Theorem Proving Graph Grammars with Attributes and Negative Application Conditions

Simone A. C. Cavalheiro[b,*], Luciana Foss[b], Leila Ribeiro[a]

[a]*Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Brazil*
[b]*Technology Development Center, Federal University of Pelotas (UFPel), Brazil*

Abstract

Graph grammars may be used to formally describe computational systems, modeling the states as graphs and the possible state changes as rules (whose left- and right-hand sides are graphs). The behavior of the system is defined by the application of these rules to the state-graphs. From a practical point of view, the extension of rules to enable description of extra conditions that must be satisfied upon rule application is highly desirable. An example is the specification of *negative application conditions*, or NACs, that describe situations that prevent the application of a rule. This extension of the basic formalism enhances the expressiveness of rules, generally allowing simpler specifications. Another extension that is fundamental for practical applications is the possibility to use data types, like natural numbers, lists, etc., as attributes of graphical elements (vertices and edges). Attributed graph grammars are well-investigated and used. However, there is a lack of verification techniques for this kind of grammar mainly due to the fact that data types are typically infinite domains, and thus techniques like model checking can not be used directly (without abstraction constructions). The present work provides a theoretical foundation for theorem proving graph grammars with negative application conditions and attributes. This is achieved by generating an event-B model from a graph grammar. Event-B models are composed by sets and axioms to define types, and by states and events to describe behavior. After constructing the event-B model that is semantically equivalent to a graph grammar, properties about reachable states may be proven using the various theorem provers available for event-B in the Rodin platform. This strategy allows the verification of systems with infinite-state spaces without using any kind of approximation.

Keywords: Graph transformation, Theorem proving

1. Introduction

Graph Grammars (or Graph Transformation Systems) [27] are well-suited for the formal specification of applications in which states have a complex topology (involving not only many types of elements but also different types of relations between them) and in which behavior is essentially data-driven, that is, events are triggered by particular configurations of the state. States are modeled by graphs and state changes, or events, by rules (that transform graphs). Many reactive systems are examples of this class of applications, like protocols for distributed and mobile systems, biological systems, and many others. Additionally to the complex states and reactive behavior, concurrency and non-determinism play an essential role in this area of applications: many events may happen concurrently, if they all are enabled, and the choice of occurrence between conflicting events is non-deterministic. Originally, graph grammar rules specified only patterns that must be present to trigger rules, in [42] an extension was proposed allowing rules to specify also negative application conditions (short NACs), that are patterns that hinder the application of the rule. Such application conditions are commonly used in non-trivial specifications because the number of rules required to specify a system might be greatly reduced. The use of values defined by abstract data types [31] as attributes of vertices and edges complements the approach. Attributed graph grammars allow the use of (elements of) data types like

*Corresponding author
Email addresses: simone.costa@inf.ufpel.br (Simone A. C. Cavalheiro), lfoss@inf.ufpel.edu.br (Luciana Foss), leila@inf.ufrgs.br (Leila Ribeiro)

natural numbers, lists, etc., as well as variables and terms in rules. Most practical applications using graph grammars are based on approaches with attributes and NACs.

To check whether a system specified using graph grammars has the desired properties, we may use verification techniques. There are various approaches available for graph grammars, some of them are reviewed in Section 2. With static analysis we may typically find problems related to typing, but also analyze whether the potential conflicts and dependencies among rules are the expected ones for the system being modeled (using critical pair analysis), or even check whether it is possible that specific sequences of rule applications generate desired results (by constructing concurrent rules). But certainly, correctness of behavior in a more general sense can not be ensured statically. Model checking techniques inspect computations of a system to verify whether the presented behavior, given by the application of the rules, corresponds to the expected one (described, for example, as formulas in some logic). The number of computations of a system depends on the number of different choices of behavior that are available at each state, and this, in turn, depends on the number of rules of the system but also on the number of different ways in which each rule may be applied in the state. Even with only one rule there may be a huge (or even infinite) number of choices if the state graph is large and/or the rule has a variable ranging over an infinite type domain. One of the strengths of graph grammars is the ability to model complex states (as graphs), possibly using attributes from general data types, and therefore it is to expect that even small specifications indeed induce a huge number of complex reachable graphs. This means that analysis techniques that involve building the whole (or even parts of) state-space of a system, like model checking, are of limited use for graph grammars with NACs and attributes. Since these grammars are the ones that are mostly used in practical applications, we started in [21, 22] investigating the use of the theorem proving technique in this context.

In order to prove properties of a system using theorem proving, we must encode the system (states and behavior) and the properties to be proven in some logic, and then use deduction to verify that the system satisfies the properties. This encoding of the system, in our case represented by a graph grammar, is a critical task, since we must ensure that it preserves the semantics of the system. Moreover, the encoding must be also useful, in the sense that it enables to express and reason about the desired properties in a suitable way. For example, consider the property *"there is always exactly one node of a specific type in any state"*. This kind of property is at a level of abstraction in which we need to be able to inspect the components of the state (vertices and edges of a graph) to verify it. This means that the encoding of the system into the logic must be such that it allows us to easily reason about such components. We may be interested in many kind of properties, like properties of computations and/or properties of reachable states. Again, since the possibility to represent complex states (as graphs) is one of the strengths of the graph grammar formalism, proving properties about reachable states is especially important. In our previous work we introduced a relational approach to graph grammars, providing an encoding of graphs and rules into relations, that allowed to generate an event-B [4] model corresponding to a graph grammar. In event-B, models are composed of states, which are sets of variables, and events, that specify possible changes of values of state variables. Abstract data types may be defined as types for variables. Events may have complex guards. The encoding of graph grammars in event-B was claimed to be equivalent to the single-pushout approach to graph grammars [30], and was inspired by Courcelle's research about logic and graphs [18]. Properties about the reachable states of the system could then be stated as *invariants*, using First-Order Logic with Set Theory, and proven using theorem provers available for event-B specifications [20, 70].

In this paper we completely formalize this translation, prove the preservation of behavior and extend the approach to deal with graph grammars with negative application conditions and attributes. In [22] we defined the theoretical foundations of the logical description of graph grammars with attributes, which is the basis for the work developed here in encoding these grammars as event-B models to allow formal verification. First we formally define the encodings for the Single- and Double-Pushout (SPO and DPO) graph grammar approaches including NACs, proving that the encoding preserves the semantics. We also show that, since the DPO does not allow rule applications that would have side effects, the encoding of DPO-rules can be simplified (because it is possible to determine which elements will be the deleted/created just by analyzing the rule). Then we provide an extension to consider (the translation of) attributes. The encodings of SPO and DPO with NACs and attributes are done at the set-theoretical level because this makes expressing and proving properties easier. Moreover, these encodings may be used as basis for the development of graph grammar tools since most tools actually work at the set-theoretic level to build the result of rule applications, and we prove the correctness of the proposed encodings with respect to the corresponding SPO and DPO categorical definitions.

The paper starts with a discussion about verification of graph grammars (Sect. 2), contextualizing and motivating

the proposed work. Section 3 defines the mathematical notation used throughout the paper and briefly introduces the event-B formalism. Section 4 presents a definition of graph grammars with NACs together with a working example, as well as a set-theoretic construction for rule application. The translation of graph grammars with NACs to event-B is described in Sect. 5 and the extention of the approach to deal with attributes is presented in Sect. 6. Section 7 shows how the proposed approach can be used to verify properties of infinite-state graph grammars and Sect. 8 contains a discussion about the obtained results. Final remarks can be found in Sect. 9. The reader that is not interested in reading in details how the translation is defined and the semantical compatibility is proven, but rather on how to use the approach for verification may skip in a first reading Sects. 5 and 6.3 and jump to Sect. 7 after understanding the definitions of event-B and graph grammars (note that some knowledge of the translation is needed in order to be able to assist the proof constructions, if necessary).

2. Verification of Graph Grammars

There are essentially three different ways to formally analyze graph grammars (or graph transformation systems): through model-checking, which basically explores the state-space directly; using static analysis techniques, which examine the structure of the rules; and applying theorem proving, which uses logic inference to ensure that desired properties are satisfied. Model checking is attractive because verification may be performed automatically, but it has the drawback of being restricted to finite (and non-huge) state-space systems. Theorem proving may be applied to analyze infinite state systems, but proofs are typically done interactively, requiring some proof skills from the user. Static analysis allows, besides type checking, to analyze the potential (but not the actual) behavior of a graph grammar (e.g., via critical pairs analysis and the construction of concurrent rules).

In the following we describe some approaches that allow the formal analysis of graph transformation systems. We do not aim to provide an exhaustive description, but to present relevant works on each kind of verification approach. We first present some widespread development environments for graph transformation systems, highlighting the analysis methods offered by each of them.

AGG [32] is a development environment for attributed graph transformation systems based on the algebraic approach to graph transformation (SPO and DPO approaches). The specification language supports typed graphs including inheritance, and rules with negative application conditions. AGG offers critical pair analysis, consistency checking and termination criteria verification for layered graph transformation systems.

Groove [69, 36] is a general-purpose graph transformation toolset offering an automatic exploration and analysis of the complete state space of reachable states. Groove implements the SPO rewriting approach but it also allows configurations to simulate the DPO approach. It supports the use of typed graphs, negative application conditions (NACs) and handles attributed graphs (attributes are edges pointing to special data nodes that represent the data values). The analysis capabilities include LTL/CTL model checking and Prolog queries. Through model checking, it is possible to explore properties of computations, that is, properties over the sequences of steps a system may engage in. The integration of a Prolog interpreter into the Groove simulator [34] allows a combined verification of static (structural) properties with dynamic ones.

VIATRA (VIsual Automated model TRAnsformations) [19] is a framework for verification and validation of model transformations described by UML diagrams, which integrates Graph Transformation and Abstract State Machines. In this approach, graph transformation rules (using a graph pattern concept) are aggregated into more complex models by abstract state machine rules, which provide a set of imperative control structures. Moreover, VIATRA supports transformation rules with path expressions or multi-objects, and negative application conditions. Model checking can be used to verify properties of UML diagrams by translating the diagrams into Promela [1]. UML statecharts are translated into concurrent objects in Promela by using VIATRA rules to obtain an Extended Hierarchical Automata (EHA) model that is post-processed to generate the corresponding Promela code. The verification results are available in the SPIN environment as message sequence charts and traces. Syntactic correctness and completeness also can be verified by planner algorithms [78], checking that the target model of the transformation is a well-formed model in the target language. Besides, semantic correctness of transformations is being verified through the translation of the transformation rules into the SAL intermediate language [12], which allows using automated symbolic analysis tools in order to verify if some dynamic property is satisfied.

Henshin [7] is an Eclipse plug-in for Eclipse Modeling Framework (EMF) model transformation. In this tool, graph transformation concepts are transferred to model transformations in the EMF. The object based modeling per-

formed by EMF is conceptually similar to specifications based on typed attributed graphs [33]. The toolset supports arbitrary mixing of different graph transformation approaches (SPO and DPO) and includes state space generation tools. The specification language supports positive and negative application conditions and rule nesting. The state space tools allow structural state invariant checking using OCL [79], qualitative model checking using CADP [35] and mCRL2 [39] and probabilistic model checking using PRISM [52]. Also, conflicts, dependencies, termination and confluence of transformations can be analyzed using AGG [15].

A common strategy that has been applied to specific approaches of graph transformation systems is the translation of the models into input languages of established tools. For instance, [25] proposed a translation of Object-Based Graph Grammars (OBGG) to Promela (the input language of the SPIN model checker). Using this approach, properties based on states and computations are described using LTL and are verified by model checking. The verification of properties based on states can only be done for systems with a static number of objects. This approach allows formulating properties considering only the OBGG specification, without the need to know the structure of the translated Promela code. The counter-examples are interpreted as OBGG derivations and are graphically represented. This translation was proven to preserve the semantics of the original graph grammar. Other approaches have been proposed to provide model checking analysis to graph transformation systems with large state spaces. Yousefian et al. [2] developed an heuristic approach based on genetic algorithms to find error states. The approach is implemented in GROOVE in order to detect deadlocks, but liveness properties may also be checked with minor changes in the set of rules.

In [10], an approach for the verification of infinite-state graph transformation was proposed. This approach is based on under- and over-approximations that allow to construct finite structures approximating system behavior, enabling automated proofs of certain properties of infinite-state systems. Approximations chains are obtained by means of the unfolding construction limited to a finite causal depth k. All valid properties in an under-approximation (k-truncation) are also valid in the complete system behavior. In this case, some liveness properties of the complete system may be checked, e.g., properties of the form "eventually A". On the other hand, valid properties of an over-approximation (k-covering) may not be valid in the complete system, but all valid properties of the complete system are also valid in the over-approximation. Properties of the form "Always A" may be checked through k-coverings. The proposed framework considers the DPO approach of graph transformation, restricted to injective matches on edges and non-deleting node rules. Contextual Petri nets are used to decrease the size of the unfoldings.The Augur tool supports this approach and uses converters in order to visualize rules and Petri graphs. Further, it allows to extract Petri nets that can be analyzed by Petri net analysis tools.

Other authors have investigated the analysis of graph transformation systems based on (relational) logic or set theory. Alloy is a simple structural modeling language, based on First-Order Logic. In [11], a methodology to analyze graph transformations by using the AGG tool and then translating the graph grammars to Alloy models was proposed. Graphs are modeled as signatures and rules as predicates. Rules consider negative application conditions, which are described by constraints in the rule predicate. In this approach properties about reachable configurations or applicability of rule sequences can be verified. Alloy does not address infinite state spaces and deals with the state explosion by requiring the definition of a maximum cardinality of the worlds under analysis.

Strecker [72] has proposed a formalization of graph transformations in a set-theoretic model using Isabelle/HOL [57]. He sketched the definition of a language for writing graph transformation programs and reasoning about them, focusing on the proof of structural properties. The language is composed of statements (e.g., a statement to apply rules repeatedly to a graph), distinguishing between success and failure states. The approach replaces the match occurrence (i.e., the applicability condition of a rule) by a formula over the graph structure, constructed in a fragment of First-Order Logic. Tran and Percebois [76] also used a relational and logical representation of graph grammars in Isabelle to verify graph transformation systems. The aim is to statically verify if a transformation preserves a particular property of a state graph, that is, to investigate general conditions for a graph grammar to preserve structural properties of rule applications (considering one derivation step). Percebois, Strecker and Tran [73, 74, 63] have also addressed the problem of locality in graph rewriting. Particularly, they analyze which properties hold if a transformation is applied to an arbitrary graph (maybe satisfying some given preconditions), focusing on (connectedness and reachability) properties involving transitive closure of edge relations of a graph. Basically, they show that verification of the preservation of reachability in a graph can be reduced to a simple "calculational" proof requiring the computation of the transitive closure of a finite, concrete set of edges (those occurring in the applicability condition of the transformation rule).

Habel, Pennemann and Rensink [43, 40, 8] have investigated the use of weakest preconditions and strongest

post-conditions for showing the correctness of programs (and transformation systems). A standard method is to construct a weakest precondition of the program relative to the post-condition and to show that the precondition implies the weakest precondition. In [43] they formally define weakest preconditions for high-level programs and give a transformation of programs and post-conditions into weakest preconditions. In [62], a correct calculus for proving graph conditions was proposed. In [61], a satisfiability algorithm was presented, proven to be correct and complete, and a decidable fragment of conditions was identified. The advantage is that the approach is general since it provides a method to verify finite- and infinite-state systems. The disadvantage is that finding invariants is hard and cannot be automated in general.

Plump et al. [65, 66, 56] have worked in the design and implementation of Graph Programs (GP), which is an experimental nondeterministic programming language for high-level problem solving in the domain of graphs. The language is based on conditional rule schemata (or "attributed" rules) for graph transformation. Graph transformation in GP is based on the DPO approach with relabeling. Poskitt and Plump have been investigating the verification of GP. In [67] a Hoare-style proof system for verifying the partial correctness of (a subset of) GP was defined. The pre- and post-conditions of the calculus are nested graph conditions with expressions, a formalism for specifying both structural graph properties and properties of labels. In [68] new proof rules and termination functions were added, which allow to guarantee that program executions always terminate (weak total correctness), or that programs always terminate and do so without failure (total correctness). More recently, a unification algorithm [46] has been proposed for a static confluence analysis of GP.

Our aim here is to enable the use of theorem provers to aid the verification of graph grammars with NACs and attributes, providing a very powerful means to check properties for this formalism. It is not our aim to build a theorem prover from scratch, but use existing ones. Thus, we started with a general notion of graph grammar with very few restrictions (injective rules) and investigated how to translate it to the input language of some theorem prover. In [20, 70, 22] we have shown how to encode the structures and behavior of graph grammars using the event-B language. In this paper, we completely formalize this approach and prove that this translation preserves the semantics of the graph grammars, an essential task since we propose to use provers available for event-B to verify properties of the corresponding graph grammars. None of the discussed approaches is suitable for proving general properties of reachable states (expressed in first order logic with set theory) of graph grammars with attributes and negative application conditions through theorem proving. More details about the kind of properties that can be proven in different approaches can be found in Sect. 9.

3. Event-B and its Mathematical Notation

Event-B [5, 4] is a state-based formalism closely related to Classical B [3] and Action Systems [9]. In this section, we present the definitions of this formalism that are used in this paper. Moreover, we define the symbols and operations used in the remainder of the paper. Table 1 shows these symbols and corresponding meanings. This corresponds to the standard event-B mathematical notation.

An event-B model consists of two parts: a static part called *context* and a dynamic part called *machine*. Event-B is based on *First-Order Logic with Set Theory*.

Definition 1 (Event-B Model, Event). *An **event-B model** $EBModel = (\mathcal{C}, \mathcal{M})$ is defined by a **context** $\mathcal{C} = (c, s, A)$ and a **machine** $\mathcal{M} = (v, I, init, E)$, where c and s are sets of constant and set names, respectively; $A(c, s)$ is a collection of axioms constraining c and s; v is a set of state variables; $I(v)$ is a model invariant limiting the possible assignments to v s.t. $\exists c, s, v \cdot A(c, s) \wedge I(v)$ - i.e. A and I characterize a non-empty set of model states; $init(v')$ is an initialization action assigning initial values to the model variables[1]; and E is a set of model* events*. An **event** is a tuple $e = (G, BA)$ where $G(v)$ is a formula, called the guard, and $BA(v, v')$ is a before-after predicate[2] Types of constants must be defined as axioms in A, whereas types of variables must be defined as invariants in I.*

[1] v' is the set of primed variables of v, corresponding to the values of these variables after the execution of an event.

[2] The before-after predicate is a binary relation that defines a relation between current and next states. It is expressed in terms of the state variables. By convention, the before values are denoted by unprimed variables (of v) and after values are denoted by primed variables (of v').

Table 1: Definition of symbols and operations

Symbol/Operation	Meaning
$\nrightarrow$	Partial functions or morphisms
$\rightarrow$	Total functions or morphisms
$\rightarrowtail$	Partial and injective functions or morphisms
$\rightarrowtail$	Total and injective functions or morphisms
$f \circ g$	Composition of functions or morphisms f and g
f^{-1}	Inverse function
$\mapsto$	Mapping relation
$\uplus$	Disjoint union
$\mathbb{N}$	Set of natural numbers
$\mathbb{P}$	The set of all subsets (power set)
$\setminus$ or $-$	Set difference
$rng(r)$	Range of a binary relation r
$dom(r)$	Domain of a binary relation r
$card(A)$	Number of elements of set A
$r[A]$	$\{y \mid \exists x.x \in A \text{ and } x \mapsto y \in r\}$ (Relational image)
$\lhd$	$A \lhd r \stackrel{def}{=} \{(a,b) \in r \mid a \notin A\}$ (Domain subtraction)
$\lhd\!\!+$	$r \lhd\!\!+ s \stackrel{def}{=} (dom(s) \lhd r) \cup s$ (Relation overriding)
$\rhd$	$r \rhd B \stackrel{def}{=} \{(a,b) \in r \mid b \in B\}$ (Range restriction)

In the following, we will use the concrete syntax of the event-B Camille editor (from the Rodin platform), that is shown in Fig. 1. In the context, set and constant names are defined, and arbitrary axioms may be used, the only requirement is that types of constants must be defined as axioms. All variables must have a type, defined by an invariant. Invariants may be used to describe the desired properties of reachable states of a system. The initialization event does not have guards, and must assign a value for each variable of the machine (these assignments may be non-deterministic, but this feature is not used in this paper since we have a concrete initial graph in a graph grammar). To define the other events that describe the behavior of the system, we may use auxiliary variables (in the any block), that must be typed in the guards of the event (where block). The guards are also used to specify the conditions that must hold in a state for the event to be enabled. Finally, the then block implements the before-after predicate: it is used to assign the new values to variables (not all variables must be changed in an event, the values of the ones that are not listed remain unchanged).

Model correctness is demonstrated by generating and discharging a collection of proof obligations that ensure that the initial state is feasible and satisfies all invariants of the model and, if any event may be executed at a state that satisfies all invariants, it will result in a state that also satisfies all invariants. Variant properties may also be specified, but are out of the scope of this paper.

Definition 2 (Model Correctness). *Given an event-B model $EBModel = (\mathcal{C}, \mathcal{M})$, with $\mathcal{M} = (v, I, init, E)$, and an event $ev = (G,BA) \in E$ or $ev = init$. The event ev is correct if the following conditions are satisfied:*

1. *Feasibility* (FIS): $\begin{cases} I(v) \wedge G(v) \Rightarrow \exists v' \cdot BA(v,v') & \text{, if } ev \in E \\ \exists v' \cdot init(v') & \text{, if } ev = init \end{cases}$
2. *Invariant Satisfaction* (INV): $\begin{cases} I(v) \wedge G(v) \wedge BA(v,v') \Rightarrow I(v') & \text{, if } ev \in E \\ I(v') & \text{, if } ev = init \end{cases}$

An event-B model is correct if all its events are correct.

