inv_srcGtype/INV	Ensure that $sourceG$ preserves its type (a total function from $edgeG$ to $vertG$) in $G0$.	P0
INITIALISATION/inv_tgtGtype/INV	Ensure that $targetG$ preserves its type (a total function from $edgeG$ to $vertG$) in $G0$.	P0
INITIALISATION/inv_tG_V/INV	Ensure that tG_V preserves its type (a total function from $vertG$ to $vertT$) in $G0$.	ML, P0 or NewPP
INITIALISATION/inv_tG_E/INV	Ensure that tG_E preserves its type (a total function from $edgeG$ to $edgeT$) in $G0$.	P0
rule/grd_vertices/WD	Ensure that the `grd_vertices` condition (see Fig. 4) is well-defined, i.e. $v \in dom(tL\_V)$, $v \in dom(mV)$ and $mV(v) \in dom(tG\_V)$, with tL_V, tG_V and mV preserving its types.	ML or P1
rule/grd_edges/WD	Ensure that the `grd_edges` condition (see Fig. 4) is well-defined, i.e. $e \in dom(tL\_E)$, $e \in dom(mE)$ and $mE(e) \in dom(tG\_E)$, with tL_E, tG_E and mE preserving its types.	ML, P1 or NewPP
rule/grd_srctgt/WD	Ensure that the `grd_srctgt` condition (see Fig. 4) is well-defined, i.e. $e \in dom(sourceL)$, $sourceL(e) \in dom(mV)$, $e \in dom(mE)$, $mE(e) \in dom(sourceG)$, $e \in dom(targetL)$, $targetL(e) \in dom(mV)$ and $mE(e) \in dom(targetG)$, with $sourceL$, mV, mE, $sourceG$, $targetL$ and $targetG$ preserving its types.	ML or P1
rule/inv_vertG/INV	Ensure that $vertG$, when modified by a rule application, preserves its type.	ML or P0
rule/inv_edgeG/INV	Ensure that $edgeG$, when modified by a rule application, preserves its type.	ML or P0
rule/inv_srcGtype/INV	Ensure that $sourceG$, when modified by a rule application, preserves its type (a total function from $edgeG$ to $vertG$).	P0
rule/inv_tgtGtype/INV	Ensure that $targetG$, when modified by a rule application, preserves its type (a total function from $edgeG$ to $vertG$).	P0
rule/inv_tG_V/INV	Ensure that tG_V, when modified by a rule application, preserves its type (a total function from $vertG$ to $vertT$).	ML, P1 or NewPP
rule/inv_tG_E/INV	Ensure that tG_E, when modified by a rule application, preserves its type (a total function from $edgeG$ to $edgeT$).	P0
rule/act_E/WD	Ensure that the modification of variable $edgeG$ is well-defined. When an e edge is deleted, ensure that e belongs to the domain of the mE component of the match (i.e $e \in dom(mE)$) and that mE preserves its type (mE: $edgeL \rightarrow edgeG$).	Automatic
rule/act_src/WD	Ensure that the modification of variable $sourceG$ is well-defined. When an e edge is deleted, ensure that e belongs to the domain of the mE component of the match (i.e $e \in dom(mE)$) and that mE preserves its type (mE: $edgeL \rightarrow edgeG$). When an e edge is created with source (or target) in a vertex v, preserved by the rule, ensure that v belongs to the domain of the mV component of the match (i.e $v \in dom(mV)$) and that mV preserves its type (mV: $vertL \rightarrow vertG$).	Automatic
rule/act_tgt/WD	Ensure that the modification of variable $targetG$ is well-defined. When an e edge is deleted, ensure that e belongs to the domain of the mE component of the match (i.e $e \in dom(mE)$) and that mE preserves its type (mE: $edgeL \rightarrow edgeG$). When an e edge is created with source (or target) in a vertex v, preserved by the rule, ensure that v belongs to the domain of the mV component of the match (i.e $v \in dom(mV)$) and that mV preserves its type (mV: $vertL \rightarrow vertG$).	Automatic
rule/act_tE/WD	Ensure that the modification of variable tG_E is well-defined. When an e edge is deleted, ensure that e belongs to the domain of the mE component of the match (i.e $e \in dom(mE)$) and that mE preserves its type (mE: $edgeL \rightarrow edgeG$).	Automatic

an invariant, indicating that it must be true for all reachable states of the system. Proofs for such properties are developed by induction: in the base case, a proof obligation is generated to guarantee the preservation of the property for the initial graph and, at the inductive step, a proof obligation is generated for the graph resulting from the application of each rule of the grammar. In general, the discharging of such proof obligations requires intervention from the user, that must have knowledge of both, the tool and the specification. The proposal of proof strategies to help the user in the development of the demonstrations for some of these properties is addressed in the next section.

5 Proof Strategies for Atomic Properties

The translation of GG in Event-B structures has enabled the use of first-order logic to express properties of reachable states of a graph grammar. However, during the development of the case studies, we noticed that, although the specification of the behaviour of the system could be rather intuitively described with graph grammars, the verification of properties was not trivial. Properties over states are properties over graphs, typically composed of different kinds of edges and vertices. In previous work [11] we have proposed patterns for the presentation, codification and reuse of property specifications. Here, we presents proof strategies for the demonstration of specific atomic properties belonging to such patterns. Particularly, we describe proof strategies for discharging the properties presented in Figure 5. Properties must be stated as invariants in the machine.

INVARIANTS

`propFin` : $finite(tG\_E \rhd \{t\})$ // The set of edges of type t of a reachable graph is finite.
`propCard` : $card(tG\_E \rhd \{t\}) = 1$ // Any reachable graph has exactly one edge of type t.
`propExEdge` : $\exists x \cdot x \in tG\_E \rhd \{t\}$ // Any reachable graph has an edge of type t.
`propExVert` : $\exists x \cdot x \in tG\_V \rhd \{t\}$ // Any reachable graph has a vertex of type t.

Fig. 5. Properties as Invariants in Event-B

For each property, we first present the steps for discharging the proof obligation for the initial graph and after for the rules. Property `propFin` is required for the discharging of `propCard`. The steps for discharging the INITIALISATION/propFin/INV generated by `propFin` $finite(tG\_E \rhd \{t\})$ for the initial graph are the following:

1. Add the hypothesis $tG\_E \rhd \{t\} = \{x\}$, replacing tG_E by its value and considering x the result of tG_E restricted to the type t for the initial graph.
2. Execute the prover PP in force P1.
3. Run prover ML.

Figure 6 presents the proof tree[1] generated for the demonstration of the proof obligation INITIALISATION/propFin/INV. Each node represents a sequent and each number (from 1 to 5) represents the rule or the prover used to discharge the corresponding sequent. A set of proof tactics and rewriting rules are available within the Rodin platform [7]. Space limitations prohibit their detailing here. After adding the hypothesis three

[1] The set of hypotheses H in proof trees are omitted. In order to provide readability we denote H different sets of hypotheses.

sequents must be proved: (i) $\vdash \top$, that is discharged automatically with the $\top$ goal rule; (ii) $\vdash tG\_E \rhd \{t\} = \{x\}$ which is automatically simplified (through sl/ds, that corresponds to a selection/deselection of hypotheses) to the sequent $H \vdash tG\_E \rhd \{t\} = \{x\}$, discharged with P1; (iii) $tG\_E \rhd \{t\} = \{x\} \vdash finite(tG\_E \rhd \{t\})$ which is discharged with ML.