The feasibility condition (FIS) states that whenever the invariants and the guard of an event are true in some state v, it is possible to obtain a state v' that satisfies the before-after predicate of this event. To ensure the feasibility of the

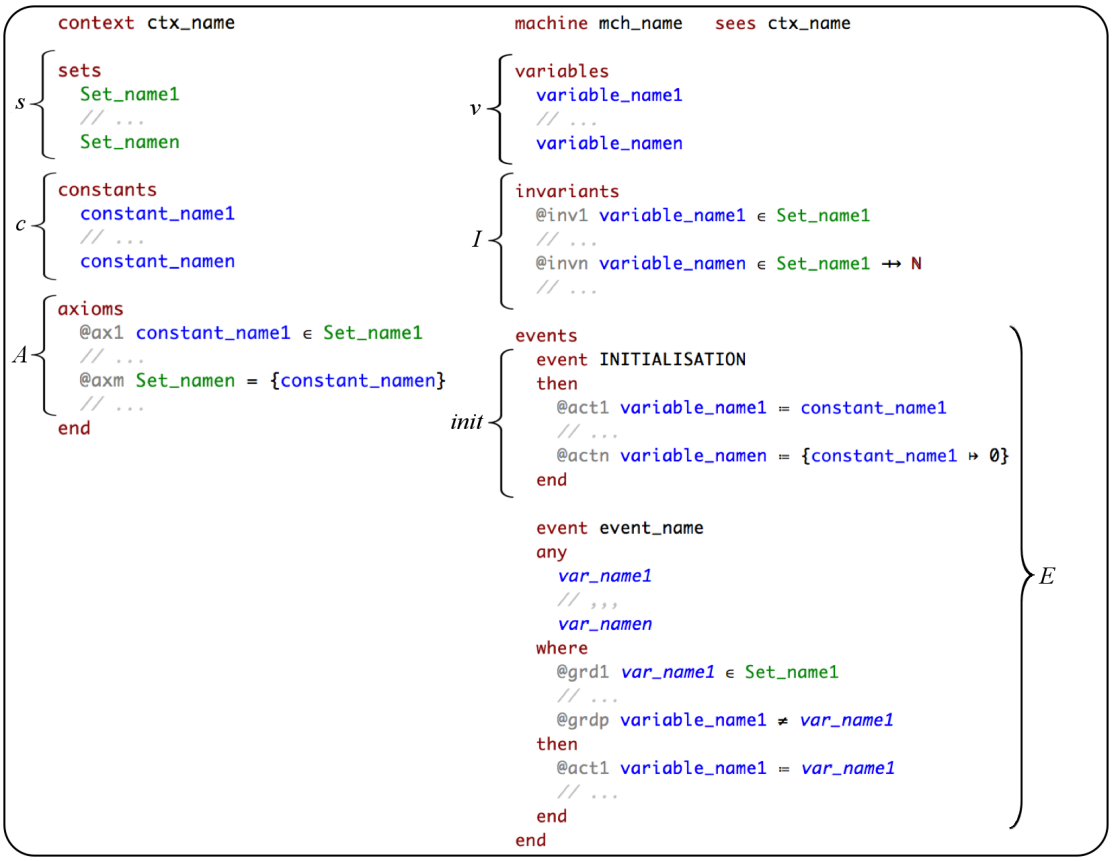


Figure 1: EventB syntax example (Camille Editor)

system, all sets in a context are assumed to be non-empty in an event-B model (sets represent types, and thus empty types are not allowed). The invariant satisfaction condition (INV) ensures that if the event occurs, it brings the system to a state v' that satisfies all model invariants. Properties that a model should exhibit are described as invariants, and thus proving that a model is correct means proving that each event does not bring a system to a state in which the desired properties do not hold. Given a model that is correct, the behavior of an event-B model is defined by a transition system, as described below.

Definition 3 (Event-B Model Behavior). *Given a correct model* $EBModel = (\mathcal{C}, \mathcal{M})$*, with* $\mathcal{C} = (c, s, A)$ *and* $\mathcal{M} = (v, I, init, E)$*, its behavior is given by a transition system* $BST = (BState, BS_0, \rightarrow)$ *where:* $BState = \{\langle v \rangle | v$ *are state variables and* $\langle v \rangle$ *is a valuation of* $v\} \cup \{Undef\}$*,* $BS_0 = Undef$*, and* $\rightarrow \subseteq BState \times BState$ *is the transition relation given by the rules:*

$$\text{start}\ \frac{init(v')}{Undef \rightarrow \langle v' \rangle}$$

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

The model is initialized (rule $start$) to a state described by $init(v')$ and then, as long as there is an enabled event (rule $transition$), the model may evolve by firing this event and computing the next state according to the event's before-after predicate. Events are atomic. In case more than one event is enabled at a certain state, demonic choice semantics applies. Note that model correctness implies that all reachable states satisfy the invariants of the model.

Refinement is a fundamental concept in event-B, since the idea is to construct a model for a system in steps, starting from a very abstract model and refining this model until it includes requirements and is at the desired level

of abstraction (that makes implementation straightforward). The refinement of an abstract model $\mathcal{M}^A$ is a concrete model $\mathcal{M}^C$ that is behaviorally related to the abstract one. In event-B, relating the models is achieved by constructing a refinement mapping between $\mathcal{M}^C$ and $\mathcal{M}^A$ and by discharging a number of refinement proof obligations. These proof obligations ensure that all computations that are possible at the concrete level were also possible at the abstract level, i.e., no new behavior (with respect to the variables that compose the abstract state) was introduced in the concrete model. When a concrete model is constructed as a refinement of an abstract model, it is not necessary to redo any proof to guarantee that the concrete model satisfies the properties stated as invariants of the abstract model.

A refinement $\mathcal{C}^C$ of a context $\mathcal{C}^A$ is obtained by adding new sets, constants and axioms. A concrete machine, which sees a concrete context $\mathcal{C}^C$, can use all concrete sets and constants, as well as the abstract ones (in $\mathcal{C}^A$). In a machine refinement, the set of variables of a concrete machine $\mathcal{M}^C$ must be completely disjoint from the collection variables of the corresponding abstract machine $\mathcal{M}^A$. In this paper, we will only consider the simplest version of refinement in event-B, that does not involve data refinement and in which concrete events may only extend the original ones, in the sense that they may introduce new guards and assignments to the newly introduced variables, but may not change the assignments to the abstract variables. With this kind of refinement no additional proof obligation is required to guarantee that all properties that were satisfied by the abstract model are also satisfied by the concrete model. We will construct an event-B model that corresponds to a graph grammar and define the attribute part as a refinement (where the concrete events extend the abstract ones). Thus, the attribute layer is constructed on top of the structural layer, and all properties (of reachable states) that are valid for a graph grammar without attributes will remain true when we add attributes to this grammar.

An extensive tool support through the Rodin Platform makes event-B especially attractive [24]. 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 theorem proving supported by a collection of theorem provers, but there is also some support for model checking[3].

4. Graph Grammars with Negative Application Conditions

Graph-based formal description techniques are a friendly means of explaining complex situations in a compact and understandable way. 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 of 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. Graph rules are used to capture the dynamical aspects of the systems. That is, from the initial state of the system (the initial graph), the application of rules successively changes the system state.

In this section, basic concepts of the algebraic Single and Double Pushout Approaches for graph grammars are presented [30, 16]. These approaches are based on graphs and graph morphisms. Graphs are structures composed of vertices and edges, which allow the description of complex situations in a visual, compact, clear, and intuitive way. In this paper we use directed edges, therefore the source and target of each edge must be defined. Graph morphisms are used in order to relate graphs, they map all elements of one graph into the corresponding elements of another graph. This mapping must preserve the source and target of each edge, i.e., if an edge e_1 is mapped to an edge e_2, the source and target vertices of e_1 must be accordingly mapped to the source and target of e_2.

Throughout the rest of the text we use special labels for formulae in order to make the relationship between a graph grammar and the corresponding event-B model clearer. The labels have the following forms: `grd_lbl`, `inv_lbl`, `axm_lbl`, `act_lbl` and `prop_lbl`, where the label prefixes identify guards, invariants, axioms, actions and properties, respectively.

Definition 4 (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, i.e.*

[3] See Rodin Platform http://www.event-b.org/. Rodin Development is supported by European Union ICT Projects DEPLOY (2008 to 2012) and RODIN (2004 to 2007).

$$\forall e \in dom(f_E) \cdot f_V(sourceG(e)) = sourceH(f_E(e)) \text{ and}$$
$$\forall e \in dom(f_E) \cdot f_V(targetG(e)) = targetH(f_E(e)) \qquad \text{(grd\_srctgt)}$$

A graph morphism is said to be total, injective or surjective if both components are total, injective or surjective functions, respectively. The category of typed graphs and (partial) typed graph morphisms is denoted by $\mathbf{GraphP}$*, the full subcategory containing only total graph morphism is denoted by* $\mathbf{Graph}$*.*

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

Definition 5 (Typed Graph, Typed Graph Morphism). *Given a type graph T, a* ***(T-)typed graph*** *is given by a tuple $G^T = (G, tG, T)$ where G is a graph and $tG = (tG_V, tG_E)$ is a typing morphism from G over T, i.e., $tG\colon G \to H$ is a* total graph morphism *(*$\texttt{inv\_tG}_\texttt{V}$ *and* $\texttt{inv\_tG}_\texttt{E}$*). A* ***(T-typed) graph morphism from*** *G^T* ***to*** *H^T is defined by a graph morphism $g = (g_V, g_E)$ from G to H, such that the typed morphism compatibility condition is satisfied:*

$$\forall v \in dom(g_V) \cdot tG_V(v) = tH_V(g_V(v)) \text{ and} \qquad \text{(grd\_tV)}$$
$$\forall e \in dom(g_E) \cdot tG_E(e) = tH_E(g_E(e)) \qquad \text{(grd\_tE)}$$

The category of typed graphs and partial typed graph morphisms is denoted by $\mathbf{TGraphP}(\mathbf{T})$, where composition and identities are defined componentwise. The full subcategory containing all typed graphs and total morphisms is called $\mathbf{TGraph}(\mathbf{T})$. In the following we will often omit the type graph superscript when it is clear form the context, that is, for a T-typed graph we will write G instead of G^T.

Now we define rules. In the SPO approach, a rule is just an injective partial graph morphism, where items that are not mapped are meant to be deleted, items that are mapped are preserved and items that are in the right-hand side and not in the left-hand side of the rule are created. The same can be expressed by a span of total injective morphisms in the DPO approach. The use of injective morphisms implies that rules are not allowed to merge or to duplicate items.

Definition 6 (Rule Morphism/Span). *Given a type graph T, an* ***SPO-rule morphism*** *is an injective T-typed graph morphism $\alpha = (\alpha_V, \alpha_E)\colon\ L \rightarrowtail R$. A* ***DPO-rule span*** *is a span of injective total T-typed graph morphisms $(\alpha L, \alpha R)$, with $\alpha L : K \rightarrowtail L$ and $\alpha R : K \rightarrowtail R$. In both cases, L is called left-hand side (LHS) and R right-hand side (RHS) of the rule.*

SPO-rule morphism $: (L \overset{\alpha}{\rightarrowtail} R)$ ***DPO-rule span*** $: (L \overset{\alpha L}{\hookleftarrow} K \overset{\alpha R}{\rightarrowtail} R)$

Given a rule α as above we define the following sets:

Deleted vertices of *L:* $RuleDel_V^\alpha = VertL \setminus dom(\alpha_V)$*, in case α is SPO,*
or $RuleDel_V^\alpha = VertL \setminus rng(\alpha L_V)$*, in case α is DPO;*

Deleted edges of *L:* $RuleDel_E^\alpha = EdgeL \setminus dom(\alpha_E)$*, in case α is SPO,*
or $RuleDel_E^\alpha = EdgeL \setminus rng(\alpha L_E)$*, in case α is DPO;*

Preserved vertices of *L:* $RulePreserv_V^\alpha = VertL \setminus RuleDel_V^\alpha$*;*

Preserved edges of *L:* $RulePreserv_E^\alpha = EdgeL \setminus RuleDel_E^\alpha$*;*

Created vertices of *R:* $RuleCreate_V^\alpha = VertR \setminus rng(\alpha_V)$*, in case α is SPO,*
or $RuleCreate_V^\alpha = VertR \setminus rng(\alpha R_V)$*, in case α is DPO;*

Created edges of *R:* $RuleCreate_E^\alpha = EdgeR \setminus rng(\alpha_E)$*, in case α is SPO,*
or $RuleCreate_E^\alpha = EdgeR \setminus rng(\alpha R_E)$*, in case α is DPO*

Since rules morphisms/spans are injective, in the following, when convenient, we will assume, without loss of generality, that α_L and α_R are inclusions, in case of DPO, and that $\alpha(i) = i$ for all $i \in dom(\alpha)$, in case of SPO.

Intuitively, a rule describes that, whenever an image of its left-hand side is found in the graph representing the state, a copy of the rule's right-hand side may replace this image. Thus, the left-hand side (or LHS) of a rule defines the items that must be present in the state to enable this rule's application. Furthermore, it is usually very convenient to be able to define also a forbidden context that prevents rule application. This is called a Negative Application Condition, or NAC [42]. NACs are defined in the same way for SPO and DPO rules: by embedding the left-hand side of the rule in the forbidden context. Then, if there is an image of this graph consisting of the LHS plus the forbidden context in the state, the rule is not applicable. A NAC may also forbid rule applications in which two different elements of the LHS of a rule are mapped to the same in the state graph. We will call the pair (LHS, NAC) of a rule as its *rule application pattern*, since these two components define the patterns that must be present/absent for rule application.

Definition 7 (Rule with Negative Application Conditions, Rule (Application) Pattern). *A **rule with negative application conditions**, or just **rule**, typed over T is a pair $r = (\alpha, AN(\alpha))$ consisting of a rule morphism/span α (SPO/DPO) with LHS L and a finite set of negative application conditions $AN(\alpha) \subseteq \mathcal{MOR}(L)$, where $\mathcal{MOR}(L)$ denotes the set of all total typed graph morphisms from L to graphs typed over T that are not isomorphisms*[4].

*A **rule (application) pattern** is the pair $(L, AN(\alpha))$ representing the items that must be present/absent for the rule to be applied. We define the following sets for each $NACj \in AN(\alpha)$ defined by a morphism $lj : L \to L^{NAC_j}$:*

***Forbidden vertices*:** $NACj_V = VertL^{NACj} \setminus lj_V[VertL]$

***Forbidden edges*:** $NACj_E = EdgeL^{NACj} \setminus lj_E[EdgeL]$

***Forbidden identifications of vertices*:** $NACj_{idV} = \{(v1, v2) | \{v1, v2\} \subseteq VertL \text{ and } lj_V(v1) = lj_V(v2)\}$

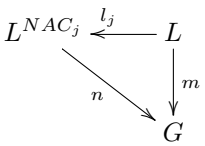
***Forbidden identifications of edges*:** $NACj_{idE} = \{(e1, e2) | \{e1, e2\} \subseteq EdgeL \text{ and } lj_E(e1) = lj_E(e2)\}$

A graph grammar is composed of a *type graph*, an *initial graph* and a *set of rules*.

Definition 8 (Graph Grammar with NACs). *A **(typed) graph grammar with NACs** is a tuple $GG = (T, G0, R)$, such that T is the type graph of the grammar, $G0$ is a graph typed over T, called initial graph, and R is a set of rules with negative application conditions typed over T.*

To apply a rule in a graph G we must ensure that G contains an image of the LHS of the rule and does not contain an image of the forbidden context (described by the NACs of the rule). When this is the case, we say that the NAC is satisfied. There are different notions of NAC satisfaction. The following definition describes *partial injective satisfaction*, which is frequently required in practical applications. This definition is equivalent to the one proposed in [44, 45], allowing arbitrary mappings from the LHS of a rule to a graph G, but requiring that forbidden items are mapped injectively. After the definition we discuss other types of satisfaction.

Definition 9 (NAC Satisfaction). *Let $r = (\alpha, AN(\alpha))$ be a T-typed rule, with left-hand side L. Let G be a T-typed graph , m be a total typed graph morphism $m = (m_V, m_E) \colon L \to G$ and $lj \in AN(\alpha)$ be a NAC of r. Morphism m **(partially injectively) satisfies NAC** $lj = (lj_V, lj_E) \colon L \to L^{NAC_j}$ if it satisfies the NAC satisfaction condition:*



[4] In the original definition of [42], although isomorphisms are allowed as NACs, they are classified as *inconsistent* NACs because they actually prevent the rule of being applied (when the forbidden context is empty, it is found in any graph where a match exists and thus the rule is never applicable). Since these NACs are useless, we explicitly forbid them.

There is no total graph morphism $n = (n_V, n_E)$: $L^{NAC_j} \to G$ *such that*

(injV) $lj_V[VertL] \triangleleft n_V : NACj_V \rightarrowtail VertG \setminus m_V[VertL]$,

(injE) $lj_E[EdgeL] \triangleleft n_E : NACj_E \rightarrowtail EdgeG \setminus m_E[EdgeL]$, *and*

(comm) $n_V \circ lj_V = m_V$ *and* $n_E \circ lj_E = m_E$.

(grd_NACj)

A morphism m ***satisfies all NACs of a rule*** *if it satisfies each individual NAC of the rule.*

In the definition above, conditions (injV) and (injE) of (grd_NACj) ensure that all forbidden items of L^{NAC_j} will be mapped injectively to items that are not in the image of the match m: n restricted to all items that are not in the image of lj must yield total and injective functions from the sets of forbidden elements to items of G that are not image of the match. Other items may be mapped arbitrarily. Definitions of NAC satisfaction depend on whether it is possible that (i) different forbidden items are mapped to the same item in G; (ii) a forbidden item is mapped to an item that is image of the match in G and (iii) non-injective matches imply satisfaction of NACs with injective ljs. The definition above rules out situations (i), (ii) and (iii). The original definition of NAC satisfaction [42] did not require (injV) and (injE): n is a general total morphism, allowing (i) and (ii) to occur in a match that satisfies a NAC. In the same paper, it was shown that restricting n to be injective would give raise to a more suitable NAC satisfaction notion, called *injective NAC satisfaction* but this would imply that any non-injective match would immediately satisfy all NACs which map L injectively in L^{NAC_j} (because it would be impossible to find an injective n that comutes with m and lj – situation (iii) above). Although it is possible to use injective satisfaction and complete the set of NACs of a rule adding NACs to prevent the application of the rule in all situations corresponding to possible non-injective matches of the rule [44], in practice the user usually wants to specify one NAC to deal with many different possible cases of rule application. Therefore, in the rest of the paper, we will only consider partial injective satisfaction. A categorical definition of this notion was proposed in [44] and [45] (called NAC Schemata). The idea is to construct an object by gluing all elements of L^{NAC_j} that are identified by m (this is done via an epi-mono factorization of m followed by a pushout) and then requiring the non-existence of an injective morphism n starting from this object and a commutativity condition analogous to (comm) of (grd_NACj). This construction ensures that the NAC is only violated when n maps (images of) elements from LHS according to m and forbidden elements injectively, which is exactly what is stated in (grd_NACj). Changing the definition above to general or injective satisfaction is straightforward: one just has to remove the restrictions on n ((injV) and (injE) of (grd_NACj)) or change them to require injective n_V and n_E.

Note, however, that this definition of NAC does not describe concretely which are the prohibited elements that prevent the rule to be applied: this prohibition is hidden in the non-existence of a morphism that commutes with the NAC morphism and the match. Using an abstract definition is interesting for theoretical investigations about graph transformations but it makes verification of concrete graph grammar models more difficult because the properties that are verified typically involve concrete elements of the state. For example, assume one wants to prove property P: *there is at most one edge of a type* Act *in any reachable graph*. Now consider a rule that creates one Act edge and has a NAC forbidding its application if there is already an edge of type Act in the graph. This rule certainly preserves property P: whenever it is applied in a graph that has property P, it will result in another graph that also satisfies P. But how can we show this just based on the fact that, additionally to a morphism from the LHS of the rule to the state graph, *(a) there is no morphism from the NAC to the state graph that commutes with the match*? Of course it is possible, but it would be much easier to reason if we start from the fact that *(b) there is no* Act *edge in the state graph*. When encoding these facts in a theorem prover, (b) follows from (a), but it makes a great difference in the proving effort if we could assume (b) directly. The question is thus whether one can present the NAC in a different (but semantically equivalent) way such that the encoding in event-B will lead to a set of facts that will ease the construction of proofs. It is indeed possible to rewrite (grdNACj) to deal concretely with the forbidden elements and forbidden identifications. This is done in Def. 10. We then show in Theo. 11 that both descriptions for NAC satisfaction condition are equivalent. Definition 10 actually makes explicit the possible reasons for the non-existence of the morphism n in (grd_NACj) of Def. 9. These are : (1) the elements forbidden by the NAC are not present or (2) the identifications forbidden by the NAC are not made (by the match). Note that we have an *or* between these two conditions because if any of this is true, the NAC is satisfied (assuming that the sets of forbidden elements and

forbidden identifications are not empty). If any of these sets is empty, the corresponding requirements ((nac1) to (nac10) below) are simply omitted from (grd_NAC′).

Definition 10 (Set Theoretic NAC Satisfaction). *Let $r = (\alpha, AN(\alpha))$ be a T-typed rule, with left-hand side L. Let G be a T-typed graph , m be a total typed graph morphism $m = (m_V, m_E)\colon L \to G$ and $lj \in AN(\alpha)$ be a NAC of r. Morphism m* ***(partially injectively) satisfies NAC*** *$lj = (lj_V, lj_E)\colon L \to L^{NACj}$ if it satisfies the disjunction:*

$$\begin{array}{l} \text{(noAsg)}\ \textit{There are no total assignments } asg_V\colon NACj_V \to VertG \textit{ and} \\ \qquad asg_E\colon NACj_E \to EdgeG \textit{ such that (nac1) to (nac8) are satisfied;} \\ \quad \textit{or} \\ \text{(noIdent)(nac9) or (nac10) are satisfied.} \end{array} \qquad \text{(grd\_NACj′)}$$

where

(nac1) $asg_V[NACj_V] \subseteq VertG \setminus m_V[VertL]$

(nac2) $asg_E[NACj_E] \subseteq EdgeG \setminus m_E[EdgeL]$

(nac3) asg_V *is an injective total function*

(nac4) asg_E *is an injective total function*

(nac5) $\forall v \in NACj_V . tG_V(asg_V(v)) = tL_V^{NACj}(v)$

(nac6) $\forall e \in NACj_E . tG_E(asg_E(e)) = tL_E^{NACj}(e)$

(nac7) $\forall e \in NACj_E . \begin{cases} sourceG(asg_E(e)) = asg_V(v) & \textit{, if } sourceL^{NAC_j}(e) = v \textit{ and } v \in NACj_V \\ sourceG(asg_E(e)) = m_V(v) & \textit{, otherwise, with } v = lj_V^{-1}(sourceL^{NAC_j}(e)) \end{cases}$

(nac8) $\forall e \in NACj_E . \begin{cases} targetG(asg_E(e)) = asg_V(v) & \textit{, if } targetL^{NAC_j}(e) = v \textit{ and } v \in NACj_V \\ targetG(asg_E(e)) = m_V(v) & \textit{, otherwise, with } v = lj_V^{-1}(targetL^{NAC_j}(e)) \end{cases}$

(nac9) $\bigvee_{(v1,v2)\in NACj_{idV}} m_V(v1) \neq m_V(v2)$

(nac10) $\bigvee_{(e1,e2)\in NACj_{idE}} m_V(e1) \neq m_V(e2)$

Theorem 11. *The NAC satisfaction conditions defined in Def. 9 and in Def. 10 are equivalent.*

PROOF. We have to prove that, whenever there guard (grd_NACj) is true, guard (grd_NACj′) is also true, and vice versa. Since these guards involve negative conditions, we will prove it by proving the contrapositive: whenever one is false, the other is also false. Note that, since (grd_NACj′) is a disjunction, we actually have to prove that $not(\texttt{grd\_NACj}) = not(\texttt{noAsg})$ *and* $not(\texttt{noIdent})$.

The NAC morphism is not an isomorphism by definition (see Def. 7). Therefore it must be non injective, non surjective, or both. Assume lj is not injective, that is, the NAC specifies forbidden identifications. In this case there must be elements in sets $NACj_{idV}$ or $NACj_{idE}$ (see Def. 7). Then if a match identifies all elements from these sets (violating (nac9) and (nac10)), there is a morphism from L^{NACj} to G that satisfies the commutativity condition of (grd_NACj) (one just have to map the item in lj that identifies two items of L to the corresponding item in G that identifies the same elements of L). Now, if lj is not surjective, there must be forbidden vertices, edges or both:
⇐: Assume (grd_NACj′) is false. Then there are asg_V and asg_E that make (nac1) to (nac8) true. We now analyze the consequence of each of these formulas being true:

(nac1) induces a mapping, say $n_V : vertL^{NACj} \to VertG$ defined for each $v \in VertL^{NACj}$ by

$$\begin{cases} n_V(v) = asg_V(v) & \text{, if } v \in NACj_V \\ n_V(v) = m_V \circ lj_V^{-1}(v) & \textit{, otherwise} \end{cases}$$

By construction, $n_V \circ lj_V = m_V$. This means that the commutativity condition (comm) of (grd_NACj) holds for vertices. Note that (nac1) also ensures that forbidden items are not mapped to the image of the match m_V.