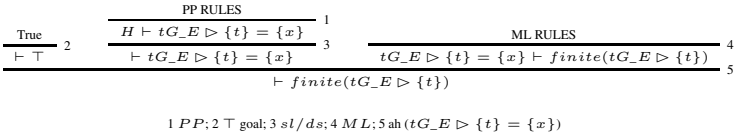


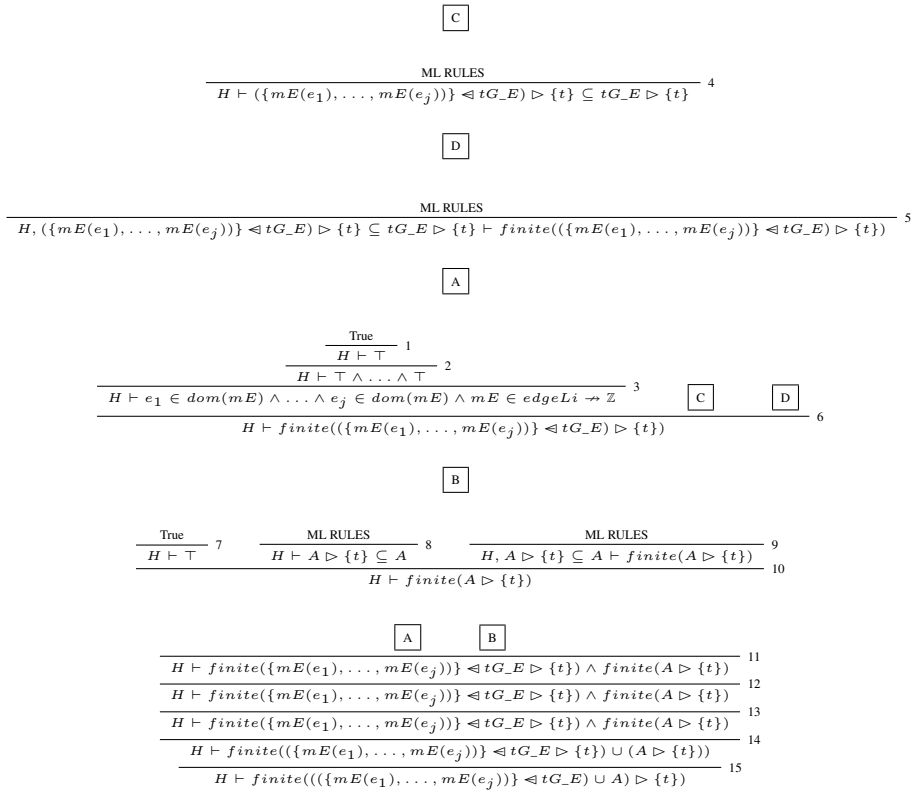
Fig. 6. Proof Tree for Discharging INITIALISATION/propFin/INV

In order to conclude the proof of `propFin`, proof obligations must be discharged for each rule of the graph grammar that modifies tG_E. These will be those that replace tG_E by its new value, determined by the action of the respective rule. In general, a rule can both delete and create new edges, then the obligation to be discharged will be of the form $finite((((\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \cup A) \rhd \{t\})$, considering that j edges are deleted and a set of A pairs are included in tG_E. In this case, the steps for discharging rule_i/propFin/INV for each rule i are the following:

1. Apply the tactic Range Distribution Left Rewrites, which after the application of some automatic tactics will generate two sequents to be proved: (i) $finite((\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \rhd \{t\})$ and (ii) $finite(A \rhd \{t\})$.
2. In order to prove (i), add $(\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \rhd \{t\} \subseteq tG\_E \rhd \{t\}$ as hypothesis, and conclude the subgoals running ML.
3. In order to prove (ii), add $A \rhd \{t\} \subseteq A$ as hypothesis, and conclude the subgoals running ML.

Figure 7 presents the proof tree generated for the demonstration of each proof obligation rule_i/propFin/INV. After applying the tactic range distribution left rewrites in goal, a sequence of automatic tactics are applied (rules 12 to 14 in the proof tree). They correspond to the applications of simplification rewriting rules and typing rewriter tactic (details are found in Rodin Proof Tactics [7]). Then, the tactic $\wedge$ goal splits the sequent into two subgoals: (i) $finite((\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \rhd \{t\})$ and (ii) $finite(A \rhd \{t\})$. The subgoal (i) is discharged adding the hypothesis $(\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \rhd \{t\} \subseteq tG\_E \rhd \{t\}$, remaining three sequents to be proved: $H \vdash e_1 \in dom(mE) \wedge \ldots \wedge e_j \in dom(mE) \wedge mE \in edgeLi \nrightarrow \mathbb{Z}$, that is discharged automatically, $H \vdash (\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \rhd \{t\} \subseteq tG\_E \rhd \{t\}$ and $H \vdash finite((\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \rhd \{t\})$, which are both discharged running ML. The subgoal (ii) is discharged adding the hypothesis $A \rhd \{t\} \subseteq A$, remaining three sequents to be proved: $H \vdash \top$, discharged with the $\top$ goal tactic, $H \vdash A \rhd \{t\} \subseteq A$ and $H \vdash finite(A \rhd \{t\})$, both discharged with ML.