(nac2) is analogous for the edges, making commutativity condition of (grd_NACj) true for edges.

(nac3) **and** (nac4) ensure that forbidden items are mapped injectively, and thus, together with (nac1) and (nac2), impliy that (injV) and (injE) are satisfied.

(nac5) ensures that n_V is type compatible: for vertices that are not forbidden, the types are the same as in $VertL$ because lj is a typed morphism. Thus, n_V must preserve these types. For vertices that are in $NACj_V$, (nac5) ensures that they have the same type in $VertL^{NACj}$ and $VertG$.

(nac6) ensures the same for edges, thus allowing to conclude that n_E is compatible with types.

(nac7) **and** (nac8) are analogous for source and target, and thus n is a total typed graph morphism.

Therefore, we conclude that (grd_NACj) is also false, because the mapping n is a total typed graph morphism that makes the diagram in Def. 9 commute (and this morphism is injective on forbidden items).
$\Rightarrow$: Assume (grd_NACj) is false. Then there is a morphism $n = (n_V, n_E) : L^{NACj} \to G$ that is a typed graph morphism and $n_V \circ lj_V = m_V \wedge n_E \circ lj_E = m_E$ and all forbidden elements are mapped injectively to items that are not image of the match. Now we just have to use n to define the assignments asg_V and asg_E. Finding these sets makes equations (nac1) to (nac4) true. The fact that n is a typed graph morphism makes (nac5) to (nac8) true. Thus, (grd_NACj′) must be false. ■

Now we define some sets that will be useful to construct the result of a rule application. Given a graph G and a rule with negative application conditions whose LHS has an image in G, we describe which are the sets of vertices and edges that will be deleted/preserved/created by this rule's application. The sets Del_V and Del_E have as elements the vertices and edges that are images of items specified for deletion in the rule. The set $Dangling$ contains all edges that will be deleted as a side effect of deleting some vertex of the graph, but whose deletion is not explicitly specified by the rule (dangling condition, see Def. 13 below), that is, edges that are not in the image of the LHS of the rule but that will have to be deleted because their source and/or target vertices are deleted. $Preserv_V$ contains all vertices that will be preserved by the rule application, and New_V and New_E have all items that shall be created by the rule application[5] (note that, in the case of edges, only edges whose source and target vertices are new or preserved can be created).

Definition 12 (Deleted, Preserved, Created Items). *Let $r = (\alpha, AN(\alpha))$ be a T-typed rule with left-hand side L, right-hand side R and m be a total morphism $m = (m_V, m_E) : L \to G$. Then we define the following sets:*

***Deleted vertices of** G:* $Del_V^{\alpha,m} = m_V[RuleDel_V^\alpha]$

***Deleted edges of** G:* $Del_E^{\alpha,m} = m_E[RuleDel_E^\alpha]$

***Dangling edges of** G:* $Dangling^{\alpha,m} = (NoSourceVertex \cup NoTargetVertex) \setminus Del_E^{\alpha,m}$, *where*[6] $NoSourceVertex = dom(sourceG \rhd Del_V^{\alpha,m})$ *and* $NoTargetVertex = dom(targetG \rhd Del_V^{\alpha,m})$

***Preserved vertices of** G:* $Preserv_V^{\alpha,m} = VertG \setminus Del_V^{\alpha,m}$

***New vertices of** R:* $New_V^{\alpha,m} = RuleCreate_V^\alpha$

***New edges of** R:* $New_E^{\alpha,m} = \{e \in RuleCreate_E^\alpha |$
$(m_V(sourceR(e)) \in Preserv_V^{\alpha,m}$ *or* $sourceR(e) \in New_V^{\alpha,m})$ *and*
$(m_V(targetR(e)) \in Preserv_V^{\alpha,m}$ *or* $targetR(e) \in New_V^{\alpha,m})\}$

There are two situations that should be handled with care when applying a graph grammar rule to a graph G: (i) when a vertex is specified for deletion and there are edges connected to it in G that are not in the image of the rule's LHS; and (ii) in the case that two distinct items of the rule, one to be deleted and the other preserved, are mapped to the same element of G, or two different deleted items are mapped to the same in G. To ensure that no such situations

[5] When the rule is applied fresh instances of elements of New_V and New_E are created.
[6] $sourceG \rhd Del_V^{\alpha,m}$ is the set of all pairs (e, v) from $sourceG$ where $v \in Del_V^{\alpha,m}$, i.e. all pairs where the source vertex has been deleted.

occur when a rule is applied, the embedding of the LHS of the rule in the state graph must satisfy a condition called *gluing condition*. In SPO, the standard rule application does not necessarily require this condition whereas in DPO this condition is mandatory (because otherwise it would not be technically possible to obtain the result of the application as a double pushout). Applying a rule (in SPO) in the presence of any of these conditions leads to effects that are not explicitly specified by the rule: handling (i) requires to delete all edges whose source or target vertices were deleted, and handling (ii) requires to delete also the item that was marked for preservation.

Definition 13 (Gluing Condition). *Let $r = (\alpha, AN(\alpha))$ be a T-typed rule, with left-hand side L. Let G be a T-typed graph and m be a total typed graph morphism $m = (m_V, m_E)\colon L \to G$. We say that m* ***satisfies the gluing condition*** *if it satisfies the following two conditions:*

Identification condition: *There is no conflict between deletion and preservation of items and there is no identification of deleted items, that is*

$$Del_V^{\alpha,m} \cap m_V[RulePreserv_V^{\alpha}] = \varnothing, \qquad \texttt{(grd\_Ident1V)}$$
$$Del_E^{\alpha,m} \cap m_E[RulePreserv_E^{\alpha}] = \varnothing, \qquad \texttt{(grd\_Ident1E)}$$
$$card(Del_V^{\alpha,m}) = card(RuleDel_V^{\alpha}),\ \textit{and} \qquad \texttt{(grd\_Ident2V)}$$
$$card(Del_E^{\alpha,m}) = card(RuleDel_E^{\alpha}). \qquad \texttt{(grd\_Ident2E)}$$

Dangling condition: *There are no edges that will be deleted that are not specified by the rule, that is*

$$Dangling^{\alpha,m} = \varnothing \qquad \texttt{(grd\_DangC)}$$

Definition 14 (Match). *Let $r = (\alpha, AN(\alpha))$ be a T-typed rule, with left-hand side L. Let G be a T-typed graph. A* ***SPO-match*** *m* ***for rule*** *r* ***in*** *G is defined by a total typed graph morphism $m = (m_V, m_E)\colon L \to G$ that satisfies all NACs of r. A* ***DPO-match*** *requires, additionally, that the gluing condition is satisfied for m.*

The application of rules in SPO and DPO are defined based on the categorical pushout construction on categories of (typed) graphs. In the SPO approach the effect of a rule application is obtained by one pushout (in categories with partial morphisms) and in the DPO approach this effect is described by two pushouts (in categories with total morphisms), where the first pushout deletes items and the second introduces the new items in the graph. According to [16, 30], the graph resulting from a rule's application is obtained by removing all items that are marked for deletion (plus the extra ones that should be deleted if the gluing condition is false, in case of SPO) and adding the items created by the rule (typically a disjoint union is used to ensure that new items are distinct from old ones). The source, target and typing functions remain unchanged for the items that are not in the image of the rule's LHS and are inherited from the rule's RHS for the newly created items. In the following we give the definition of SPO rule application: we give the standard categorical definition and show an explicit construction of the result of the rule's application. In Theo. 16 we prove that this construction is indeed the pushout in the category of typed graphs and partial typed graph morphisms.

Definition 15 (Rule Application). *Let $r = (\alpha, AN(\alpha))$ be a T-typed rule with an SPO rule pattern $(L, AN(\alpha))$, and $m = (m_V, m_E)$ be a match of r in a T-typed graph G^T. A* ***rule application*** *$G \overset{r,m}{\Rightarrow} H$, or the application of r to G at m, is defined by the pushout (1) in the category $\mathbf{TGraphP}(\mathbf{T})$. If r is a DPO rule and m satisfies the gluing condition, the application of the rule is defined by pushouts (2) and (3) in category $\mathbf{TGraph}(\mathbf{T})$.*

$$\begin{array}{ccc} L & \xrightarrow{\alpha} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow \\ G & \longrightarrow & H \end{array} \qquad\qquad \begin{array}{ccccc} L & \xleftarrow{\alpha L} & K & \xrightarrow{\alpha R} & R \\ {\scriptstyle m}\downarrow & (2) & \downarrow & (3) & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

The components of the typed graph (H, tH, T), with $H = (vertH, edgeH, sourceH, targetH)$, are defined below (up to isomorphism), where $Dangling^{\alpha,m} = \varnothing$ and $New_E^{\alpha,m} = RuleCreate_E^{\alpha}$ in case of DPO (the gluing condition ensures that there will be no dangling edges and newly created edges are only generated if the corresponding source and target vertices are present in H):

$$VertH = (VertG \backslash Del_V^{\alpha,m}) \uplus New_V^{\alpha,m} \qquad \text{(act\_V)}$$

$$EdgeH = (EdgeG \backslash (Del_E^{\alpha,m} \cup Dangling^{\alpha,m})) \uplus New_E^{\alpha,m} \qquad \text{(act\_E)}$$

$$\begin{aligned} sourceH \quad = \quad & ((Del_E^{\alpha,m} \cup Dangling^{\alpha,m}) \lhd sourceG) \cup \\ & \{e \mapsto m_V(sourceR(e)) \mid e \in New_E^{\alpha,m} \wedge sourceR(e) \in RulePreserv_V^{\alpha}\} \cup \\ & \{e \mapsto sourceR(e) \mid e \in New_E^{\alpha,m} \wedge sourceR(e) \in New_V^{\alpha,m}\} \end{aligned} \qquad \text{(act\_src)}$$

$$\begin{aligned} targetH \quad = \quad & ((Del_E^{\alpha,m} \cup Dangling^{\alpha,m}) \lhd targetG) \cup \\ & \{e \mapsto m_V(targetR(e)) \mid e \in New_E^{\alpha,m} \wedge targetR(e) \in RulePreserv_V^{\alpha}\} \cup \\ & \{e \mapsto targetR(e) \mid e \in New_E^{\alpha,m} \wedge targetR(e) \in New_V^{\alpha,m}\} \end{aligned} \qquad \text{(act\_tgt)}$$

$$tH_V = (Del_V^{\alpha,m} \lhd tG_V) \cup \{v \mapsto tR_V(v) \mid v \in New_V^{\alpha,m}\} \qquad \text{(act\_tV)}$$

$$tH_E = ((Del_E^{\alpha,m} \cup Dangling^{\alpha,m}) \lhd tG_E) \cup \{e \mapsto tR_E(e) \mid e \in New_E^{\alpha,m}\} \qquad \text{(act\_tE)}$$

Remarks:

1. Note that the sets $New_V^{\alpha,m}$ and $New_E^{\alpha,m}$ are constructed in different ways (see Def. 12) because it might not be possible to create an edge if one of its vertices is deleted as a side effect of the chosen match (this can only happen if the gluing condition is not satisfied, which is allowed in SPO).
2. The result of a rule application is the same for SPO and DPO in case of matches satisfying the gluing condition [53, 30]. Therefore we will not show here explicitly how to construct the graph D (which is formally obtained by a pushout complement construction) and will prove only that the graph given in Def. 15 corresponds to the puhsout in $\mathbf{TGraphP}(\mathbf{T})$, the fact that it also corresponds to the result of a DPO rule application follows.

Theorem 16 (PO). *The construction given in Def. 15 is the pushout object in category* $\mathbf{TGraphP}(\mathbf{T})$.

PROOF. In the following we will explain intuitively how pushouts in the involved categories are constructed. The formal definitions are given in Appendix A.

Pushouts in $\mathbf{TGraph}(\mathbf{T})$ can be constructed componentwise (in $\mathbf{Graph}$, that, in turn, can be constructed componentwise in $\mathbf{Set}$) [16, 28]. If we consider diagram (1) in $\mathbf{TGraph}(\mathbf{T})$, where α is injective, the result H is obtained (up to isomorphism) by creating one copy of each vertex/edge that is in $rng(m)$ and adding all elements from R and G that are not in the images of α (created elements) and m (preserved elements), respectively. Once the sets of vertices and edges are constructed, source, target and typing functions follow (there is only once choice that would make the diagram commute). Pushouts in $\mathbf{TGraphP}(\mathbf{T})$ are more involved because, although morphisms may be partial, graphs must remain "total structures", that is, source, target and typing functions must be total. In [51] it was shown that it is possible to construct these pushouts in a quasi-componentwise way[7]: the construction is done componentwise (in $\mathbf{GraphP}$ and $\mathbf{SetP}$) followed by a free construction that "totalizes" the result such that it becomes a valid graph (actually, the biggest graph that makes the diagram commute) (for more details see Def. 37 in Appendix A).
Diagram (1) depicts the application of an SPO-rule to a (total) match, thus H will be obtained as follows:

[7] In [51] it was shown that this kind of construction of pushouts is possible in several categories, called generalized graph structures.

V_H will be the pushout object in $\mathbf{SetP}$ of α_V and m_V. Since α_V is injective and m_V is total, the construction is similar to the total case discussed above, we just have to, additionally, remove from G all vertices that are images of some element of L that is marked for deletion (not in $dom(\alpha_V)$). That is, V_H is obtained by removing from G all vertices marked for deletion and adding all vertices created by α (see Prop. 38 in Appendix A).

E_H can be constructed as a pushout of edges analogously to the pushout of vertices, but we then have to perform an additional step that deletes all edges whose source or target vertex is not in V_H (if any of such edges remain, the result would not be a graph). These edges are exactly the ones we put in the set called $Dangling^{\alpha,m}$ (see Prop. 39 in Appendix A).

Source*, *target*, *typing functions are then obvious: they are the only choices that make the diagram commute. For every vertex/edge in H, its type will be the one in G if it was preserved, and the one in R if it was created by the rule application. For every edge that was preserved, we take the same source/target as in G. For the edges that were created by α, we have to check whether the source/target is a new vertex or a vertex from G, and map it accordingly.

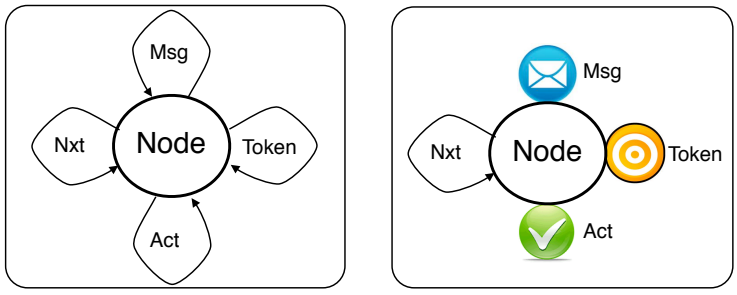
■

When the gluing condition is satisfied it is possible to make the definition of the result of a rule application more concrete, since we know exactly beforehand which elements will be deleted and created (there can be no extra deletion due to dangling or identification conditions being false). This will be exploited further in Sect. 5 (Def. 21).

4.1. Example: Token Ring Protocol

We illustrate the use of graph grammars with NACs 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 [75]. 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 is sent from station to station. 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 stations may be added and deleted at any time.

The type graph for this example is depicted in Fig. 2(a). This graph defines a single type of vertex Node, and four types of edges Msg (Message), Token (Token), Nxt (Next) and Act (Active Station). Node represents a network station and Msg defines a frame of data. The stations are connected by edges of type Nxt. The Tok represents the token bit pattern. A station is active (Act), if it is transmitting a message on the network. There can be only one active station on a ring at each time. In order to make the graph representation more intuitive, we use a different graphical notation for some edges, as can be seen in Fig. 2(b).



(a) Type graph (b) Graphical notation

Figure 2: Types for token-ring protocol.

Figure 3 presents the graph grammar TR_{GG} for the example. The type graph T is defined as described above (see Fig. 2). The initial graph G0 defines a ring with three nodes (named 1, 2 and 3), and four edges (named 1, 3,

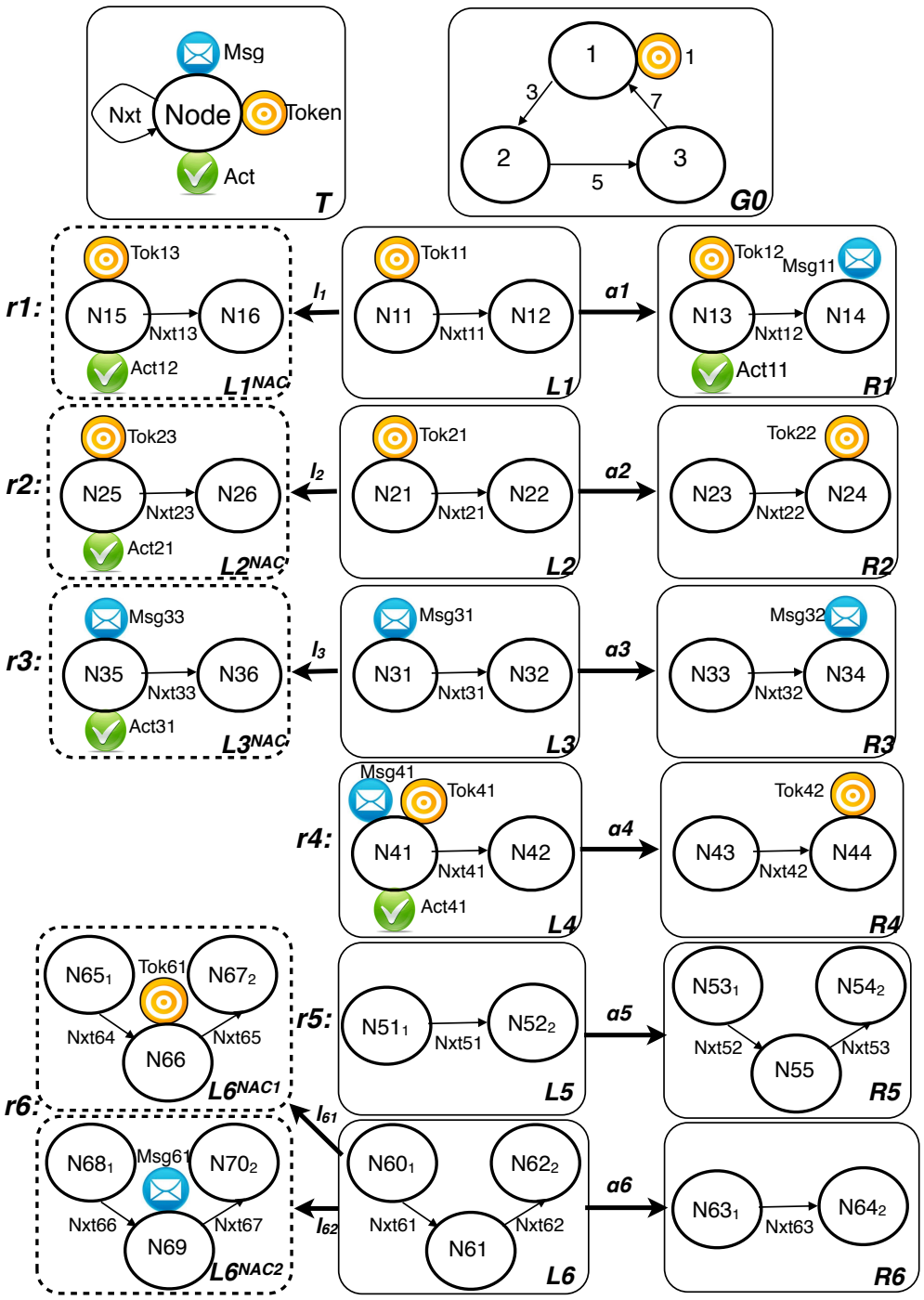


Figure 3: Token Ring Graph Grammar with NACs TR_{GG}.

5 and 7). The type of the nodes is Node and the types of edges are implicitly given by the graphical representation (edge 1 is of type Tok and the others are of type Nxt). Initially the Token (edge 1) is at station 1 and no station is transmitting information on the network. The behavior of the protocol is modeled by rules. The component αi of a rule ri describes how items are mapped from the left- to the right-hand side of the rule, whereas each component l_{ij} models the j-th NAC of rule r_i (when r_i has just one NAC, j index is omitted). In Fig. 3, these mappings are omitted, when they are obvious; otherwise, they are defined by subscribed numbers. Moreover, the position of rule elements favors the mapping visualization. For example, for the rule r_1, α_1 is defined by $\alpha_{1_V}(\mathsf{N11}) = \mathsf{N13}$, $\alpha_{1_V}(\mathsf{N12}) = \mathsf{N14}$ and $\alpha_{1_E}(\mathsf{Tok11}) = \mathsf{Tok12}$, $\alpha_{1_E}(\mathsf{Nxt11}) = \mathsf{Nxt12}$; and for the rule r_5, α_5 is defined by $\alpha_{5_V}(\mathsf{N51}) = \mathsf{N53}$, $\alpha_{5_V}(\mathsf{N52}) = \mathsf{N54}$ and $\alpha_{5_E} = \varnothing$. NACs of rules $r1$, $r2$ and $r3$ restrict the application of these rules to non-active stations. Rules $r1$ and $r2$ specify the behavior of the protocol when a non-active station receives a token: it may hold the token and send a message to the next node, becoming an active station (rule $r1$) or simply pass the token to the next station (rule $r2$). By rule $r3$ if a non-active station receives a message, it may pass the message to the next node. If the receiving node is an active station, then rule $r4$ can be applied, removing the message from the ring, turning the station inactive and sending the token to the next station. Rule $r5$ is applied to insert a new station into the ring, and rule $r6$ deletes a station from the ring. NACs of rule $r6$ restrict the deletion of stations to those that do not have a token (and consequently can not be active, this will be proven later) and do not hold a message[8]. This model has an infinite state-space and generates an infinite number of possible computations.

Notice that we require two separate NACs, $L_6 \rightarrow L_6^{NAC_1}$ and $L_6 \rightarrow L_6^{NAC_2}$, in rule $r6$. This means that the node to be removed may not have a Tok edge and may not have a Msg edge (if any of these situations occur, the rule can not be applied). This rule has a different effect comparing to rule $r6'$ depicted in Fig. 4. In that case, the node to be removed may not have a Tok and a Msg edges, i.e. *the two edges* must not exist at the same time. This latter rule would be applicable in a situation in which a node has only one Msg edge, allowing it to be removed (and consequently removing the Msg edge from the ring, if we consider SPO semantics).

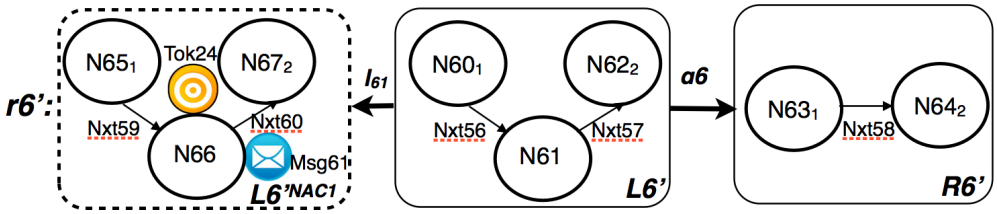


Figure 4: Rule $r6'$.

There are many properties that may be required for the token-ring protocol presented in this section. Here we discuss how two of them can be guaranteed for our graph grammar specification:

At each time, there is exactly one token in the ring (`uniqueTok`): For all graphs reachable from the initial graph $G0$ applying rules of TR_{GG} there is exactly one edge of type Tok. This property is satisfied, since the initial graph satisfies the property and each rule that creates a Tok edge, also deletes another edge of this same type. Moreover, the rule that deletes a vertex can not be applied to vertices that have a Tok edge;

At each time, at most one station is transmitting through the network, and it is holding the token bit pattern (`uniqueAct`): For all reachable graphs of TR_{GG} there is at most one Act edge and, if such edge exists, there is a Tok edge in the same vertex. Intuitively, we can see that this property holds for our grammar because: it is true for the initial graph; the rule that creates edges of type Act has a NAC forbidding its creation if there already exists an edge of this type in the same vertex, as well as requiring the existence of a Tok edge in the same vertex; and, finally, by using the `uniqueTok` and the fact that a Tok edge can be deleted only if there is no Act edge in the same vertex (otherwise the Act edge is also deleted).

[8]If DPO approach is adopted the NACs of rule $r6$ are not necessary because the dangling condition hinders rule application when the forbidden itens are present

5. Translation of Graph Grammars to Event-B Models

Now we define the translation of a graph grammar with NACs to an event-B model. This translation can be applied only to graph grammars defined by finite structures. Roughly speaking, static components of a graph grammar (type graph and rules) will be translated to event-B context elements, that is, sets, functions and axioms that define data types and elements that do not change during the execution of the model. The event-B machine defines a state (set of variables), invariants, and events. The state of the event-B model will contain variables that describe a typed graph (sets of vertices and edges, and the source, target and typing functions). Invariants are used to define the types of these variables. There will be an event (initialization event) to generate the initial state corresponding to the initial graph of the grammar and a set of events corresponding to the rules of the grammar (one for each rule). Recall that empty types are not allowed in an event-B context, and therefore in case that one of the sets used in initial graph or rules of the graph grammar is empty, we must omit its definition in the translation. For example, if a LHS L has no edges, we have to omit the definitions of the set $EdgeL$ and functions $sourceL$, $targetL$ and tL_E in the translated event-B model (these elements, when they exist, are defined in the context because they never change during the application of the rules). But an equation like $Dangling = \varnothing$ is allowed as a guard of an event (because this is not a definition, but rather a test). In the definitions of the translation throughout this paper, we will not state explicitly that empty sets should be omitted, but this is a general rule that applies to all such definitions.

The translation of a graph grammar will be done according to the following steps:

T1 Translate the type graph;

T2 Translate the state graph and initial graph; and

T3 Translate the rules.

The translation will be defined from a graph grammar to an event-B model using the Camille syntax, instead of using the mathematical definition of event-B. Although one could argue that this is a rather syntactical level, this choice has two main reasons: the definitions of the translation become easier to read and understand; and the reader gets a precise idea of how contexts and machines look like in the Rodin environment, where the proofs have to be carried out, sometimes with user support (therefore, knowing the exact names and structures used in the translation is important). To improve the text flow, all figures describing the translation of the running example are gathered together in Appendix B.

Notation: *To define the translation of a graph grammar to an event-B model, we use black italic fonts for items that must be replaced by concrete values (depending on the GG being translated) and non-italic blue fonts for items that must be used literally (i.e., that are independent of the GG being translated). When quantification over concrete elements of a GG is needed (for example, over all elements of a set of vertex types $VertT$), we use a vertical bar to indicate the scope of the quantification. Comments are introduced by $//$. Moreover, we will often use sub and superscripts in the context and machine specifications: these are not allowed in the tool, but we will use them here for a cleaner presentation.*

We will use the following auxiliary functions, which map sets into strings:

- $elements$: takes a (non-empty finite) set and returns a string representing all elements, each inside curly brackets. That is, $elements(S) = \{s1\}$,..., $\{sn\}$, for all $s_i \in S$.
- set: takes a (non-empty finite) set and returns a string representing all elements inside curly brackets. Namely, $set(S) = \{s1,...,sn\}$, for all $si \in S$.
- $prefixed\_set$: takes a (non-empty finite) set S and a string s and returns a string representing all elements of S prefixed with s inside curly brackets. Namely, $prefixed\_set(S, \mathtt{s}) = \{\mathtt{s\_}s1,...,\mathtt{s\_}sn\}$, for all $si \in S$.
- $list$: takes a set and returns a string representing all elements separated by blank spaces. Namely, $list(S) = s1\ \ s2\ \ ...\ \ sn$, for all $si \in S$.
- $prefixed\_list$: takes a set S and a string s and returns a string representing all elements of S prefixed with s separated by black spaces. Namely, $prefixed\_list(S, s) = \mathtt{s\_}s1\ \ \mathtt{s\_}s2\ \ ...\ \ \mathtt{s\_}sn$, for all $si \in S$.

Note that, since functions are sets of pairs, the constructions above also apply to functions, where the syntax used to denote each pair $(a, f(a))$ is $a \mapsto f(a)$.

Step T1. Translation of the type graph. A type graph T is translated in an event-B context, where all vertex types, all edge types and the names of source and target functions are specified as constants. The names of the sets of vertices and edges are specified as sets and in the axioms, these sets and functions are explicitly defined.

Definition 17 (Translation of a Type Graph). *Given a graph* $T = (VertT, EdgeT, sourceT, targetT)$ *we define its translation* $\mathcal{T}^T$ *as the event-B context elements:*

sets

VertT

EdgeT

constants

$list(VertT)$

$list(EdgeT)$

sourceT

targetT

axioms

axm_VertT : partition(VertT, $elements(VertT)$)

axm_EdgeT : partition(EdgeT, $elements(EdgeT)$)

axm_srcTtype : sourceT $\in$ EdgeT $\rightarrow$ VertT

axm_srcTdef : partition(sourceT, $elements(sourceT)$)

axm_tgtTtype : targetT $\in$ EdgeT $\rightarrow$ VertT

axm_tgtTdef : partition(targetT, $elements(targetT)$)

The $\text{partition}(S, s_1, \ldots, s_n)$ predicate states that the sets $s_1, \ldots, s_n$ constitute a partition of S. The union of all elements of a partition is S and all elements are disjoint. In this definition, the **partition** predicate is used to define a set with the listed elements (this is used to avoid having to add axioms stating, for each pair of vertex or edge types, that they are different). Figure B.11 shows the translation of the type graph of the token ring example (see Fig. 3) following Def. 17. VertT and EdgeT are sets known in the model. All vertex and edge types, sourceT and targetT are specified as constants. In the axioms, all sets and functions are explicitly described. Note that, from axm_EdgeT it follows, for example, that Tok $\neq$ Act due to the **partition** construction.

Step T2: Translation of state graph and initial graph. A state graph is modeled by a set of variables describing its set of vertices, set of edges, source, target and typing functions. The initialization event is a special event in an event-B model that is executed before any other. In our encoding, this event creates the initial graph of the graph grammar. This is done by assigning initial values to the variables that correspond to graph $G0$. We use natural numbers to denote edges and vertices of the state graph.

Definition 18 (Translation of State Graph and Initial Graph). *A state graph is defined by the following variables and invariants in event-B:*

variables

VertG

EdgeG

sourceG

targetG

tG_V

tG_E

invariants

inv_VertG : VertG $\in \mathbb{P}(\mathbb{N})$

inv_EdgeG : EdgeG $\in \mathbb{P}(\mathbb{N})$

$\texttt{inv\_sourceG} : \texttt{sourceG} \in \texttt{EdgeG} \rightarrow \texttt{VertG}$

$\texttt{inv\_targetG} : \texttt{targetG} \in \texttt{EdgeG} \rightarrow \texttt{VertG}$

$\texttt{inv\_tG}_\texttt{V} : \texttt{tG}_\texttt{V} \in \texttt{VertG} \rightarrow \texttt{VertT}$

$\texttt{inv\_tG}_\texttt{E} : \texttt{tG}_\texttt{E} \in \texttt{EdgeG} \rightarrow \texttt{EdgeT}$

Given an initial graph $G0^T = (G0, tG0, T)$ *with* $VertG0 \subseteq \mathbb{N}$ *and* $EdgeG0 \subseteq \mathbb{N}$*, its translation* $\mathcal{T}^{G0}$ *to event-B is defined by the following initialisation event:*

events

event INITIALISATION

then

$\texttt{act\_VertG} : \texttt{VertG} := set(VertG0)$

$\texttt{act\_EdgeG} : \texttt{EdgeG} := set(EdgeG0)$

$\texttt{act\_srcG} : \texttt{sourceG} := set(sourceG0)$

$\texttt{act\_tgtG} : \texttt{targetG} := set(targetG0)$

$\texttt{act\_tG}_\texttt{V} : \texttt{tG}_\texttt{V} := set(tG0_V)$

$\texttt{act\_tG}_\texttt{E} : \texttt{tG}_\texttt{E} := set(tG0_E)$

end

The specification of the variables, invariants and initialization event of the token ring example (Fig. 3) are detailed in Fig. B.12.

Step T3: Translation of rules. In order to translate the LHS of a rule L^T, we must extend the previous translation of T with the specification of graph L and of the corresponding typing morphisms. Similarly, vertices and edges of L, and the names of functions must be specified as constants. The names of the sets of vertices and edges of L must be described as sets. The axioms must define all sets and functions explicitly.

Definition 19 (Translation of the LHS of a Rule Pattern). *Given a rule pattern* $P = (L^T, AN(\alpha))$*, the translation of* L *and* t_L *to event-B, denoted by* $\mathcal{T}^P$*, is defined by the following elements:*

sets

$\texttt{VertL}$

$\texttt{EdgeL}$

constants

$list(VertL)$

$list(EdgeL)$

$\texttt{sourceL}$

$\texttt{targetL}$

$\texttt{tL}_\texttt{V}$

$\texttt{tL}_\texttt{E}$

axioms

analogous to axioms of $\mathcal{T}^T$, for graph L, including axioms that define tL_V and tL_E :

$\texttt{axm\_tL}_\texttt{V} : \texttt{tL}_\texttt{V} \in \texttt{VertL} \rightarrow \texttt{VertT}$

$\texttt{axm\_tL}_\texttt{V}\texttt{\_def} : \texttt{partition}(\texttt{tL}_\texttt{V}, elements(tL_V))$

$\texttt{axm\_tL}_\texttt{E} : \texttt{tL}_\texttt{E} \in \texttt{EdgeL} \rightarrow \texttt{EdgeT}$

$\texttt{axm\_tL}_\texttt{E}\texttt{\_def} : \texttt{partition}(\texttt{tL}_\texttt{E}, elements(tL_E))$

Figure B.13 shows the event-B description of the LHS of $r1$ rule pattern (using Def. 19). The specifications of graph $L1$ is analogous to T. Typing morphisms are added as constants, which are concretely defined in the axioms.

Due to the definition of set theoretic NAC satisfaction (Def. 10), we do not need to translate the NAC-morphisms to the event-B context. This is because the corresponding guard conditions are described in terms of L and G only.

A rule application is implemented by an event occurrence. Whenever there is a match that satisfies the NACs of the rule (the rule can be applied), the corresponding event is enabled, and its occurrence updates the state-graph in the same way the rule does. According to the next definition an event may happen when there are concrete values for variables ($\mathtt{m_V}$, $\mathtt{m_E}$, ..., $\mathtt{New_E}$, $\mathtt{new_E}$) that satisfy all guard conditions of the event. Guard conditions `grd_mV`, `grd_mE`, `grd_tV`, `grd_tE` and `grd_srctgt` ensure that the pair $(\mathtt{m_V}, \mathtt{m_E})$ is actually a match from the left-hand side of the rule to the state-graph G. Guard conditions prefixed with `grd_NAC` ensure that the match satisfies the NACs. Guard conditions `grd_NewV` and `grd_NewE` ensure that the created items will be fresh elements, while guard conditions `grd_newV` and `grd_newE` establish bijections between the set of identifiers of the new vertices (respect. edges) and the set of created vertices (respect. edges). These bijections are necessary to define source/target and typing of created items, guaranteeing that they are fresh items (consequently implementing the disjoint union for created vertices and edges). The remaining guards describe the sets of deleted, preserved and created items, following Def. 12. Actions update the state-graph according to the rule specification (assignments implement exactly the formulas of Def. 15[9]).

Definition 20 (Translation of a Rule Application). *Given an SPO-rule $r = (\alpha : L^T \to R^T, AN(\alpha))$, the translation $\mathcal{T}^r$ of the rule r to event-B is given by the following event:*

event r

any

$\mathtt{m_V}$ // *match of vertices*
$\mathtt{m_E}$ // *match of edges*
$\mathtt{Del_V}$ // *deleted vertices*
$\mathtt{Preserv_V}$ // *preserved vertices*
$\mathtt{New_V}$ // *new vertices*
$\mathtt{new_V}$ // *function relating new vertices to the ones in RHS of r*
$\mathtt{Del_E}$ // *deleted edges*
`Dangling` // *dangling edges*
$\mathtt{New_E}$ // *new edges*
$\mathtt{new_E}$ // *function relating new edges to the ones in RHS of r*

where

`grd_mV` : $\mathtt{m_V} \in \mathtt{Vert}L \to \mathtt{VertG}$ // *total function mapping vertices*
`grd_mE` : $\mathtt{m_E} \in \mathtt{Edge}L \to \mathtt{EdgeG}$ // *total function mapping edges*
`grd_DelV` : $\mathtt{Del_V} := \mathtt{m_V}[set(RuleDel_V)]$ // *set of deleted vertices of G*
`grd_PreV` : $\mathtt{Preserv_V} = \mathtt{VertG} \setminus \mathtt{Del_V}$ // *set of preserved vertices of G*
`grd_NewV` : $\mathtt{New_V} \subseteq \mathbb{N} \setminus \mathtt{VertG}$ // *set of created vertices*
`grd_newV` : $\mathtt{new_V} \in \mathtt{New_V} \rightarrowtail\!\!\!\!\!\!\to set(RuleCreate_V)$ // New_V *is isomorphic to* $RuleCreate_V$
`grd_DelE` : $\mathtt{Del_E} := \mathtt{m_E}[set(RuleDel_E)]$ // *set of deleted edges of G*
`grd_Dang` : $\mathtt{Dangling} = \mathtt{dom}((\mathtt{sourceG} \rhd \mathtt{Del_V}) \cup (\mathtt{targetG} \rhd \mathtt{Del_V})) \setminus \mathtt{Del_E}$ // *dangling edges of G*
`grd_NewE` : $\mathtt{New_E} \subseteq \mathbb{N} \setminus \mathtt{EdgeG}$ // *set of created edges*
`grd_newE` : $\mathtt{new_E} \in \mathtt{New_E} \rightarrowtail\!\!\!\!\!\!\to \{\mathtt{e} \in set(RuleCreate_E) \mid (\mathtt{m_V}(\mathtt{source}R(\mathtt{e})) \in \mathtt{Preserv_V} \vee \mathtt{source}R(\mathtt{e}) \in \mathtt{new_V}[\mathtt{New_V}]) \wedge (\mathtt{m_V}(\mathtt{target}R(\mathtt{e})) \in \mathtt{Preserv_V} \vee \mathtt{target}R(\mathtt{e}) \in \mathtt{new_V}[\mathtt{New_V}])\}$
`grd_tV` : $\forall \mathtt{v} \cdot \mathtt{v} \in \mathtt{Vert}L \Rightarrow \mathtt{t}L_V(\mathtt{v}) = \mathtt{tG_V}(\mathtt{m_V}(\mathtt{v}))$ // *vertex type compatibility*
`grd_tE` : $\forall \mathtt{e} \cdot \mathtt{e} \in \mathtt{Edge}L \Rightarrow \mathtt{tL_E}(\mathtt{e}) = \mathtt{tG_E}(\mathtt{m_E}(\mathtt{e}))$ // *edge type compatibility*
`grd_srctgt` : $\forall \mathtt{e} \cdot \mathtt{e} \in \mathtt{Edge}L \Rightarrow \mathtt{m_V}(\mathtt{source}L(\mathtt{e})) = \mathtt{sourceG}(\mathtt{m_E}(\mathtt{e})) \wedge \mathtt{m_V}(\mathtt{target}L(\mathtt{e})) = \mathtt{targetG}(\mathtt{m_E}(\mathtt{e}))$ // *source/target compatibility*

[9]Since `grd_NewV` in Def. 20 guarantees that $\mathtt{New_V}$ contains names of vertices that are not in the state graph, union is used in `act_V` instead of disjoint union of Def. 15 (and analogous treatment is given to edges).

$\forall NACj \in AN(\alpha)\cdot$ *// NAC satisfaction*

$\quad$ grd_NACj:$\neg(\exists \overline{v}_1,\ldots,\overline{v}_n,\overline{e}_1,\ldots,\overline{e}_m.$ *where* $v_i \in NACj_V$ *and* $e_i \in NACj_E$

$\quad\quad \{\overline{v}_1,\ldots\overline{v}_n\} \subseteq \texttt{VertG} \setminus \texttt{m}_\texttt{V}[\texttt{Vert}L] \wedge$ *// (nac1′)*

$\quad\quad \{\overline{e}_1,\ldots,\overline{e}_m\} \subseteq \texttt{EdgeG} \setminus \texttt{m}_\texttt{E}[\texttt{Edge}L] \wedge$ *// (nac2′)*

$\quad\quad \forall v1, v2 \in NACj_V$ *with* $v1 \neq v2$

$\quad\quad\quad \overline{v1} \neq \overline{v2}$ *// (nac3′)*

$\quad\quad \forall e1, e2 \in NACj_E$ *with* $e1 \neq e2$

$\quad\quad\quad \overline{e1} \neq \overline{e2}$ *// (nac4′)*

$\quad\quad \forall v \in NACj_V$ *with* $tL_V^{NACj}(v) = tv.$

$\quad\quad\quad \texttt{tG}_\texttt{V}(\overline{v}) = tv \ \wedge$ *// (nac5′)*

$\quad\quad \forall e \in NACj_E$ *with* $tL_E^{NACj}(e) = te.$

$\quad\quad\quad \texttt{tG}_\texttt{E}(\overline{e}) = te \ \wedge$ *// (nac6′)*

$\quad\quad\quad \begin{cases} \texttt{sourceG}(\overline{e}) = \overline{v} & \text{, if } sourceL^{\texttt{NAC}_j}(e) = v \text{ and } v \in NACj_V \\ \texttt{sourceG}(\overline{e}) = \texttt{m}_\texttt{V}(v) & \text{, otherwise, with } v = lj_V^{-1}(sourceL^{\texttt{NAC}_j}(e)) \end{cases}$ *// (nac7′)*

$\quad\quad\quad \wedge$

$\quad\quad\quad \begin{cases} \texttt{targetG}(\overline{e}) = \overline{v} & \text{, if } targetL^{\texttt{NAC}_j}(e) = v \text{ and } v \in NACj_V \\ \texttt{targetG}(\overline{e}) = \texttt{m}_\texttt{V}(v) & \text{, otherwise, with } v = lj_V^{-1}(targetL^{\texttt{NAC}_j}(e)) \end{cases}$ *// (nac8′)*

$\quad\quad) \ \vee$

$\quad\quad (\bigvee_{(v1,v2)\in NACj_{idV}} \texttt{m}_\texttt{V}(v1) \neq \texttt{m}_\texttt{V}(v2)) \ \vee$ *// (nac9′)*

$\quad\quad (\bigvee_{(e1,e2)\in NACj_{idE}} \texttt{m}_\texttt{E}(e1) \neq \texttt{m}_\texttt{E}(e2))$ *// (nac10′)*

then

$\quad \texttt{act\_V} : \texttt{VertG} := (\texttt{VertG} \setminus \texttt{Del}_\texttt{V}) \cup \texttt{New}_\texttt{V}$

$\quad \texttt{act\_E} : \texttt{EdgeG} := (\texttt{EdgeG} \setminus (\texttt{Del}_\texttt{E} \cup \texttt{Dangling})) \cup \texttt{New}_\texttt{E}$

$\quad \texttt{act\_src} : \texttt{sourceG} := ((\texttt{Del}_\texttt{E} \cup \texttt{Dangling}) \lhd \texttt{sourceG}) \cup$

$\quad\quad \{\texttt{e} \mapsto \texttt{m\_V}(\texttt{source}R(\texttt{new}_\texttt{E}(\texttt{e}))) \mid \texttt{e} \in \texttt{New}_\texttt{E} \wedge \texttt{source}R(\texttt{new}_\texttt{E}(\texttt{e})) \in \texttt{Preserv}_\texttt{V}\} \cup$

$\quad\quad \{\texttt{e} \mapsto \texttt{source}R(\texttt{new}_\texttt{E}(\texttt{e})) \mid \texttt{e} \in \texttt{New}_\texttt{E} \wedge \texttt{source}R(\texttt{new}_\texttt{E}(\texttt{e})) \in \texttt{new}_\texttt{V}[\texttt{New}_\texttt{V}]\}$

$\quad \texttt{act\_tgt} : \texttt{targetG} := ((\texttt{Del}_\texttt{E} \cup \texttt{Dangling}) \lhd \texttt{targetG}) \cup$

$\quad\quad \{\texttt{e} \mapsto \texttt{m\_V}(\texttt{target}R(\texttt{new}_\texttt{E}(\texttt{e}))) \mid \texttt{e} \in \texttt{New}_\texttt{E} \wedge \texttt{target}R(\texttt{new}_\texttt{E}(\texttt{e})) \in \texttt{Preserv}_\texttt{V}\} \cup$

$\quad\quad \{\texttt{e} \mapsto \texttt{target}R(\texttt{new\_E}(\texttt{e})) \mid \texttt{e} \in \texttt{New}_\texttt{E} \wedge \texttt{target}R(\texttt{new}_\texttt{E}(\texttt{e})) \in \texttt{new}_\texttt{V}[\texttt{New}_\texttt{V}]\}$

$\quad \texttt{act\_tV} : \texttt{tG}_\texttt{V} := (\texttt{Del}_\texttt{V} \lhd \texttt{tG}_\texttt{V}) \cup \{\texttt{v} \mapsto \texttt{t}R_\texttt{V}(\texttt{new}_\texttt{V}[\texttt{New}_\texttt{V}]) \mid \texttt{v} \in \texttt{New}_\texttt{V}\}$

$\quad \texttt{act\_tE} : \texttt{tG}_\texttt{E} := ((\texttt{Del}_\texttt{E} \cup \texttt{Dangling}) \lhd \texttt{tG}_\texttt{E}) \cup \{\texttt{e} \mapsto \texttt{t}R_\texttt{E}(\texttt{new}_\texttt{E}[\texttt{New}_\texttt{E}]) \mid \texttt{e} \in \texttt{New}_\texttt{E}\}$

end

In the previous definition, the before-after predicates (actions) basically translate formulas of Def. 15 to event-B notation, implementing (according to Theo. 16) rule application in the SPO-approach. This translation requires that, additionally to the LHS, the RHS, rule morphisms and NACs of rules are explicitly part of the event-B context. The translation of the RHS would be analogous to Def. 19, and the rule morphism could be defined in the obvious way (as a pair of concrete functions mapping the LHS to the RHS of the rule). Although it would be possible to generate an event-B model using such translation of rule application, proving properties over the generated events would not be easy due to the fact that the guards of the event are stated at a very abstract level, as will be explained in the sequel.