The steps for discharging the obligation INITIALISATION/propCard/INV generated by `propCard` ($card(tG\_E \rhd \{t\}) = 1$) for the initial graph are the following:



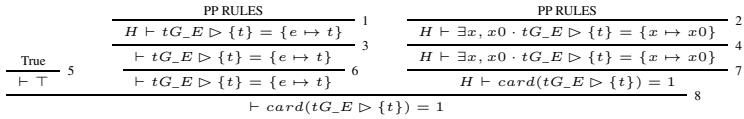
1 $\top$ goal; 2 Simplification Rewrites ; 3 Generalised MP; 4 ML; 5 ML; 6 ah $((\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \rhd \{t\} \subseteq tG\_E \rhd \{t\})$; 7 $\top$ goal; 8 ML; 9 ML; 10 ah $(A \rhd \{t\} \subseteq A)$; 11 $\wedge$ goal; 12 Simp. rewrites; 13 Type Rewrites; 14 Simp. Rewrites; 15 Range Distribution Left Rewrites in Goal

Fig. 7. Proof Tree for Discharging rule_i/propFin/INV

1. Add the hypothesis $tG\_E \rhd \{t\} = \{e \mapsto t\}$, replacing tG_E by its initial value and considering $e \mapsto t$ the pair resultant of tG_E restricted to the type t for the initial graph.
2. Run prover PP in force P1 (lasso operation is applied to the common hypotheses).
3. Run prover PP in force P1.

Figure 8 presents the proof tree. After adding the hypothesis, three sequents must be proved: (i) $\vdash \top$, that is discharged automatically with the $\top$ goal rule; (ii) $tG\_E \rhd \{t\} = \{e \mapsto t\}$ and (iii) $\exists x, x0.\ tG\_E \rhd \{t\} = \{x \mapsto x0\}$, both discharged with P1.

The obligations generated by `propCard` for each rule will be those that replace tG_E by its new value. Since a rule can both delete and create new edges, then the general obligation to be discharged will be $card((((\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \cup \{ed_1 \mapsto t_1, \ldots, ed_k \mapsto t_k\}) \rhd \{t\}) = 1$, considering that j edges are deleted and k edges are created. In fact, if this property is valid, the t edge or is preserved or is deleted and created by a rule application. Then, we divide our tactic into two subcases:



1 PP; 2 PP; 3 sl/ds; 4 sl/ds; 5 $\top$ goal; 6 simplification rewrites; 7 simplification rewrites; 8 ah($tG\_E \rhd \{t\} = \{e \mapsto t\}$)

Fig. 8. Proof Tree for Discharging INITIALISATION/propCard/INV

The t edge is preserved: 1. Apply the default post-tactics, which simplifies the property to $\exists x, x0.((\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \cup \{ed_1 \mapsto t_1, \ldots, ed_k \mapsto t_k\} \rhd \{t\}) = \{x \mapsto x0\}$.

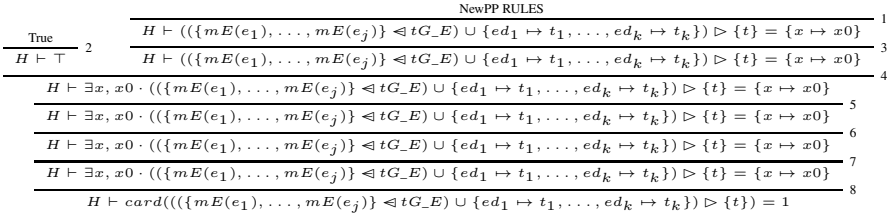
2. Instantiate variables in goal with x, $x0$, converting the goal to $(\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \cup \{ed_1 \mapsto t_1, \ldots, ed_k \mapsto t_k\}) \rhd \{t\} = \{x \mapsto x0\}$.
3. Run NewPP with lasso.

   Figure 9 presents the generated proof tree.

The t edge is deleted and a t edge is created: 1. Add $card(\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E \rhd \{t\}) = 0$ as hypothesis, which will generate three sub-goals to be proved: (i) $finite(\{mE(e_1), \ldots, mE(e_j)\} \lhd tG\_E \rhd \{t\})$; (ii) $\{mE(e_1), \ldots, mE(e_j)\} \lhd tG\_E \rhd \{t\} = \varnothing$; and (iii) $\exists x, x0.((\{mE(e_1), \ldots, mE(e_j)\}) \lhd tG\_E) \cup \{ed_1 \mapsto t_1, \ldots, ed_k \mapsto t_k\} \rhd \{t\}) = \{x \mapsto x0\}$.