Now we may ask whether it would be possible to make a more concrete definition of rule application as well. The short answer is: only if we assume that all matchs satisfy the gluing condition. In the general case of SPO rewriting it is not possible to know just by looking at the rule which items will be deleted/created because this may depend on the concrete match that is chosen (remember that if the match does not satisfy the gluing condition, side effects may happen). On the other hand, considering restricted cases of rule applications, for instance, SPO with gluing condition (or DPO), or alternatively if we restrict the SPO-approach to rules that do not delete vertices and do not have two elements of the same type, these sets can be statically defined, independently of the match. That is, for any match, the elements to be created or deleted can be calculated a priori. In such cases, the specification of the variables that define the new state-graph can be made more concrete and the relations between the elements of the resulting state-graph are evident. Consequently, the process of proving properties over reachable graphs becomes much simpler. In the following, we specify the translation of a rule application considering the restrictions discussed above, and call this DPO-rule application. This translation can be used in cases where created and deleted items can be determined just

by examining the rules. These cases may be either DPO or SPO with the restrictions discussed above. This definition can be considered as a simplification of Def. 20, since the sets $\texttt{New}_\texttt{V}$, $\texttt{New}_\texttt{E}$, $\texttt{Del}_\texttt{V}$, $\texttt{Del}_\texttt{E}$ and $\texttt{Preserv}_\texttt{V}$ can be statically calculated. Guard conditions are included to guarantee that the gluing condition is satisfied.

Definition 21 (Translation of a DPO-rule Application). *Given a DPO-rule $r = (\alpha, AN(\alpha))$, the translation $\mathcal{T}^r_{DPO}$ of the rule r to event-B event is given by the following event:*

event r
  any
    $\texttt{m}_\texttt{V}$ // *match vertices*
    $\texttt{m}_\texttt{E}$ // *match edges*
    $\texttt{Del}_\texttt{V}$ // *deleted vertices*
    $\texttt{Preserv}_\texttt{V}$ // *preserved vertices*
    $\texttt{Del}_\texttt{E}$ // *deleted edges*
    $\texttt{Dangling}$ // *dangling edges*
    $prefixed\_list(New_V, \texttt{new})$ // *vertices created by the rule*
    $prefixed\_list(New_E, \texttt{new})$ // *edges created by the rule*
  where
    $\texttt{grd\_mV} : \texttt{m}_\texttt{V} \in \texttt{Vert}L \rightarrow \texttt{VertG}$ // *total function mapping vertices*
    $\texttt{grd\_mE} : \texttt{m}_\texttt{E} \in \texttt{Edge}L \rightarrow \texttt{EdgeG}$ // *total function mapping edges*
    $\texttt{grd\_DelV} : \texttt{Del}_\texttt{V} := \texttt{m}_\texttt{V}[set(RuleDel_V)]$ // *set of deleted vertices of G*
    $\texttt{grd\_PreV} : \texttt{Preserv}_\texttt{V} = \texttt{VertG} \setminus \texttt{Del}_\texttt{V}$ // *set of preserved vertices of G*
    $\texttt{grd\_DelE} : \texttt{Del}_\texttt{E} := \texttt{m}_\texttt{E}[set(RuleDel_E)]$ // *set of deleted edges of G*
    $\texttt{grd\_Dang} : \texttt{Dangling} = \texttt{dom}((\texttt{sourceG} \rhd \texttt{Del}_\texttt{V}) \cup (\texttt{targetG} \rhd \texttt{Del}_\texttt{V})) \setminus \texttt{Del}_\texttt{E}$ // *dangling edges of G*
  $\forall v \in New_V$ // *fresh ids of vertices*
    | $\texttt{grd\_new\_v} : \texttt{new\_v} \in \mathbb{N} \setminus \texttt{VertG}$
  $\forall e \in New_E$ // *fresh ids of edges*
    | $\texttt{grd\_new\_e} : \texttt{new\_e} \in \mathbb{N} \setminus \texttt{EdgeG}$
  $\forall v_i, v_j \in New_V$ *with* $v_i \neq v_j$ // *new vertices can not have the same id*
    | $\texttt{grd\_diff}v_iv_j : \texttt{new\_}v_i \neq \texttt{new\_}v_j$
  $\forall e_i, e_j \in New_E$ *with* $e_i \neq e_j$ // *new edges can not have the same id*
    | $\texttt{grd\_diff}e_ie_j : \texttt{new\_}e_i \neq \texttt{new\_}e_j$
    $\texttt{grd\_tV} : \forall \texttt{v} \cdot \texttt{v} \in \texttt{Vert}L \Rightarrow \texttt{t}L_\texttt{V}(\texttt{v}) = \texttt{tG}_\texttt{V}(\texttt{m}_\texttt{V}(\texttt{v}))$ // *vertex type compatibility*
    $\texttt{grd\_tE} : \forall \texttt{e} \cdot \texttt{e} \in \texttt{Edge}L \Rightarrow \texttt{t}L_\texttt{E}(\texttt{e}) = \texttt{tG}_\texttt{E}(\texttt{m}_\texttt{E}(\texttt{e}))$ // *edge type compatibility*
    $\texttt{grd\_srctgt} : \forall \texttt{e} \cdot \texttt{e} \in \texttt{Edge}L \Rightarrow \texttt{m}_\texttt{V}(\texttt{source}L(\texttt{e})) = \texttt{sourceG}(\texttt{m}_\texttt{E}(\texttt{e})) \wedge \texttt{m}_\texttt{V}(\texttt{target}L(\texttt{e})) = \texttt{targetG}(\texttt{m}_\texttt{E}(\texttt{e}))$ // *source/target compatibility*
  $\forall NAC_j \in AN(\alpha) \cdot$ // *NAC satisfaction*
    | $\texttt{grd\_NAC}j$: *See Def. 10*
    $\texttt{grd\_Ident1V} : \texttt{Del}_\texttt{V} \cap \texttt{m}_\texttt{V}[RulePreserv_V] = \varnothing$ // *Identification condition 1 on vertices*
    $\texttt{grd\_Ident2V} : \texttt{card}(\texttt{Del}_\texttt{V}) = \texttt{card}(set(RuleDel_V))$ // *Identification condition 2 on vertices*
    $\texttt{grd\_Ident1E} : \texttt{Del}_\texttt{E} \cap \texttt{m}_\texttt{E}[RulePreserv_E] = \varnothing$ // *Identification condition 1 on edges*
    $\texttt{grd\_Ident2E} : \texttt{card}(\texttt{Del}_\texttt{E}) = \texttt{card}(set(RuleDel_E))$ // *Identification condition 2 on edges*
    $\texttt{grd\_DangC} : \texttt{Dangling} = \varnothing$ // *Dangling condition*
  then
    $\texttt{act\_V} : \texttt{VertG} := (\texttt{VertG} \setminus \texttt{Del}_\texttt{V}) \cup prefixed\_set(New_V, \texttt{new})$
    $\texttt{act\_E} : \texttt{EdgeG} := (\texttt{EdgeG} \setminus \texttt{Del}_\texttt{E}) \cup prefixed\_set(New_E, \texttt{new})$
    $\texttt{act\_src} : \texttt{sourceG} := (\texttt{Del}_\texttt{E} \lhd \texttt{sourceG}) \cup$
      {
        $\forall e \in New_E$ *with* $v = sourceR(e) \in New_V$
          | $\texttt{new\_}e \mapsto v$
        $\forall e \in New_E$ *with* $v = sourceR(e) \in RulePreserv_V$
          | $\texttt{new\_}e \mapsto \texttt{m}_\texttt{V}(v)$
      }

$\texttt{act\_tgt} : \texttt{targetG} := (\texttt{Del}_\texttt{E} \lhd \texttt{targetG}) \cup$
{
$\forall e \in New_E$ *with* $v = targetR(e) \in New_V$
$\texttt{new\_}e \mapsto v$
$\forall e \in New_E$ *with* $v = targetR(e) \in RulePreserv_V$
$\texttt{new\_}e \mapsto \texttt{m}_\texttt{V}(v)$
}
$\texttt{act\_tV} : \texttt{tG}_\texttt{V} := (\texttt{Del}_\texttt{V} \lhd \texttt{tG}_\texttt{V}) \cup$
{
$\forall v \in New_V$ *with* $tv = tR_V(v)$.
$\texttt{new\_}v \mapsto tv$
}
$\texttt{act\_tE} : \texttt{tG}_\texttt{E} := (\texttt{Del}_\texttt{V} \lhd \texttt{tG}_\texttt{E}) \cup$
{
$\forall v \in New_E$ *with* $te = tR_E(e)$.
$\texttt{new\_}e \mapsto te$
}
end

The result of an SPO rule application for matches that satisfy the gluing condition is the same as DPO. It is possible to observe that Def. 21 is equivalent to Def. 20, for the DPO-approach. It just adds the gluing condition and replaces the results of the assignments by concrete sets.

Application of rule $r1$ is translated to an event of the event-B model in Fig. B.13. This event may happen when there are concrete values for variables that satisfy all the guard conditions of the event. The actions update the state graph (graph G) according to the rule. In this example two new edges (of types `Act` and `Msg`) are created. The source and target vertices of `Act11` and `Msg11` are, respectively, the images of vertices `N11` and `N12` under $\texttt{m}_\texttt{V}$.

Finally, the event-B model of a graph grammar with NACs is obtained by translating the type graph and the LHS of rules to the event-B context (following Defs. 17 and 19), characterizing the sets that define a state-graph as variables, describing the types of the variables as model invariants and translating the initial graph to the initialization event (according to Def. 18) and mapping each rule application to an event (Def. 20).

Definition 22 (Event-B Model of a Graph Grammar with NACs). *Given a graph grammar with negative application conditions $GG = (T, G0, R)$, **the event-B model describing this graph grammar** $\mathcal{T}^{GG}$ is defined by the event-B model composed of: the context, which is obtained as described in translations $\mathcal{T}^T$ (Def. 17) and $\mathcal{T}^P$ (Def. 19), for each $r \in R$ with pattern rule P; and the machine, which is obtained by the state definition and initial graph translation $\mathcal{T}^{G0}$ (Def. 18) and the translation $\mathcal{T}^r$ (Def. 20 or Def. 21), for each $r \in R$.*

We used the Rodin Platform [24, 6] to specify the example and develop the mathematical proofs[10]. The support offered for theorem proving through the platform allows one to: browse the proof structure; select hypotheses and lemmas to be used; invoke different provers integrated to the platform; define and select tactics to be used; among others [6]. In Sect. 7 we discuss how to state and prove properties about the generated event-B models.

6. Dealing with Attributed Graphs

To be useful in practical applications, graph grammars have been extended by the notion of *attributes*, which are basically data values associated to vertices and/or edges of graphs. The resulting notion of an attributed graph has thus two components: a structural (or graphical) part (a graph) and a data part. These components are typically linked by attribution functions, attribute nodes and/or edges. Depending on the approach, the use of variables and terms as attributes in rules as well as constraints over these variables may be allowed, giving the specifier a more suitable level of abstraction with respect to grammars over non-attributed graphs.

[10]The full specification including all proofs can be found at `https://goo.gl/CUjek1`

There are many ways to define attributed graphs within the algebraic approach to graph grammars. There are approaches that keep the graph and the data parts as distinct components (for example, a graph and an algebra), linked in some way, (e.g. [31],[41]) and approaches that try to cope with both in an uniform way, viewing both structures as graphs (e.g. [49]) or as data types (e.g. [54]). Another aspect that distinguishes the approaches is how changes in attribute values are modeled. Some do not allow changes of attributes (the vertex/edge whose attribute should change must be deleted and re-created with the new value), some use special attribute edges/nodes to allow such changes, and others do relabeling via the use of partial attribute functions.

In this paper, our aim is to use an attribute notion that allows to prove properties about the structural part of the graphs involved in a graph grammar (like the ones proven in the last section) as well as properties involving attributes. The definition of attributed graph should be such that any property that was proven over the structural part would still hold when attributes are added, without having to do any change in the corresponding properties or proofs. To accomplish this, we must leave the existing notion of graph unchanged and construct the attribution layer on top of it. An attributed graph should be seen as a refinement of the underlying non-attributed graph, in the sense that all behaviors that are possible in the refined system can be translated to possible behaviors of the original system. That is, adding attributes does not add behavior concerning the structural (graphical) part of the graphs. Moreover, although graphs must not necessarily be finite for theorem proving purposes, we need a finite way to define the translation when a rule with attributes is enabled. And finally, the notion of attribute should be applicable to the SPO and DPO approaches. With these requirements in mind, we review some of the existing approaches to attributed graphs.

One of the first approaches for attributed graphs was presented in [54]. There, both the structure and the data part were modeled as algebras. The resulting framework, however, was not very rich since categories of algebras (especially with partial functions) do not exhibit many properties that are needed for the rewriting theory of the algebraic approach. An approach to perform verification of attributed GTS based on the ideas of [54] was presented in [50]. In this approach, there is an attribution function mapping elements from the graphical part to the data part of the graph (here the data part is not seen as an algebra, but rather as a set of values). The disadvantage is that it is not possible to change the value of the attribute of a vertex without deleting this vertex (because a simple change of attribute would not be compatible with the original attribution function, and this compatibility is a requirement for the definition of morphisms). In [50] this drawback does not play a role since only edges are attributed, and all edges belonging to the left-hand side of a rule must be deleted.

In previous versions of GROOVE [49] data values were modeled as term graphs. In this approach, rewriting takes place at two different levels: normal graph rewriting for the graphical part and term graph rewriting for the data part of the attributed graph. However, modeling data types as terms has some disadvantages: many data types can be more naturally expressed as “textual" terms; resolution for many of the most used equational systems (like natural numbers, booleans, strings, lists, ...) is already efficiently implemented, whereas there are some limitations for term graph rewriting. Moreover, this technique presented for GROOVE is for finite state graph transformation systems. The current implementation is based on the notion described in [31] that is discussed below.

There are a series of papers that use partial functions to describe attributes, and the notion of relabeling [41] to change the values associated to vertices/edges. In such approaches, rules must be spans, since it is mandatory that in the interface graph of the span (K) the attributes that will be changed by a rule are undefined. Using a (partial) function leads to a simple notion of attribute, suitable for situations in which at most one attribute of each kind is allowed (most practical situations are like this). Some examples of this approach for the DPO are [41, 64, 38] and for the SqPO (SesquiPusOUt) is [26].

The approach presented in [31] defines an attributed graph as a graph in which some vertices are actually data values, and some edges are attribution edges, that is, all data values are considered as vertices and there are special edges connecting graphical vertices to these data vertices (however, data values and attribute edges are explicitly separated from "structural" vertices and edges). This approach is simple, very general and has a very nice theory but, for (automated) verification purposes, it is not directly useful because typically data types involve infinite sets of values, and consequently graphs will be infinite structures (because data values are vertices), hindering a faithful implementation in tools. In [60, 59] this framework was extended to allow a more general definition of graph. In this extension, vertices and edges of a graph may be attributed by variables, and graphs have an extra component that specifies constrains on the values of these attributes. Although more general, symbolic graphs are more tractable since the attributes of the graphs are restricted to variables (see [59] for a discussion).

In [22] we presented an approach to attributed graphs in the context of relational graph grammars. This approach

was inspired by both [31] and [50]. On the one hand, there is a special kind of edges of the graph, called *attribute edges* or simply *attributes*, that are used to describe attribution of vertices. But on the other hand, there is a function assigning a data value to each of these attribute edges. This way, it is possible to model change of attribute values in a framework in which graphs are not infinite (because data values need not be part of the graph). Although this approach is interesting because the framework remains unchanged to a great extent, the fact that attributes are actually part of the graph (the set of edges is partitioned into structural and attribute edges) makes it more difficult to reason separately about the structural and the attribute parts of the graph.

The approach that will be presented in the following sections is based on [22], [31] and [59]. We continue the work in [22] and use definition of attributed graph that is basically the one using *E-graphs* of [31]. We allow only variables as attributes in rules, and use equations in the same sense that constraints are used in [59], more general constraints will be discussed in Sect. 9.

6.1. Attributed Graph Grammars

Algebraic specifications may be used to define data types, and algebras to describe the values that may be used as attributes. We assume that the reader is familiar with algebraic specifications (basic concepts will be informally introduced as necessary).

A *signature* $SIG = (S, OP)$ consists of a set S of sorts and a set OP of constants and operations symbols. Given a set of variables X (of sorts in S), the *set of terms* over SIG is denoted by $T_{OP}(X)$ (this is defined inductively by stating that all variables and constants are terms, and then all possible applications of operation symbols in OP to existing terms are also terms). An *equation* is a pair of terms $(t1, t2)$, and is usually denoted by $t1 = t2$. A *specification* is a pair $SPEC = (SIG, Eqns)$ consisting of a signature and a set of equations over this signature. An *algebra* for specification $SPEC$, or $SPEC$-algebra, consists of one set for each sort symbol of SIG, called *carrier set*, and one function for each operation symbol of SIG such that all equations in $Eqns$ are satisfied (satisfaction of one equation is checked by substituting all variables in the equation by values of corresponding carrier sets and verifying whether the equality holds, for all possible substitutions). Given two $SPEC$-algebras, a homomorphism between them is a set of functions mapping corresponding carrier sets that are compatible with all functions of the algebras. The set obtained by the disjoint union of all carrier sets of algebra A is denoted by $\mathcal{U}(A)$.

In this paper we will only define attributes of vertices; attributes of edges can be defined analogously. The following definition is basically the same of E-graphs in [31], where we call the set of attribute edges $AttrG$, and the corresponding source and target functions $attrvG$ and $valG$, respectively.

Definition 23 (Attributed Graph). *Given a specification $SPEC$, an **attributed graph** is a tuple $AG = (G, A, AttrG, valG, attrvG)$ where $G = (VertG, EdgeG, sourceG, targetG)$ is a graph, A is a $SPEC$-algebra, $AttrG$ is a set, called **set of attributes**, and*

$$valG\colon AttrG \to \mathcal{U}(A), \quad attrvG\colon AttrG \to VertG$$

*are total functions. Elements of $AttrG$ are called **attribute edges**, or just **attributes**.*

*A **(partial) attributed graph morphism** g between attributed graphs AG and AH is a triple $g = (g_{Graph}, g_{Alg}, g_A)$ consisting of a graph morphism $g_{Graph} = (g_V, g_E)$, an algebra homomorphism g_{Alg} and a partial function $g_A : AttrG \to AttrH$ between the corresponding components that are compatible with the attribution:*

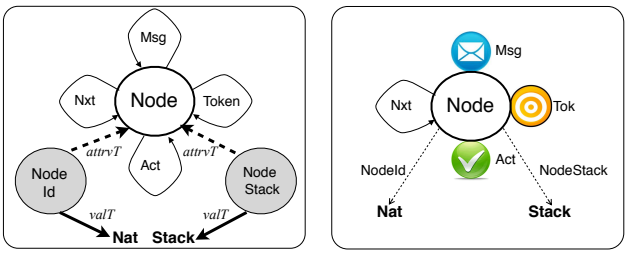
$$\forall a \in dom(g_A) \cdot g_{Alg}(valG(a)) = valH(g_A(a)) \text{ and} \qquad \texttt{(grd\_val)}$$
$$g_V(attrvG(a)) = attrvH(g_A(a)) \qquad \texttt{(grd\_attrv)}$$

An attributed graph morphism g is called total or injective if all components are total or injective, respectively.

The category of attributed graphs and partial attributed graph morphisms is well-defined because it is an instance of a generalized graph structure [51]. Pushouts can be constructed analogously to pushouts in $\mathbf{TGraphP}(\mathbf{T})$: constructing pushouts in the underlying categories ($\mathbf{GraphP}$, $\mathbf{Alg}$ and $\mathbf{SetP}$) followed by a free construction that removes all elements of the pushout of attribute edges $AttrH$ such that $attrvH$ and $valH$ (obtained by the universal property of corresponding pushouts) become total functions.

The role of the type graph is to define the types of vertices and edges of instance graphs. It is thus adequate that the part of the type graph describing data elements consists of names of types. Therefore, we require that the algebra of the type graph is a final one, that is, an algebra in which all carrier sets are singletons. In practice, we will use the name of the corresponding sort as the only element in a carrier set representing it. With respect to the attributes, there may be many different kinds of attributes for the same vertex, and this is described by the existence of many of such edges connected to the same vertex of the type graph.

For example, Fig. 5(a) shows a type graph T with two types of attributes, one natural number and one stack. The set of attributes is $AttrT = \{\mathsf{NodeId}, \mathsf{NodeStack}\}$, and the algebra is the final algebra corresponding to the (conditional) algebraic specification shown in Fig. 6 (defining the types natural numbers, booleans and stack and some other operations that will be used in the example). The bold solid arrow describes function $valT$ and the bold dashed arrow defines $attrvT$. For a better visualization, throughout the paper we will use the notation shown in Fig. 5(b) to represent this type graph.



(a) Attributed type graph T (b) Graphical notation

Figure 5: Type graph for the token-ring protocol with attributes

TRing : **sorts** Bool, Nat, Stack
opns
true : → Bool
false: → Bool
0 : → Nat
succ : Nat → Nat
≠ : Nat × Nat → Bool
empty : → Stack
push : Nat × Stack → Stack
pop : Stack → Stack
top : Stack → Nat
notTwice : Stack → Bool
eqns
∀n ∈ Nat, s ∈ Stack:
≠(0,0)=false
≠(0,succ(n))=true
≠(succ(n),0)=true
≠(succ($n1$),succ($n2$))=≠($n1$,$n2$)
pop(push(n, s))= s
pop(empty)=empty
top(push(n,s))= n
top(empty)=0
notTwice(empty)=true
notTwice(push(n, empty))= true
($top(s) \neq n \vee n = 0$) ⇒ notTwice(push(n,s))= notTwice(s)
notTwice(push(n,push(n,s)))= false

Figure 6: Specification $SPEC_{\mathbf{TRing}}$

Graph $G0$ (Fig. 7) is typed over T (the morphism is given by using the same graphical notation to mapped items). The morphism on the algebra component is not shown: the (final) algebra of T has as carrier sets $T_{Nat} = \{\mathsf{Nat}\}$, $T_{Stack} = \{\mathsf{Stack}\}$ and $T_{Bool} = \{\mathsf{Bool}\}$, and the algebra for $G0$ has $G_{Nat} = \{0, 1, 2, 3, \ldots\}$, $G_{Bool} = \{true, false\}$ and G_{Stack} is the set of terms containing the constructors $empty$, $push$, 0 and $succ$ (isomorphic to the carrier set

of the quotient term algebra for $SPEC_{\textbf{TRing}}$). In this case, there is only one possible homomorphism between the algebras of G and T: mapping all natural numbers to the element Nat, $true$ and $false$ to Bool and all values of G_{Stack} to Stack.

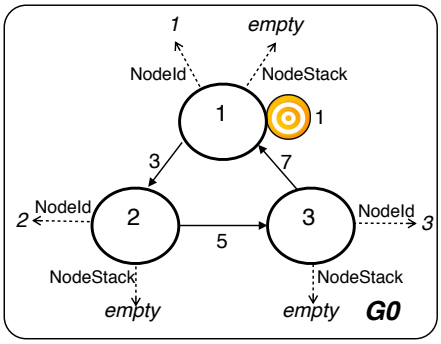


Figure 7: Attributed Graph $G0$ Typed over T

Definition 24 (Attributed Type Graph, Typed Attributed Graphs). *Given a specification $SPEC$, an **attributed type graph** is an attributed graph $AT = (T, A, AttrT, valT, attrvT)$ in which all carrier sets of A are singletons.*

*A **typed attributed graph** is a tuple $AG^{AT} = (AG, tAG, AT)$, where AG is an attributed graph, AT is an attributed type graph and $tAG : AG \to AT$ is a total attributed graph morphism called attributed typing morphism.*

*A **typed attributed graph morphism** between graphs AG^{AT} and AH^{AT} with attributed type graph AT is an attributed graph morphism g between AG and AH such that g only relates elements of the same type, that is*

$$\forall a \in dom(g_A) \cdot tAG_A(a) = tAH_A(g_A(a)) \qquad (\texttt{grd\_tA})$$

Since in the following we will be dealing only with typed attributed graphs, we will omit the word "typed".

Rules specify patterns of behavior of a system. Therefore, it is natural that variables and expressions (terms) are used for the data part of the graphs. We will restrict possible attributes in left- and right-hand sides and in NACs to be variables, and the possible relations between these variables will be expressed by equations associated with each rule. When applying a rule, all its equations will be required to be satisfied by the chosen assignment of values to variables. The following definition is a slight modification of the usual descriptions of rules using attributed graphs. Usually, a quotient term algebra satisfying all equations of the specification plus the rule equations is used as attribute algebra. This gives raise to a simple and elegant definition. However, since here our aim is to find a finite representation of attributed graph grammars in terms of event-B structures, this standard definition is not suitable (in a quotient term algebra, each element of a carrier set is an equivalence class of terms, and this set is typically infinite for many useful data types). Therefore, we use just terms as attributes, that is, we use the term algebra over the signature of the specification as attribute algebra (in the definition below, we equivalently use the term algebra over a specification without equations). In such an algebra, each carrier set consists of all terms that can be constructed using the operations defined for the corresponding sort, functions just represent the syntactical construction of terms (for example for a term t and algebra operation op^A corresponding to an operator op in the signature, we would have $op^A(t) = op(t)$). Consequently, all terms are considered to represent different values in a term algebra, since they are syntactically different. The satisfaction of the equations will be dealt with in the match construction, that is, in the application of a rule. Actually, to obtain a definition that can be translated in a suitable way to event-B, we restrict attributes to be variables, as in symbolic graphs [59], and use equations to relate these variables to terms (symbolic graphs allow more general constraints). In the next definitions we consider only the SPO approach. Definitions for the DPO approach can be obtained analogously.

Definition 25 (Attributed Rule with NACs). *Given a specification $SPEC = (SIG, Eqns)$. An **attributed rule with NACs** over $SPEC$ with type AT is a tuple $attRule = (\alpha, X, RuleEqns, AN(\alpha))$, where*

- *X is a set of variables over the sorts of $SPEC$;*
- *$\alpha : AL^{AT} \rightarrowtail AR^{AT}$ is an injective attributed graph morphism over the specification $(SIG, \varnothing)$, with $AL = (L, T_{OP}(X), AttrL, valL, attrvL)$ and $AR = (R, T_{OP}(X), AttrR, valR, attrvR)$, in which the algebra component is the identity on the term algebra $T_{OP}(X)$, $rng(valL) \subseteq X$ and $rng(valR) \subseteq X$ (the algebra remains unchanged by rule applications, LHS and RHS only contain variables);*
- *$RuleEqns$ is a set of (conditional) equations using terms of $T_{OP}(X)$;*
- *$AN(\alpha) \subseteq \mathcal{AMOR}(AL^{AT})$ is the set of NACs, where $\mathcal{AMOR}(AL^{AT})$ denotes the set of all total attributed graph morphisms over the specification $(SIG, \varnothing)$ from AL^{AT} to graphs (containing only variables as attribute values) typed over AT that are not isomorphisms such that the algebra component is the identity.*

Given an attributed rule with NACs as above, we define the following sets:

***Deleted attributes of** AL:* $RuleDel^{\alpha}_{A} = AttrL \setminus dom(\alpha_A)$;

***Preserved attributes of** AL :* $RulePreserv^{\alpha}_{A} = AttrL \setminus RuleDel^{\alpha}_{A}$;

***Created attributes of** AR :* $RuleCreate^{\alpha}_{A} = AttrR \setminus rng(\alpha_A)$;

***Forbidden attributes of** $NACj$:* $NACj_A = AttrL^{NACj} \setminus rng(lj_A)$, *for all* $lj \in AN(\alpha)$

***Forbidden identifications of attributes**:* $NACj_{idA} = \{(a1, a2) | \{a1, a2\} \subseteq AttrL \text{ and } lj_A(a1) = lj_A(a2)\}$

Note that in this definition we do not require that variables that appear only in the right-hand side of the rule be involved in equations. The effect of such a situation in a rule application is that a value for the corresponding attribute will be generated non-deterministically.

An example of attributed rule is shown in Fig. 8. Variables nid and ns are used as attribute values in the left-hand side of this rule. There is one equation that restricts the application of the rule. The equation states that the rule can only be applied if the NodeId attribute of node N11 is not on the top of the corresponding NodeStack. By abuse of notation, we write top(ns) $\neq$ nid instead of $\neq$ (top(ns), nid) = true. The NAC of this rule does not impose any additional restriction on attributes. To apply this rule, besides finding a match for the graphical part of the rule that satisfies the NAC, we have to find an assignment of values to variables nid and ns that satisfies the equation of the rule. The effect of the application on attributes will be that the value of the $NodeStack$ of the node corresponding to N11 (N13 in $R1$) will be updated to include the node's identifier (nid).

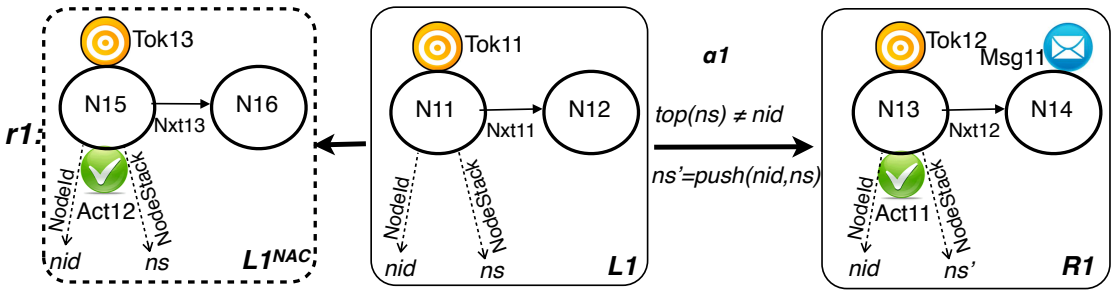


Figure 8: Attributed Graph Rule with NACs

An attributed graph grammar with respect to some specification of data types $SPEC$ is composed of an *attributed type graph*, an *initial graph* and a *set of rules*.

Definition 26 (Attributed Graph Grammar). *Given a specification $SPEC$ and a $SPEC$-algebra A, a **(typed) attributed graph grammar** is a tuple $AGG = (AT, AG0, R)$, such that AT (the type of the grammar) is an attributed type graph over $SPEC$, $AG0$ (the initial graph of the grammar) is an attributed graph typed over AT using algebra A, and R is a set of rules over $SPEC$ with type AT .*

In order to define a match, we have to relate, additionally to the graph morphism, the variables of the rule to the actual values of carrier sets of the algebra of the graph in which the rule shall be applied. Moreover, the match construction must ensure that all equations of the specification and the rule equations are satisfied by the chosen assignment of values to variables. This will be achieved by first, lifting the rule to a corresponding one having a quotient term algebra as attribute algebra. This is a standard construction in algebraic specification. Then, the actual match will include an algebra homomorphism from this quotient term algebra to the actual algebra used in the graph to which the rule is being applied. The existence of this homomorphism guarantees that all necessary equations are satisfied. The NAC satisfaction for attributed matches is analogous to that specified in Def. 9.

Definition 27 (Attributed Match, NAC Satisfaction). *Let a specification* $SPEC = (SIG, Eqns)$, *a rule with NACs over* $SPEC$ $attRule = (\alpha, X, RuleEqns, AN(\alpha))$, $\alpha : AL^{AT} \rightarrowtail AR^{AT}$, *with* $AL = (L, T_{OP}(X), AttrL, valL, attrvL)$, *and a* $SPEC$ *attributed graph* AG^{AT} *be given. An* ***attributed match*** $m : \overline{AL^{AT}} \rightarrow AG^{AT}$ *is a total attributed graph morphism* $m = (m_{Graph}, m_{Alg}, m_A)$ *such that* $\overline{AL^{AT}} = (L, T_{eq}(X), AttrL, \overline{valL}, attrvL)$, *where* $T_{eq}(X)$ *is the algebra obtained by constructing the quotient term algebra of the specification* $(SIG, Eqns \cup RuleEqns)$ *using the set of variables* X, *and, for all elements* $a \in AttrL$, $\overline{valL}(a) = [valL(a)]$. *An* ***attributed match satisfies a NAC*** $lj \colon AL^{AT} \rightarrow {AL^{NAC_j}}^{AT}$, *if there is no a total attributed graph morphism* $n \colon {AL^{NAC_j}}^{AT} \rightarrowtail AG^{AT}$ *that is injective on forbidden items such that* $n \circ lj = m$. *An* ***attributed match satisfies all NACs of a rule*** *if it satisfies each individual NAC of the rule.*

Practically, given a set of variables X and an algebra A, if we define an evaluation function $eval : X \rightarrow \mathcal{U}(A)$, there is a unique way to construct the algebra homomorphism (in case it exists for this assignment). First, we check whether all equations in $Eqns \cup ruleEqns$ are satisfied by this assignment. If not, this assignment of values to variables can not lead to an algebra homomorphism, and thus no match can exist using this $eval$ function. Otherwise, we build the extension of $eval$ to (equivalence classes of) terms, that will be denoted by $\overline{eval} : T_{eq}(X) \rightarrow \mathcal{U}(A)$. This is the homomorphism we are looking for.

Definition 28 (Deleted, Preserved, Created Items). *Let* $r = (\alpha, X, RuleEqns, AN(\alpha))$ *be a (typed) attributed rule with left-hand side* AL, *right-hand side* AR *and* m *be a match* $m = (m_V, m_E, m_{Alg}, m_A) : AL \rightarrow AG$. *Then we define the following sets:*

Deleted attributes of AG**:** $Del_A^{\alpha,m} = m_A[RuleDel_A^{\alpha}]$

Dangling attributes of AG**:** $Dangling_A^{\alpha,m} = dom(attrvG \rhd Del_V^{\alpha,m}) \setminus Del_A^{\alpha,m}$

New attributes of AR**:** $New_A^{\alpha,m} = \{a \in RuleCreate_A^{\alpha} \mid m_V(attrvR(a)) \in Preserv_V^{\alpha,m} \ \vee \ attrvR(a) \in New_V^{\alpha,m}\}$

The definition of attributed rule application is an extension of Def. 15, where the algebra component of the resulting graph is the same as the original graph (since rules do not change the algebra) and the attributes are obtained in the same way as (graphical) edges of the graph (i.e. analogous to the corresponding edge components in Def. 15).

Definition 29 ((Attributed) Rule Application). *Given a specification* $SPEC$, *a rule over* $SPEC$ *with type* AT $attRule = (\alpha, X, ruleEqns, AN(\alpha))$ *with* $\alpha : (L, T_{OP}(X), AttrL, valL, attrvL) \rightarrowtail (R, T_{OP}(X), AttrR, valR, attrvR)$, *and a match* $m : (L, T_{eq}(X), AttrL, \overline{valL}, attrvL) \rightarrow (G, AlgG, AttrG, valG, attrvG)$ *which satisfies all NACs in* $AN(\alpha)$, *the* ***application*** *of rule* $attRule$ *at match* m *results in the typed attributed graph* AH, *with* $AH = (H, AlgH, AttrH, valH, attrvH)$, *where*

H *is the resulting graph of applying rule* $L \rightarrow R$ *to graph* G *(as in Def. 15);*

$AlgH = AlgG$;

$$AttrH = (AttrG \backslash (Del_A^{\alpha,m} \cup Dangling_A^{\alpha,m})) \uplus New_A^{\alpha,m} \qquad \text{(act\_Attr)}$$

$$valH = ((Del_A^{\alpha,m} \cup Dangling_A^{\alpha,m}) \lhd valG) \cup \{a \mapsto valR(a)) \mid a \in New_A^{\alpha,m}\} \quad \text{(act\_val)}$$

$$\begin{aligned} attrvH \quad = \quad & ((Del_A^{\alpha,m} \cup Dangling_A^{\alpha,m}) \lhd attrvG) \cup \\ & \{a \mapsto m_V(attrvR(a)) \mid a \in New_A^{\alpha,m} \wedge attrvR(a) \in RulePreserv_V^{\alpha}\} \cup \\ & \{a \mapsto attrvR(a) \mid a \in New_A^{\alpha,m} \wedge attrvR(a) \in New_V^{\alpha,m}\} \end{aligned} \quad \text{(act\_attrv)}$$

$$tH_A = ((Del_A^{\alpha,m} \cup Dangling_A^{\alpha,m}) \lhd tG_A) \cup \{a \mapsto tR_A(a)) \mid a \in New_A^{\alpha,m}\} \quad \text{(act\_tA)}$$

The gluing condition of Def. 13 can be extended to consider attributes in order to guarantee that all attributes of a deleted vertex are also deleted. For rule applications using SPO this condition is optional whereas using DPO is mandatory.

Definition 30 (Gluing Condition for Attributes). *Let $attRule = (\alpha, X, ruleEqns, AN(\alpha))$ be a typed attributed rule, with $\alpha : AL^{AT} \rightarrowtail AR^{AT}$. Let AG^{AT} be a typed attributed graph and $m = (m_{Graph}, m_{Alg}, m_A) \colon AL^{AT} \to AG^{AT}$ be an attributed match. We say that m* ***satisfies the gluing condition for attributes*** *if it satisfies the following two conditions:*

Identification condition: *There is no conflict between deletion and preservation of attributes and there is no identification of deleted attributes, that is*

$$Del_A^{\alpha,m} \cap m_A[RulePreserv_A^{\alpha}] = \varnothing, \quad \text{(grd\_Ident1A)}$$
$$card(Del_A^{\alpha,m}) = card(RuleDel_A^{\alpha}). \quad \text{(grd\_Ident2A)}$$

Dangling condition: *There are no attributes that will be deleted that are not specified by the rule, that is*

$$Dangling_A^{\alpha,m} = \varnothing \quad \text{(grd\_DangCA)}$$

6.2. Example: Extension of the Token Ring

Now we will extend the example presented in Sect. 4.1. The extension is shown in Fig. 9. Each vertex will have two attributes: NodeId and NodeStack, of types Nat and Stack, respectively. The first attribute represents the identity number of the node, whereas the second holds a record about the messages received/sent by each node. This is done in the following way: each time a node creates a message in the ring, it stores its identifier in its stack (rule r_1); when it receives a message generated by another node, it stores a zero in its stack (rule r_3). Rule r_1 describes the situation in which a node generates a message in the ring. It can only be applied if the node that is generating the message is not on top of its stack attribute, that is, if the node was not the last one that generated a message. The aim of this procedure is to ensure a kind of fairness property for this system. When the last message was generated by a different node, a zero will be stored on the top of the stack. Rule r_5 generates new nodes in the ring. Here, a new NAC was added to the original rule r_5 to ensure that there is no node with identifier nid in the ring when the rule is applied. Note that the equations require that nid is a (non-zero) fresh identifier: it is different from identifiers nid1, nid2 and to any nid3 (here we use equality because nid3 belongs to the NAC, this equation has the purpose of extending the forbidden context to attributes). The effect of this rule application is not deterministic, since many values for nid may satisfy the equations. In rule $r6$ the attributes of the deleted node were added to the LHS to enable rule application (otherwise this rule would not be applicable in the DPO setting because there would be dangling attributes). Since NACs must include the LHS of the rule, these attributes were also included in the NACs. Rules r_2 and r_4 are not changed, and therefore are not shown in this figure.

Note that this grammar is a strict extension of the one without attributes shown in Fig. 3. We just added the attribute structure, but neither new vertices nor edges were inserted. This way, we make sure that all properties that were proven for the original system remain true, and need not be verified again for the modified system. This will become evident in the next section, when we define the translation to event-B.

For this system, additionally to the already discussed and proven properties, we would like to prove:

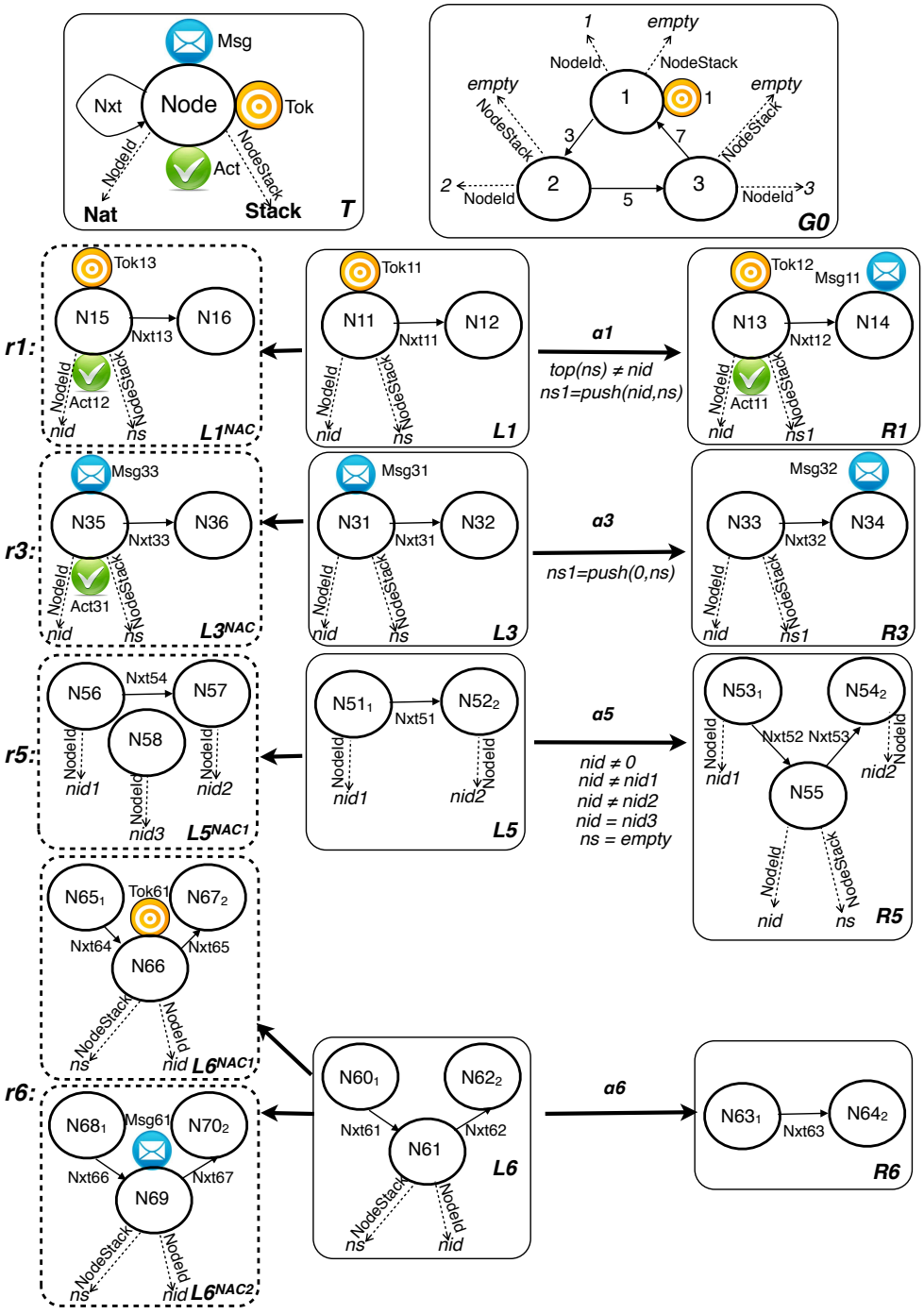


Figure 9: Token Ring with Attributes

Attribute uniqueness (uniqueAttr): Each node has at most one attribute of each kind (one NodeId and one NodeStack).

Attribute completeness (allAttr): Each node has all defined attributes (NodeId and NodeStack).

Fairness (fairness): A node does not generate two messages in a row in the ring.

Not null identities (notNullId): Identities of nodes are different from zero.

Uniqueness of identities (diffId): Nodes have different identities.

6.3. Translation to Event-B

The attribute layer of a graph grammar will be implemented as an event-B refinement of the layer defining the structural part of the graph grammar (defined in Sect. 5). A refinement allows to (i) add new data types, (ii) add new (state) variables, (iii) refine state variables to obtain a more detailed state, (iv) refine the existing events, and (v) add new events. Refinements generate proof obligations to guarantee that the refined system does not add new computations to the original system affecting refined structures or variables, that is, if a system Sys is refined to a system RefSys, any computation of RefSys can be translated to a valid computation of Sys by removing events and variables that were included in the refinement.

The translation of an attributed graph grammar will be done according to the following steps:

AT1 Translate the structural part of the graph grammar;

AT2 Translate the data types;

AT3 Translate the type graph attributes;

AT4 Translate the state graph and initial graph attributes; and

AT5 Translate the attributed rules.

Step AT1. Translation of the structural part of an attributed graph grammar. This step is accomplished by following the steps **T1**, **T2** and **T3** of translation defined in Sect. 5.

Step AT2. Translation of the data types of an attributed graph grammar. In this step, we translate the algebraic specification of the attributed graph grammar by the definition of corresponding types in the context of an event-B project. This context is formally declared as an extension of the one without attributes (defined in Step **AT1**). Event-B has three pre-defined data types: booleans, natural numbers and integers. If these data types are used as attributes of the grammar, we suggest to use directly the pre-defined types instead of constructing new ones (because there are already a lot of operations available and dealing with these types in proofs is easier). Operations over these types that are needed in the specification and are not pre-defined must be included in the event-B context (analogously to the definition of operations over newly declared types described below).

Definition 31 (Translation of Data Types). *Given a specification* $SPEC = (S, OP, Eqns)$*, the translation* $\mathcal{T}_A^{SPEC}$ *of the data types from a specification* $SPEC$ *is defined by the following context elements*

sets

$\forall s \in S$
$\quad \mid s$

constants

$\forall op_s \in OP \vee op_{w,s} \in OP$
$\quad \mid op$

axioms

$\forall op_s \in OP$

| axm_op: $op \in s$

$\forall op_{w,s} \in OP$, *with* $|w| = n$

| axm_op: $op \in w(1) \times \ldots \times w(n) \rightarrow s$

$\forall (X, L, R) \in Eqns$, *with* $x_{i_{s_i}} \in X$

| $\forall x_1, \ldots, x_n \cdot x_1 \in s_1 \wedge \cdots \wedge x_n \in s_n \Rightarrow L = R$

Each sort of the algebraic specification is declared in the sets block of the event-B context, and operation symbols will be declared as constants. Types of operations (domain and range sets) will then be defined as axioms. Equations of the algebraic specification are also translated to axioms (this is a direct translation, we just have to introduce the corresponding universal quantification of variables in front of each equation). For the specification in Fig. 6 the corresponding event-B context, using the pre-defined data types BOOL (for Bool) and $\mathbb{N}$ (for Nat), is shown in Fig. B.14.

Step AT3. Translation of the type graph attributes. Besides the algebra component, an attributed type graph has a set of attribute edges ($AttrT$), functions $attrv_T$ and val_T connecting attributes to (graph) vertices and values, respectively. These components are also translated to context elements (since the type graph does not change during a graph grammar computation). $AttrT$ is declared as a set and $attrv_T$ and val_T as constants, together with the concrete values that belong to $AttrT$. Moreover, a set called $DataType$ is defined containing the names of the sorts defined in the specification. These set and functions are then defined in the axioms part of the event-B context.