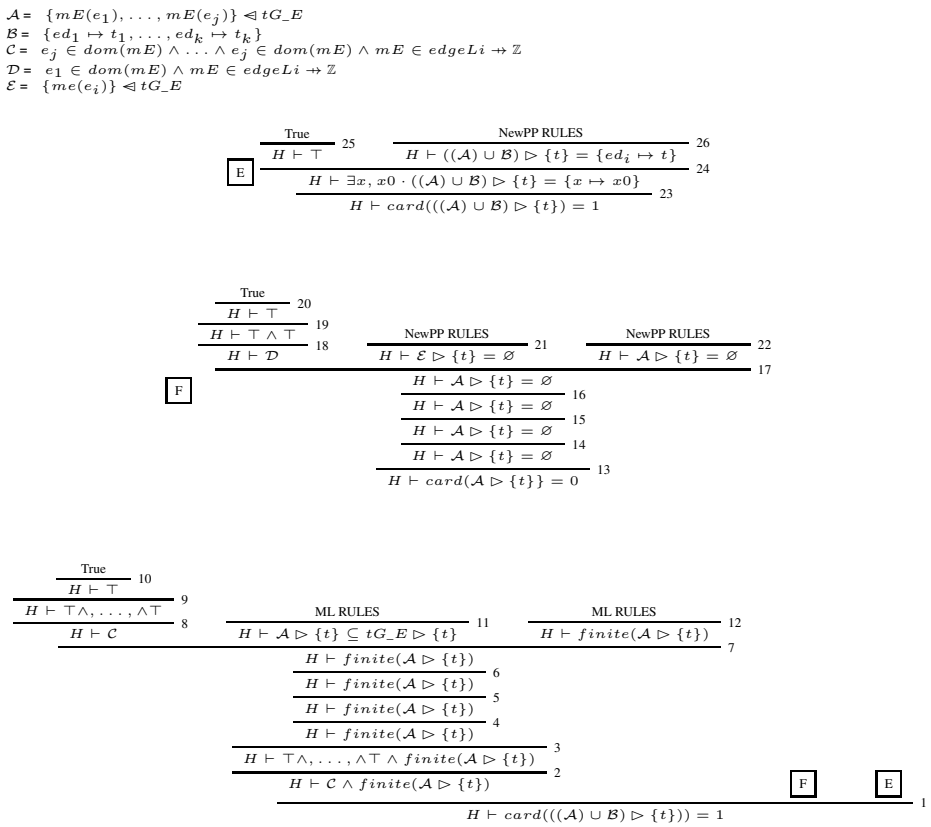
2. In order to proof (i), add $\{mE(e_1), \ldots, mE(e_j)\} \lhd tG\_E \rhd \{t\} \subseteq tG\_E \rhd \{t\}$ as hypothesis, and conclude the sub-goals running ML.
3. In order to proof (ii), add $\{mE(e_i)\} \lhd tG\_E \rhd \{t\} = \varnothing$ as hypothesis, such that e_i is the t deleted edge, and discharge the sub-goals running NewPP with lasso.
4. In order to proof (iii), instantiate the variable of the existential quantifier with ed_i and t, such that ed_i is the added t edge, and discharge the sub-goal with NewPP with lasso.

   Figure 10 presents the generated proof tree.



1 $NewPP$; 2 $\top$ goal; 3 sl/ds; 4 $\exists$ goal (inst x, $x0$); 5 $\exists$ hyp ($\exists x, x0 \cdot tG\_E \rhd \{t\} = \{x \mapsto x0)\}$); 6 simplification rewrites; 7 type rewrites; 8 simplification rewrites