Definition 32 (Translation of Attributed Type Graph). *Given an attributed type graph* $AT = (T, A, AttrT, valT, elemT)$ *over specification* $SPEC = (S, OP, Eqns)$*, the translation* $\mathcal{T}_A^{AT}$ *of the attributed type graph is defined by the following context elements:*

sets

AttrT

DataType

constants

attrvT

valT

$s \in S$

| sSort

$at \in AttrT$

| at

axioms

axm_AttrT : partition(AttrT, $elements(AttrT)$)

$\forall x, y \in AttrT$ *with* $x \neq y$

| axm_attrTDiffxy: $x \neq y$

axm_data : partition(DataType, $elements(\{sSort | s \in S\})$)

$\forall x, y \in \{sSort | s \in S\}$ *with* $x \neq y$

| axm_dataDiffxy: $x \neq y$

axm_attrvT : attrvT $\in$ AttrT $\rightarrow$ VertT

axm_attrvTdef : partition(attrvT, $elements(attrvT)$)

axm_valT : valT $\in$ AttrT $\rightarrow$ DataType

axm_valTdef : partition(valT, $elements(valT)$)

An example of the translation of an attributed type graph is illustrated in Fig. B.15. Note that the set DataType contains the names of the sorts defined in $SPEC_{\mathbf{TRing}}$. We used the names StackSort, NatSort and BoolSort not to confuse with the names Stack, $\mathbb{N}$ (the pre-defined name of the set of natural numbers) and BOOL (the pre-defined name of the set of booleans), that are the names of the carrier sets of the algebra.

Step AT4: Translation of state graph and initial graph attributes. In this step we extend the state variables of the event-B machine to include the attribute components: set $AttrG$ and functions $attrv_G$, val_G and the typing function tG_A. In the translation to event-B, we created one value function for each different attribute. The axioms define the types of the set and functions and ensure the compatibility conditions of the types (each attribute edge can only be connected with a value of the corresponding type of that attribute). The initial graph is translated to the INITIALIZATION event, where we generate the initial assignments for the variables introduced in this layer.

Definition 33 (Translation of State Graph and Initial Graph Attributes). *Given an attributed type graph* $AT = (T, A, AttrT, valT, elemT)$ *and an initial graph* $AG0 = (G0, A, AttrG0, valG0, elemG0)$ *typed over* AT. *The translation* $\mathcal{T}_A^{AT,AG0}$ *of the attributes of the state and the initial graphs is defined by the following machine elements:*

variables

$\mathtt{AttrG}$

$\mathtt{attrvG}$

$\mathtt{tG_A}$

$\forall x \in AttrT$

$\mid \mathtt{valG}_x$

invariants

$\mathtt{inv\_AttrG}: \mathtt{AttrG} \in \mathbb{P}(\mathbb{N})$

$\mathtt{inv\_attrvG}: \mathtt{attrvG} \in \mathtt{AttrG} \rightarrow \mathtt{VertG}$

$\mathtt{inv\_tGA}: \mathtt{tG_A} \in \mathtt{AttrG} \rightarrow \mathtt{AttrT}$

$\forall a \in AttrT$, *where* $valT(a) = s$

$\mid \mathtt{inv\_valG}a: \mathtt{valG}_a \in \mathtt{AttrG} \nrightarrow s$

// define one valG function for each attribute type. If there is a predefined type in event-B that corresponds to sort s, it must be used.

$\forall a1, a2 \in AttrT$ *with* $a_1 \neq a_2$

$\mid \mathtt{inv\_Diff}a1a2: \mathtt{dom}(\mathtt{valG}_{a1}) \cap \mathtt{dom}(\mathtt{valG}_{a2}) = \varnothing$

// each attribute has only one value associated to it.

$\forall ta \in AttrT$

$\mid \mathtt{inv\_type}ta: \forall \mathtt{a} \cdot \mathtt{a} \in \mathtt{AttrG} \land \mathtt{a} \in \mathtt{dom}(\mathtt{tG_A} \rhd \{ta\}) \Rightarrow \mathtt{a} \in \mathtt{dom}(\mathtt{valG}_{ta})$

// all attributes are associated to values.

events

event INITIALISATION extends INITIALISATION

then

$\mathtt{act\_AttrG}: \mathtt{AttrG} := set(AttrG0)$

$\mathtt{act\_attrvG}: \mathtt{attrvG} := set(attrvG0)$

$\mathtt{act\_tGA}: \mathtt{tG_A} := set(tG0_A)$

$\forall ta \in AttrT$

$\mid \mathtt{act\_valG}ta: \mathtt{valG}_{ta} := set(\{v \mid v \in AttrG0 \land tG0_A(v) = ta\} \lhd valG0)$

end

The event-B machine obtained from the translation of the attributes of the state of the token ring example is shown in Fig. B.16. In the INITIALIZATION event we only show the attributions to new variables, since for the others the attributions remain the same. Although it is possible to use any $SPEC$-algebra to define the values of attributes of the initial graph of a grammar, to be able to define a general translation to event-B one should only use as attributes values of the pre-defined data types (natural numbers and booleans) or terms from corresponding term algebras. In

event-B, pre-defined sets are interpreted as initial models, that is, without junk and confusion (all elements in $\mathbb{N}$ are different and there is no element in $\mathbb{N}$ that is not a natural number), whereas any interpretation is possible as model for the user-defined sets, the only axioms that are assumed as valid for these sets are the ones defined in the context. This is interesting since it allows a great flexibility for the implementation of these sets. If the required properties follow from the axioms, any implementation that satisfies that axioms will be possible for this specification.

Step AT5: Translation of the attributed rules. The last step of the translation is to describe how attributes of rules are modeled in event-B. Here, instead of creating sets of attributes and corresponding values (variables) in the context, like the translation of sets of vertices and edges of the rules, we adopted a different approach that makes it easier to deal with attributes in proofs. First, variable names for deleted and dangling attribute edges are defined, as well as a variable name for the match component of attributes. Moreover, variable names are defined for each created attribute edge and for each variable used in the rule. The types of these variables must be declared as guards of the event. The guards must also ensure that only assignments of values to these variables that form a match in the sense of Def. 27 enable this event. This is done by requiring the compatibility of these values with the graph and typing structure of the rule. Each equation of the rule shall also be stated as a guard. Finally, the actions of the event involve updating the attribute state variables.

Definition 34 (Translation of an Attributed Rule). *Given an attributed rule $r = (\alpha, X, ruleEqns, AN(\alpha))$, its translation $\mathcal{T}_A^r$ is given by the following context elements and Event, which extends the Event r of the structural part:*

sets

AttrL

constants

$a \in AttrL$

$\mid\ a$

attrvL

tL_A

axioms

axm_Attr$L$: partition(Attr$L$, $elements(AttrL)$)

axm_attrv$L$: attrv$L \in$ Attr$L \rightarrow$Vert$L$

axm_attrv$L$def : partition(attrv$L$, $elements(attrvL)$)

axm_tL_A :t$L_A \in$ Attr$L \rightarrow$AttrT

axm_t$L_A$def : partition(tL_A, $elements(tL_A)$)

Event $r \mathrel{\widehat{=}}$

extends r

any

...

m_A // match attribute edges

Del_A // deleted attribute edges

$Dangling_A$ // dangling attribute edges

$\forall a \in New_A$ // attribute edges created by the rule

$\mid\ new\_a$

$\forall x \in X$ // variables used in the rule

$\mid\ x$

where

...

grd_mA : $m_A \in$ Attr$L \rightarrow$AttrG // total function mapping attribute edges

grd_DelA : $Del_A := m_A[set(RuleDel_A)]$ // set of deleted attributes of G

$\texttt{grd\_DangA}: \texttt{Dangling}_\texttt{A} = \texttt{dom}((\texttt{attrvG} \rhd \texttt{Del}_\texttt{V}) \setminus \texttt{Del}_\texttt{A})$ // dangling attributes of G

$\forall a \in New_A$ // fresh ids of attribute edges

$\quad \texttt{grd\_new\_}a : \texttt{new\_}a \in \mathbb{N} \setminus \texttt{AttrG}$

$\forall a_i, a_j \in New_A$ with $a_i \neq a_j$ // new attribute edges can not have the same id

$\quad \texttt{grd\_diff}a_i a_j : \texttt{new\_}a_i \neq \texttt{new\_}a_j$

$\forall x \in X_s$ // define types of variables

$\quad \texttt{grd\_}x : x \in s$

$\forall a \in AttrL$

$\quad \texttt{grd\_attrv}a : \texttt{m}_\texttt{V}(\texttt{attrv}L(a)) = \texttt{attrvG}(a)$ // compatibility between attr nodes and graph nodes

$\quad \texttt{grd\_tA}a : \texttt{t}L_\texttt{A}(a) = \texttt{tG}_\texttt{A}(\texttt{m}_\texttt{A}(a))$ // compatibility of types of attribute nodes of L and G

$\quad \texttt{grd\_val}a : valL(a) = \texttt{valG}_{\texttt{t}L_A(a)}(\texttt{m}_\texttt{A}(a))$ // match values of G to the variables

$\forall NAC_j \in AN(\alpha)$ // NAC satisfaction

$\quad \texttt{grd\_NAC}j$: *Extends Def. 10 by:*

$\qquad \{\overline{a}_1, \ldots \overline{a}_n\} \subseteq \texttt{AttrG} \setminus \texttt{m}_\texttt{A}[\texttt{Attr}L] \wedge$ $(nac7')$

$\qquad \forall a \in NACj_A$ with $tL_A^{NAC_j}(a) = ta$

$\qquad\quad \texttt{tG}_\texttt{A}(\overline{a}) = ta \ \wedge$ $(nac8')$

$\qquad\quad \texttt{valG}_{tG_A(\overline{a})}(\overline{a}) = val \ \wedge$, with $val = valL^{\texttt{NAC}_j}_{tL_A^{\texttt{NAC}_j}(a)}(a)$ $(nac9')$

$\qquad\quad \begin{cases} \texttt{attrvG}(\overline{a}) = \overline{v} & \text{, if } attrvL^{\texttt{NAC}_j}(a) = v \text{ and } v \in NACj_V \\ \texttt{attrvG}(\overline{a}) = \texttt{m}_\texttt{V}(v) & \text{, otherwise, with } v = lj_V^{-1}(attrvL^{\texttt{NAC}_j}(a)) \end{cases}$ $(nac10')$

$\qquad \vee \ (\bigvee_{(a1,a2) \in NACj_{idA}} \texttt{m}_\texttt{A}(a1) \neq \texttt{m}_\texttt{A}(a2))$ $(nac11')$

$\texttt{grd\_Ident1A}: \texttt{Del}_\texttt{A} \cap \texttt{m}_\texttt{A}[set(RulePreserv_A)] = \varnothing$ // Identification condition 1 on attribute edges

$\texttt{grd\_Ident2A}: \texttt{card}(\texttt{Del}_\texttt{A}) = \texttt{card}(set(RuleDel_A))$ // Identification condition 2 on attribute edges

$\texttt{grd\_DangCA}: \texttt{Dangling}_\texttt{A} = \varnothing$ // Dangling condition on attributes

$\forall eqn_i \in RuleEqns$ // Equations of the rule

$\quad \texttt{grd\_eqn}i : eqn_i$

then

...

$\texttt{act\_A}: \texttt{AttrG} := (\texttt{AttrG} \setminus \texttt{Del}_\texttt{A}) \cup prefixed\_set(New_A, \texttt{new})$

$\forall a \in AttrL$ where $ta = tL_A(a)$

$\quad \texttt{act\_val}ta : \texttt{valG}_{ta} := (\texttt{Del}_\texttt{A} \lhd \texttt{valG}_{ta}) \cup set(\{\texttt{new\_}ca \mapsto v | ca \in New_A \wedge v = valR_{tR_A(ca)}(ca)\})$

$\texttt{act\_attrv}: \texttt{attrvG} := (\texttt{Del}_\texttt{A} \lhd \texttt{attrvG}) \cup set(APresV \cup ANewV)$, where

$\quad APresV = \{\texttt{new\_}a \mapsto \texttt{m}_\texttt{V}(v) \mid a \in New_A \ \wedge \ attrvR(a) \in RulePreserv_V \ \wedge \ v = attrR(a)\}$

$\quad ANewV = \{\texttt{new\_}a \mapsto \texttt{new\_}v \mid a \in New_A \ \wedge \ attrvR(a) \in New_V \ \wedge \ \texttt{new\_}v = attrR(a)\}$

$\texttt{act\_tA}: \texttt{tG}_\texttt{A} := (\texttt{Del}_\texttt{A} \lhd \texttt{tG}_\texttt{A}) \cup set(\{\texttt{new\_}a \mapsto ta \mid a \in New_A \wedge ta = tR_A(a)\})$

end

As an example, rule r_1 is translated to the event in Fig. B.17, where only items that are added with respect to the original event are depicted. If there are new NACs for a rule, the corresponding sets have to be introduced in the context (following the definitions of Sect. 5) and also as guards of the corresponding event. In case the rule just changes the value of an attribute, it would be possible to "reuse" the attribute node: in the example, the 4 actions of the then block in Fig. B.17 would be substituted by the action

@act_valGNodeStack $\texttt{valG\_NodeStack} := \{\texttt{mA(NodeStack1)} \mapsto \texttt{ns1}\} \lhd\!\!+ \texttt{valG\_NodeStack}$.

Definition 35 (Event-B Model of an Attributed Graph Grammar with NACs). *Given an attributed graph grammar $AGG = (AT, AG0, R)$ over the specification $SPEC = (S, OP, Eqns)$, **the event-B model describing this graph grammar** is defined by the event-B model that extends the context and refines the machine of the structural part of AGG as described by translations $\mathcal{T}_A^{SPEC}$ (Def. 31), $\mathcal{T}_A^{AT}$ (Def. 32), $\mathcal{T}_A^{AT,AG0}$ (Def. 33) and $\forall r \in R \cdot \mathcal{T}_A^r$ (Def. 34).*

Figure 10 shows the overall structure of the event-B context and machine resulting from the translation of the graph grammar of Fig. 9. Note that the context and machine are an extension resp. refinement of the corresponding components of the event-B model without considering attributes.

context ctx_trAll1
extends ctx_trAll
sets
... - set names defined in Figs. B.14(b), B.15(b) and B.17(b)
constants
... - constant names Figs. B.14(b), B.15(b) and B.17(b)
axioms
... - axioms defined in Figs. B.14(b), B.15(b) and B.17(b)
end

machine mch_trAll1 refines mch_trAll
sees ctx_trAll1
variables
... - variable names defined in Fig. B.16(b)
invariants
... - invariants defined in Fig. B.16(b)
events
Initialisation
... - event defined in Fig. B.16(c)
Event *rule1* $\widehat{=}$
... - event defined in Fig. B.17(c)
... - events corresponding to other rules
end

Figure 10: Translation of a Graph Grammar: Context and Machine

7. Proving Properties

After translating a graph grammar to an event-B model, one may use this model to prove properties. Appendix B presents a summary of the translation of the Token Ring graph grammar to the corresponding event-B model. This summary sets out a roadmap of the steps involved in the translation. Thereafter, the property to be proven may be stated. Properties about reachable states are typically stated as invariants and are proven by induction: in the base case, the property is verified for the initial graph (initialization event) and then, for the inductive step, the property is verified for each rule (event) applicable to a reachable graph that has the desired property. The proof process can be semi-automated. In many cases, the proof of a property may depend on the establishment of a set of other properties or theorems. However, it is important to notice that all these auxiliary theorems can be used as simplification rules, and will probably be reused in future proofs.

7.1. Stating Properties

The first step to verify a property is to formally state this property. In the case of event-B, the properties are specified as state invariants, that is, properties that have to be valid in all reachable states of a system, using First-Order Logic with Set Theory. Properties involve state variables, and may involve also any sets and operations defined in the context of the model. Therefore, to be able to formulate properties about a graph grammar using the approach proposed in this paper, the user must know how the elements of graphs and rules are denoted in the event-B model that corresponds to the graph grammar. Table 2 illustrates the translation of the most common notions related to a graph grammar to the corresponding event-B notation.

These notions can be used to favor the use of specification patterns for properties over graph structures proposed in [23]. That work establishes 17 pattern classes in which functional and structural requirements of reachable states of graph grammars can be classified. The patterns are based on a library of pre-defined functions that describe common characteristics or typical elements of graphs. In order to adopt such proposal to state graph properties, one must identify how to specify the graph components described in this library. Some of these components are the ones listed in Table 2.

Example: Properties about the Graph Structure. The properties that were informally stated for the token ring protocol in Sect. 4.1 can be formalized as:

(uniqueTok) At each time, there is only one token in the ring:
$card(\mathtt{tG_E} \rhd \{\mathtt{Tok}\}) = 1$

Table 2: Event-B Terms

Description	**Event-B Term**
Set of types of vertices	$\texttt{VertT}$
Set of types of edges	$\texttt{EgdeT}$
Source of types of edges	$\texttt{sourceT} : \texttt{EdgeT} \rightarrow \texttt{VertT}$
Target of types of edges	$\texttt{targetT} : \texttt{EdgeT} \rightarrow \texttt{VertT}$
Vertices of graph G	$\texttt{VertG}$
Edges of graph G	$\texttt{EdgeG}$
Source of edges of graph G	$\texttt{sourceG} : \texttt{EdgeG} \rightarrow \texttt{VertG}$
Target of edges of graph G	$\texttt{targetG} : \texttt{EdgeG} \rightarrow \texttt{VertG}$
Type of vertices of graph G	$\texttt{tG}_\texttt{V} : \texttt{VertG} \rightarrow \texttt{VertT}$
Type of edges of graph G	$\texttt{tG}_\texttt{E} : \texttt{EdgeG} \rightarrow \texttt{EdgeT}$
Edge e of type t	$e \in \texttt{EdgeG} \land \texttt{tG}_\texttt{E}(e) = t$
Vertex v of type t	$v \in \texttt{VertG} \land \texttt{tG}_\texttt{V}(v) = t$
Set of edges of type t	$dom(\texttt{tG}_\texttt{E} \rhd \{t\})$
Set of vertices of type t	$dom(\texttt{tG}_\texttt{V} \rhd \{t\})$
Cardinality of the set of vertices	$card(\texttt{VertG})$
Cardinality of the set of edges	$card(\texttt{EdgeG})$
Cardinality of edges of type t	$card(\texttt{tG}_\texttt{E} \rhd \{t\})$
Cardinality of vertices of type t	$card(\texttt{tG}_\texttt{V} \rhd \{t\})$
Loop edge e	$e \in \texttt{EdgeG} \land \texttt{sourceG}(e) = \texttt{targetG}(e)$
Loop edge e of type t	$e \in \texttt{EdgeG} \land \texttt{tG}_\texttt{E}(e) = t \land \texttt{sourceG}(e) = \texttt{targetG}(e)$
Attribute a of type t	$a \in \texttt{AttrG} \land \texttt{tG}_\texttt{A}(a) = t$
Attribute a of type t of vertex v	$a \in \texttt{AttrG} \land \texttt{tG}_\texttt{A}(a) = t \land \texttt{attrvG}(a) = v$
Attribute a of type t of vertex v with value x	$a \in \texttt{AttrG} \land \texttt{tG}_\texttt{A}(a) = t \land \texttt{attrvG}(a) = v \land \texttt{valG}_t(a) = x$

(uniqueAct) At each time, at most one station is transmitting through the network, and it is holding the token bit pattern:

$$card(\texttt{tG}_\texttt{E} \rhd \{\texttt{Act}\}) \leq 1 \land$$
$$(\forall ea \cdot ea \in \texttt{EdgeG} \land \texttt{tG}_\texttt{E}(ea) = \texttt{Act} \Rightarrow$$
$$(\exists et \cdot et \in \texttt{EdgeG} \land \texttt{tG}_\texttt{E}(et) = \texttt{Tok} \land \texttt{sourceG}(et) = \texttt{sourceG}(ea)))$$

Example: Properties about Attributes. Figure 9 shows an extension of the Token Ring example including attributes. Since this model was defined as a refinement of the original model (shown in Fig. 3), all properties already proven are still valid. The fact that this layer is actually a refinement is checked automatically by the Rodin platform. If necessary, proof obligations are generated. In the case of the translation proposed here, no proof obligations to ensure the correctness of the refinement were necessary (because we only added new types and variables and events only act on these new variables).

The properties involving attributes will now be formally stated as invariants. Note that to state these properties it is necessary to know the sets and functions that form the definition of an attributed graph. Moreover, due the way attributes are defined, it is not straightforward to obtain the values of attributes that are associated to a vertex. This definition, modeling E-graphs, is very general, allowing, for example, that each vertex has more than one attribute of each kind. However, for many applications, like in the example, this is not desired, and therefore invariants like the first and the second below are needed (see a discussion on this topic in Sect. 8).

(uniqueAttr) Attribute uniqueness: Any node has at most one attribute of each kind.

$$\forall a1, a2, v \cdot a1 \in \texttt{AttrG} \land a2 \in \texttt{AttrG} \land v \in \texttt{VertG} \land$$
$$a1 \mapsto v \in \texttt{attrvG} \land a2 \mapsto v \in \texttt{attrvG} \land$$
$$\texttt{tG}_\texttt{A}(a1) = \texttt{tG}_\texttt{A}(a2)$$
$$\Rightarrow a1 = a2$$

(allAttr) Attribute completeness: Any node has all defined attributes (NodeId and NodeStack).

$$\forall v \cdot v \in \texttt{VertG} \Rightarrow \\ (\exists aid \cdot aid \in \texttt{AttrG} \land \texttt{tG}_\texttt{A}(aid) = \texttt{NodeId} \land \texttt{attrvG}(aid) = v) \\ \land \\ (\exists as \cdot as \in \texttt{AttrG} \land \texttt{tG}_\texttt{A}(as) = \texttt{NodeStack} \land \texttt{attrvG}(as) = v)$$

(fairness) Fairness: A node does not generate two messages in a row in the ring.

$$\forall a \cdot a \in \texttt{AttrG} \land \texttt{tG}_\texttt{A}(a) = \texttt{NodeStack} \Rightarrow \texttt{notTwice}(\texttt{valG\_NodeStack}(a)) = \mathsf{TRUE}$$

(notNullId) Not null identities: Identities of nodes are different from zero.

$$\forall a \cdot a \in \texttt{AttrG} \land \texttt{tG}_\texttt{A}(a) = \texttt{NodeId} \Rightarrow \texttt{valG\_NodeId}(a) \neq 0$$

(diffId) Uniqueness of identities: Nodes have different identities.

$$\forall a1, a2, v1, v2 \cdot a1 \in \texttt{AttrG} \land a2 \in \texttt{AttrG} \land v1 \in \texttt{VertG} \land v2 \in \texttt{VertG} \land \\ a1 \mapsto v1 \in \texttt{attrvG} \land a2 \mapsto v2 \in \texttt{attrvG} \land \\ \texttt{tG}_\texttt{A}(a1) = \texttt{NodeId} \land \texttt{tG}_\texttt{A}(a2) = \texttt{NodeId} \land \\ v1 \neq v2 \\ \Rightarrow \texttt{valG\_NodeId}(a1) \neq \texttt{valG\_NodeId}(a2)$$

7.2. Proving Properties about Graph Grammars in the Rodin Environment

Given a property p expressed as an invariant in an event-B model obtained from a graph grammar, the kinds of proof obligations (POs) that are (automatically) generated for p are:

Well-definedness (WD): Property p itself is well-defined. Typically, well-definedness POs involve proving that functions are applied to elements belonging to their domains and that cardinality is calculated only for finite sets. The generation of these POs depends on the kind of property being stated, and most of them are proven automatically.