Fig. 9. Proof Tree for Discharging rule_i/propCard/INV



1 ah ($card(\{mE(e_1), \ldots, mE(e_j)\} \lhd tG\_E \rhd \{t\}) = 0$); 2 generalised MP; 3 simplification rewrites; 4 type rewrites; 5 simplification rewrites; 6 ∃ hyp ($\exists x, x0 \cdot tG\_E \rhd \{t\} = \{x \mapsto x0\}$); 7 ah($\mathcal{A} \rhd \{t\} \subseteq tG\_E \rhd \{t\}$); 8 generalised MP; 9 simplification rewrites; 10 ⊤ goal; 11 ML; 12 ML; 13 simplification rewrites; 14 type rewrites; 15 simplification rewrites; 16 ∃ hyp ($\exists x, x0 \cdot tG\_E \rhd \{t\} = \{x \mapsto x0\}$); 17 ah ($\mathcal{E} \rhd \{t\} = \varnothing$); 18 generalised MP; 19 simplification rewrites; 20 ⊤ goal; 21 NewPP; 22 NewPP; 23 simplification rewrites; 24 ∃ goal (inst ed_i, t); 25 ⊤ goal; 26 NewPP

Fig. 10. Proof Tree for Discharging rule_i/propCard/INV

In order to discharge the proof obligation INITIALISATION/propExEdge/INV generated by property `propExEdge` ($\exists x \cdot x \in tG\_E \rhd \{t\}$) for the initial graph just run NewPP. Again, since a rule can preserve, delete and create edges, then we divide our proof strategies for obligations rule_i/propExEdge/INV in three cases.

All t edges are preserved: Run NewPP with lasso.

An t edges is created: (a) Instantiate existential variable in goal with $ed_i \mapsto t$, such that ed_i is the t edge that is created. (b) Run ML.

An t edge is deleted, but an t edge is preserved (a) Instantiate existential variable in goal with $mE(ed_i) \mapsto t$, such that ed_i is the t edge that is preserved. (b) Run NewPP with lasso.

In order to discharge the proof obligation INITIALISATION/propExVert/INV generated by property `propExVert` ($\exists x \cdot x \in tG\_V \rhd \{t\}$) for the initial graph just run

NewPP. For such property no proof obligations are generated for rules. This is because the component that map vertices in rules are total and injective, and then vertices can not be deleted. Proving that we have a vertex of type t in the initial graph, no other rule can delete it. Previous work has addressed that this restriction in the model is not a severe limitation for many practical applications [6].

6 Conclusions and Future Work

In this paper we presented the proof obligations generated by Rodin platform when specifying a graph grammar system in Event-B structures, indicating the strategies to discharge them. We also propose strategies of proofs for the verification of some atomic properties, declared as invariants in the model.

One of the disadvantages of using theorem proving as verification technique is that it requires user interaction during the development of the proofs, but on the other hand, it allows the verification of systems with huge or infinite state spaces. This work constitutes the first step towards the reduction of expertise required from the user when adopting such an approach. Strategies for discharging other kind of properties are being proposed. Particularly, tactics for all patterns proposed in [11] are under development. We are also investigating to which extent the theory of refinement, which is well-developed in Event-B, could be used to validate a stepwise development based on graph grammars.

References

1. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Handbook of graph grammars and computing by graph transformation, pp. 247–312. World Scientific Publishing Co., Inc., River Edge (1997)
2. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning (in 2 volumes). Elsevier and MIT Press (2001)
3. de Mello, A.M., Junior, L.C.L., Foss, L., da Costa Cavalheiro, S.A.: Graph grammars: A comparison between verification methods. In: WEIT, pp. 88–94 (2011)
4. da Costa, S.A., Ribeiro, L.: Verification of graph grammars using a logical approach. Sci. Comput. Program. 77(4), 480–504 (2012)
5. Ribeiro, L., Dotti, F.L., da Costa, S.A., Dillenburg, F.C.: Towards theorem proving graph grammars using Event-B. ECEASST 30 (2010)
6. da Costa, S.A.: Relational approach of graph grammars. PhD thesis, UFRGS, Brazil (2010)
7. DEPLOY: Event-B and the rodin platform (Mai 2013), `http://www.event-b.org/` (last accessed Mai 2013)
8. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer (STTT) 12(6), 447–466 (2010)
9. Tanenbaum, A.: Computer Networks, 4th edn. Prentice Hall Professional Technical Reference (2002)
10. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York (2010)
11. da Costa Cavalheiro, S.A., Foss, L., Ribeiro, L.: Specification patterns for properties over reachable states of graph grammars. In: Gheyi, R., Naumann, D. (eds.) SBMF 2012. LNCS, vol. 7498, pp. 83–98. Springer, Heidelberg (2012)