Initialization event establishes p (INIT): The aim of this kind of PO is to guarantee that the initialization event builds a state that satisfies property p. The goal of this PO is thus the property with the variables VertG, EdgeG, sourceG, targetG, tG_V and tG_E substituted by the corresponding values of the initial state. Proofs of **INIT** POs are usually not very difficult because they involve concrete finite sets.

Rule events preserve p (INV): The goal of this PO is to prove that the application of a rule preserves property p, that is, assuming that the property is valid before the rule is applied (induction hypothesis), it is still valid after rule application. The formula to be proven is build by substituting in the property each of the variables by the assignments made to them in the corresponding rule event. Considering the effect of a rule application, the resulting sets of vertices/edges are obtained by deleting some elements and adding new ones (source, target and typing functions are adapted accordingly). Of course, the proof style depends on the kind of property, but we noticed that proofs for rule events are often split in 2 cases: (i) for items that are preserved, usually the induction hypothesis is used; (ii) for newly created items, usually the property is instantiated with them and proved concretely. In the Rodin environment, usually one **INV** PO is generated for each rule, even if the rules perform similar actions. When proving, it is possible to copy the whole or parts of proof trees from one PO to the other.

Depending on the property and the rules, **WD** and **INV** POs may not be necessary (they will not be generated by the Rodin environment in these cases).

To prove **INIT** POs it is necessary to know how the initial graph is encoded in event-B. In our translation, vertices and edges of all reachable graphs are denoted by natural numbers, therefore to aid the proof (if necessary) one has to look at the initialization event, verify which concrete sets of natural numbers were assigned to the sets VertG and EdgeG, as well how the corresponding source, target and typing functions are defined, and reason using these concrete values. Proofs of **INV** POs are usually constructed by reasoning about how the property is affected by deletion and creation of items, because this is the effect of a rule application. To be able to aid the proof one has to know how

elements of the rule's LHS and NACs are translated. For each rule with LHS L, analogous sets and functions as shown in Table 2 are created and may be used in proofs. Additionally, one may need other terms related to the rule application, which are shown in Table 3. In the following we illustrate how to construct proofs for two properties of the running example.

Table 3: Event-B Terms

Description	**Event-B Term**
Match of vertices	$\texttt{mV} : \texttt{VertL} \rightarrow \texttt{VertG}$
Match of edges	$\texttt{mE} : \texttt{EdgeL} \rightarrow \texttt{EdgeG}$
Vertices of G matched by the rule	$\texttt{mV}[\texttt{VertL}]$
Edges of G matched by the rule	$\texttt{mE}[\texttt{EdgeL}]$
Vertices of G marked for deletion	Del_V
Edges of G marked for deletion	Del_E
Dangling edges of G	Dangling
Fresh vertex v created by the rule	new_v
Fresh edge e created by the rule	new_e
Preserved vertices of G	$\texttt{Preserv}_\texttt{V}$
Preserved edges of G	$\texttt{Preserv}_\texttt{E}$
Forbidden vertex v in a NAC	forb_v
Fordibben edge e in a NAC	forb_e
Match of attributes	$\texttt{m}_\texttt{A} : \texttt{AttrL} \rightarrow \texttt{AttrG}$
Attributes of G marked for deletion	Del_A
Dangling attributes of G	Dangling_A
Fresh attribute a created by the rule	new_a
Forbidden attribute a in a NAC	forba

The process of building proofs is analogous to programming: it is strongly advisable to split complex proofs into smaller proofs (usually called lemmas) that can be easily constructed and reused. The best way to split a proof depends on the kind of property that is being proved. For the proof of the Token Ring example, 6 auxiliary lemmas were used (stated as invariants). The well-definedness of formulas involving cardinality requires that sets are finite (cardinality can only be calculated for finite sets). To avoid proving the finiteness of the sets used in properties uniqueTok and uniqueAct many times (in different proof obligations), we stated their finiteness as auxiliary invariants (and thus could use these facts in other proofs, after they were proven). We also stated auxiliary invariants (propTok and propAct) ensuring that the edges of type Tok and Act are always loops (have the same source and target).

(propFin_V) $\mathit{finite}(\texttt{VertG})$

(propFin_E) $\mathit{finite}(\texttt{EdgeG})$

(propFinTok) $\mathit{finite}(\mathit{dom}(\texttt{tG}_\texttt{E} \rhd \{\texttt{Tok}\}))$

(propFinAct) $\mathit{finite}(\mathit{dom}(\texttt{tG}_\texttt{E} \rhd \{\texttt{Act}\}))$

(propTok) $\forall e \cdot e \in \texttt{EdgeG} \land \texttt{tG\_E}(e) = \texttt{Tok} \Rightarrow \texttt{sourceG}(e) = \texttt{targetG}(e)$

(propAct) $\forall e \cdot e \in \texttt{EdgeG} \land \texttt{tG\_E}(e) = \texttt{Act} \Rightarrow \texttt{sourceG}(e) = \texttt{targetG}(e)$

All properties described in this paper were proven using the Rodin tool (version 3.1). Considering the 8 properties without attributes, only properties propTok, propAct, uniqueTok and uniqueAct generated **WD** POs, that were proven automatically. Two (out of 8) **INIT** POs were also proven automatically. The others involved either inserting the supersets of the relations being considered in each case (and select the *finite of a relation* option of

the prover), or instantiating the property with each of the values of the initial state (proof by cases). For property propFin_V only 2 **INV** POs for rules $r5$ and $r6$ were generated (because these are the only rules that modify the set of vertices). For the other properties, one **INV** PO was generated for each rule. All proofs for propFin_V and propFin_E were automatically generated. For propFinTok and propFinAct the proofs were almost automatic (basically, performed by clicking the union sign in the formula and then using the ML prover). For the other properties it was necessary to divide the proof in cases, use the induction hypothesis, and add some hypothesis to help conclude the proof. To give the reader a better insight in how a complex proof can be performed using the Rodin tool, we present the guidelines of the proof of property uniqueTok.

Proof of (uniqueTok):

Basis (INIT): *The graph generated by the initialization event exhibits property* uniqueTok. The prover generates the goal, that is obtained by substituting $\mathtt{tG_E}$ in uniqueTok by its initial value ($\{1 \mapsto \mathtt{Tok}, 3 \mapsto \mathtt{Nxt}, 5 \mapsto \mathtt{Nxt}, 7 \mapsto \mathtt{Nxt}\}$). Then just by adding the hypothesis $\{1 \mapsto \mathtt{Tok}, 3 \mapsto \mathtt{Nxt}, 5 \mapsto \mathtt{Nxt}, 7 \mapsto \mathtt{Nxt}\} \rhd \{\mathtt{Tok}\} = \{1 \mapsto \mathtt{Tok}\}$ the prover can conclude the proof. The corresponding proof tree and goal can be found in Appendix C - Fig. C.18.

Inductive step (INV): *Assuming that the graph representing the current state exhibits property* uniqueTok*, the application of any of the 6 rules (events) preserves this property.* These cases can be grouped into 3 classes (using the tool, each case must be dealt with separately), the goal and commented proof trees for one example of each class can be found in Appendix C:

$r1$**:** *A* Tok *edge is preserved.*
To prove this we just have to add an hypothesis stating that the type of all created edges is not Tok and then the prover concludes the proof using the induction hypothesis (Fig. C.19).

$r2, r4$**:** *A* Tok *edge is deleted and re-created.*
Here we may add hypothesis stating that the created edge is of type Tok (which is automatically proven), and that all preserved edges are not of type Tok. The latter involves proving that mE(Tok21) is of type Tok, where Tok21 is the element of type Tok of the LHS of rule $r2$ that is being deleted. This is true due to the fact that matches are compatible with typing, grd_edges ensures this fact (we instantiate this guard with Tok21). Then using the induction hypothesis the proof is concluded (since the only edge of type Tok is deleted, all preserved items are not of type Tok – see Fig. C.20).

$r3, r5, r6$**:** *No* Tok *edge is needed.*
This proof can be built by adding hypothesis stating that created and deleted edges are not of type Tok. For the latter, it is necessary to instantiate grd_edges with the edges deleted by the rule to show that they are not of type Tok (Fig. C.21). Note that although rule $r6$ that deletes a vertex, the proof is analogous to $r3$ and $r5$ since the dangling condition is satisfied (the corresponding guard of the event must be true to enable it), and thus no edge that is not explicitly marked for deletion by the rule is deleted.

Properties with attributes are proven analogously, for the initialization event and inductively for each rule. As an example, we show how property fairness was proven (commented proof trees can be found in Appendix C).

Proof of (fairness):

Basis (INIT): *The graph generated by the initialization event exhibit property* fairness. This was proven automatically.

Inductive step (INV): *Assuming that the graph representing the current state exhibits property* fairness*, the application of any of the 6 rules (events) preserves this property.* Since this property involves an universal quantifier and an implication, the first step is usually to free the quantified variable and assume the left-side of the implication to be true (this is done by clicking the $\forall$ and $\Rightarrow$ signs in the goal). These 6 different rules can be grouped into 4 classes, the goal and proof tree for one example of each class can be found in Appendix C:

$r1, r3$**:** *A* `NodeStack` *attribute is updated.*
To prove this we divided the proof in cases: (i) for attributes that are preserved, and (ii) for attributes that are generated (updated). For the first case we instantiate the induction hypothesis to prove the validity of the property. For the latter case, we instantiate the induction hypothesis and use the definition of the operation `notTwice` to complete the proof (Fig. C.22).

$r2, r4$**:** Since these rules do not involve attributes, no proof obligation is generated.

$r5$**:** *A* `NodeStack` *and a* `NodeId` *attributes are generated.*
This proof is analogous to rule $r1$ and $r3$, except that for the second case, we further split the proof in cases, considering the new attribute to be `NodeId` or `NodeStack`. For both cases the prover can finish the proof automatically (Fig. C.23).

$r6$**:** *No attributes are updated/generated.*
This case is trivial (the proof is automatically performed).

8. Discussion

The main aim of this paper is to formally define the translation from graph grammars to event-B. The definitions presented in Sects. 5 and 6 are general allowing the translation of any graph grammar (involving only finite structures) to an event-B model. Currently, this translation has to be carried out manually, we are working on the construction of a tool that implements this task. After the graph grammar is translated, the properties to be checked must be stated as invariants in the event-B model. Regardless of the way in which the event-B model is obtained (by manual or automatic translation), when this model is loaded and checked by the Rodin tool, proof obligations (POs) are automatically generated. These proof obligations describe the necessary proofs that have to be carried out to check the correctness of the model, that is, that the model satisfies all desired properties (stated as invariants). The Rodin tool immediately tries to discharge all generated POs, for some it may succeed, for others user aid may be necessary.

For the Token Ring example the number of generated proof obligations (POs) is summarized in Table 4. We divided the POs into 2 classes: *Translation POs* and *Application POs*. The former are POs that were generated for invariants and guards introduced by the translation process (all invariants of Def. 18 and Def. 33, as well as **WD** of guards of events modeling rules). Basically these invariants guarantee a kind of type correctness of the translation: the initial event and all events implementing rules only assign values belonging to the allowed types to the state variables. Translation POs do not need to be proven, since they are true because the translation is well-defined (the result of a rule application is an object of the corresponding category, and the event-B model faithfully represents this object). We have proven all translation POs for the examples presented in this paper to evaluate the effort needed to complete the proofs. Application POs are the ones that represent proofs obligations regarding properties specified by the user. These are the ones that the user actually wants to verify for a given grammar. In the example, we stated 8 properties over the structural part of the Token Ring and 5 over the model involving attributes. Remind that the example has 7 events (one initialization and 6 corresponding to rules). Table 4 shows a summary of the POs that were generated for the case study, classified in translation and application POs. Within each class, column **POs** indicates the total number of POs, **Auto** indicates the number of POs that were automatically discharged by the tool, **Easy** depicts the number of POs that were proven just by a series of clicks on symbols (like $\forall$, $\Rightarrow$, $\vee$, ...) and clicks on some of the Roding provers, and column **Manual** contains proofs in which it was necessary to add hypotheses and/or instantiate hypotheses manually (these are typically the most difficult operations for non-experts when building a proof). Although half of the application POs fall into the **Manual** class, most of the times the proofs for a property were very similar in different events, and it was possible to copy and paste part or complete proof trees in the Rodin tool.

Most POs of properties involving attributes were not automatically proven, and one of the reasons for this is the indirection introduced by the conceptual model of attributes being used (E-graphs), that requires two functions (a relation) to relate an item to its attribute values (in the paper, functions val and $attrv$). This is a very general notion of attribute and therefore any restriction to the structure of reachable graphs must be stated as invariant. For example, two of the properties required for graphs in the case study were (i) *attribute uniqueness*: there may be at most one attribute (edge) of each kind connected to the same vertex (that is, at most one value for this attribute is associated to each vertex), and (ii) *attribute completeness*: in an attributed graph, all attributes of each vertex must be defined (that

Table 4: Proof obligations for the Token Ring example

Event-B Component	Translation POs				Application POs			
	POs	Auto	Easy	Manual	POs	Auto	Easy	Manual
TokenRingContext	0	0	0	0	0	0	0	0
TokenRingMachine	64	60	1	3	56	14	18	24
TokenRingAttContext	32	32	0	0	8	8	0	0
TokenRingAttMachine	70	48	9	13	30	7	0	23

is, once an attribute edge exists in the type graph, there must be a corresponding value in any instance graph). These requirements make sense in practice, since when a list of attributes is defined for a vertex, typically one wants that all vertices of each graph will have values for those attributes (completeness), and these values are unique (uniqueness). If we had a simpler, but more restricted version of attributes, for example, based on an attribution function and relabeling (like in [22], [41], [26]), the number of invariants would be reduced (as well as corresponding proof obligations) and possibly other proofs involving attributes would be simpler (maybe automatically generated).

On the other side, the use of more general constraints, like the ones used in symbolic graphs [59], is possible, and generally would not make proofs more difficult (more constraints on rules mean more restrictions to rule application, which are translated to more guards and will be available as hypotheses when proving properties of the involved event). Any formula in First-Order Logic with Set Theory can be easily translated to event-B. Note that these constraints may even involve the sets of vertices and edges of the graph, giving raise to a rich framework to define rules.

Another issue that seems important when stating properties involving attributes is the need for constructions that play the role of NACs for attributes. For example, a construction that allows to state that a rule may be applied if there is no element of the graph with some specific value as attribute. This may be expressed using the existing framework (like property *uniqueness of identities* above), but it would be interesting for practical applications to provide a syntax for some pre-defined attribute NACs (like, for example, putting *new(nid)* as an equation of the rule to express the fact that nid is not attribute of any vertex of the actual graph). Like this, other properties involving attributes may be pre-defined to ease the construction of specifications.

The size of the Rodin model resulting from the translation proposed in this paper is directly related to the size of the input graph grammar. The context depends on the size of the type graph, the specification of data types and the left hand side of each rule and the NACs, whereas the machine depends on the size of the initial graph and the number of rules. The growth of the event-B model is basically linear. Only when we need to differentiate elements like created vertices/edges, sorts, types of attributes and forbidden elements, the number of axioms, guards and invariants grows quadratically with the number of these elements; however, in general, the number of such elements is very small with respect to the size of a graph grammar. The number of generated application POs for a model obtained from a graph grammar, which are the POs that need to be proven to check correctness of the system with respect to the required properties, is determined by the number of rules and properties to be checked. Considering a graph grammar with n rules, each property to be proven generates in the worst case $n + 2$ proof obligations (POs): 1 well-definedness (**WD**, showing that the property itself is well defined); 1 for the initialization event (**INIT**, to ensure that the initial state of the grammar has the property) and one for each rule (**INV**, to show that the rule preserves the property). Note that not always **WD** and **INV** POs are generated for every property/event.

9. Final Remarks

In this paper we formally define a translation of graph grammars into event-B models, considering negative application conditions and attributes. NACs allow the definition of patterns that prevent rule application whereas attributes allow the construction of more convenient specifications (where not only graphical elements are allowed, but also data types). The use of these features enhances the expressiveness of the rules, generally simplifying specifications. In addition, we show that the rule application construction used in the translation is the pushout object in the category of typed graphs and partial morphisms, proving that our translation preserves the graph grammar behavior.

Compared to other approaches for verification of graph grammars, our approach is suitable for the analysis of systems with very large and complex state descriptions as well as with infinite state-spaces, which is a significant advantage since in graph grammars the states tend to be large and/or have involved internal structures. The proposed verification approach may be used for a great variety of graph grammars used in practical applications. To verify a graph grammar, the first step is to translate the grammar to a corresponding event-B model (in this work this translation was defined, we are working on a tool to implement it). Then, properties are stated as invariants and corresponding proofs are constructed by induction: first, the property is proven for the initial state and then, assuming it is valid for some state, one must prove that each rule of the grammar preserves this property. The proofs may be aided by the theorem provers of the Rodin platform [6], which automatically generates all necessary proof obligations as well as tries to discharge them. Although it is usually not possible to conclude all proofs automatically (since the language used to specify properties is very expressive, see discussion below), the process of constructing the proofs typically gives the developer useful insights into the system under analysis: by knowing which hypotheses are needed to prove a property, the developer understands how design decisions (expressed as graphs, attributes, rules, etc.) have influence on this property. Among other approaches that aim to prove properties of graph grammars, Tran and Percebois's work [76] is the closest to the one proposed in this paper. It relies also on our previous work [70, 22], but extends it in a different way: the aim is to give conditions that guarantee that a property is satisfied by a rule that will be applied at a certain graph (given a graph G, a rule r without NACs and a property P, they specify for some classes of properties what must hold for their preservation). Preservation of properties relies essentially on how elements introduced/deleted by the rule are matched in the graph. Instead we propose to prove properties based only on the property and rule structure: if we prove that a property P is preserved by rule r (with or without NACs), P will be preserved for any application of rule r to any graph. Another approach that proposes to prove properties of graph grammar using theorem proving was presented in [72], where the authors give a translation of graph transformations to Isabelle using path expressions as an alternative means to describe rule applicability. Some hints of how to prove structural properties and state well-formed graph and preservation properties are given. Although this approach does deal with NACs (in a non standard way – using path expressions), attributed graphs are not addressed. In both approaches [76, 72], the authors gave examples of encodings of graph transformation systems in the Isabelle theorem prover, but no general and complete translation was proposed.

Concerning the kind of properties that can be proven, in this work we only considered invariants, but it would also be possible to use the proposed translation to prove properties using variants[11] (like termination, for example). This is also supported by event-B and corresponding tools. All properties handled with invariants are properties over reachable states. However, it would also be possible to prove properties over computations (like reachability) by including new variables in the state of the machine, e.g. a *trace variable*, and reasoning about these variables (in the same way it is done for CSP and Unified Theories of Programming [58], for example). In our approach invariants in event-B must be described as formulas in First-Order Logic with Set Theory. Although great variety of interesting graph grammar properties may be expressed in this framework (Set Theory provides a framework comparable to Second Order Logic [14, 77]), different logics may provide more suitable constructs for describing some properties, like Temporal Logic [55], which is tailored to analyze computations, or Linear Logic [37], which is suitable to reason about resources. Besides the expressive power to state properties, three other reasons for the choice of event-B and the Rodin tool were (i) the semantics of event-B is very close to the semantics of graph grammars: the basic semantical construction of graph grammars, the derivation step, can be faithfully interpreted by the built-in notion of occurrence of an event-B event; (ii) given a property to be verified, the proof is automatically split in cases (for the initial state and for each rule, if necessary) and corresponding proof obligations are generated; and (iii) Rodin offers a very intuitive graphical interface for constructing proofs, showing the proof tree, hypothesis that may be used, goal, and offering at each point possible actions just by clicking corresponding symbols of the formulas (for example, $\forall$ for instantiation, $\Rightarrow$ for modus ponens or modus tollens, $\vee$ for disuntcion or to introduce proof by cases, etc.). Moreover, different provers may be used inside the Rodin environment. It would be interesting to investigate how to perform encodings of graph grammars to enable the use of theorem provers based on Higher-Order Logic, like Isabelle [57] or Coq [13], for example. In these cases, however, the notion of derivation step can not make use of a built-in notion of event or transition, making the translation more involved.

[11] A variant in event-B is an expression whose value must decrease with the occurrence of each (convergent) event [4].

During the verification process, we noticed that it might be a very complex task to state the desired property as well as to develop the proof, even for developers with a strong theoretical background. A property over a complex graph structure is not always straightforward to define, even using First-Order Logic, which is a language that is reasonably well understood. Therefore, to enable the use of theorem provers in practice, we suggest the construction of patterns of properties for graph grammars, as well as patterns for corresponding proofs. These patterns could be a great aid for the developer both to state the *right* property and develop its proof. Initial ideas about patterns and proof strategies for graph grammars can be found in [23] and [48, 47], respectively. We plan to improve these works, integrate the approaches and extend them to consider NACs and attributes.

Currently, we are investigating the definition of a theory for graphs that may ease the process of constructing proofs for graph-like structures in the Rodin platform using the Theory-plug-in. We are also working on an implementation of the translation proposed in the present paper. As future work we can also extend the translation to event-B to support more general approaches like AGREE [17] and Symbolic Graphs [60, 59]. The first one provides the possibility of specifying copy of edges as side effect of copy of nodes and the second one allows the definition of more general restrictions (not only equations) for rule applications.

Acknowledgement

The authors gratefully acknowledge financial support received from FAPERGS and CNPq.

References

[1] *Promela language reference*. http://spinroot.com/spin/Man/promela.html.

[2] 2014. A heuristic solution for model checking graph transformation systems. *Applied Soft Computing*, **24**, 169 – 180.

[3] Abrial, J. R. 2005. *The B-book: Assigning programs to meanings*. Cambridge University Press.

[4] Abrial, Jean-Raymond. 2010. *Modeling in event-B: System and software engineering*. Cambridge University Press.

[5] Abrial, Jean-Raymond, & Hallerstede, Stefan. 2007. Refinement, decomposition, and instantiation of discrete models: Application to event-B. *Fundamenta Informaticae*, **77**(1-2), 1–28.

[6] Abrial, Jean-Raymond, Butler, Michael, Hallerstede, Stefan, Hoang, Thai Son, Mehta, Farhad, & Voisin, Laurent. 2010. Rodin: An open toolset for modelling and reasoning in event-B. *International Journal on Software Tools for Technology Transfer*, **12**(6), 447–466.

[7] Arendt, Thorsten, Biermann, Enrico, Jurack, Stefan, Krause, Christian, & Taentzer, Gabriele. 2010. Henshin: Advanced concepts and tools for in-place EMF model transformations. *Pages 121–135 of: 13th International Conference on Model Driven Engineering Languages and Systems: Part I*. MODELS'10. Springer-Verlag.

[8] Azab, Karl, Habel, Annegret, Pennemann, Karl-Heinz, & Zuckschwerdt, Christian. 2006. ENFORCe: A system for ensuring formal correctness of high-level programs. *Electronic Communications of the EASST*, **1**.

[9] Back, Ralph-Johan, & Sere, Kaisa. 1989. Stepwise refinement of action systems. *Pages 115–138 of:* van de Snepscheut, Jan L. A. (ed), *International Conference on Mathematics of Program Construction*. Springer-Verlag.

[10] Baldan, Paolo, Corradini, Andrea, & König, Barbara. 2008. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, **206**(7), 869–907.

[11] Baresi, Luciano, & Spoletini, Paola. 2006. On the use of Alloy to analyze graph transformation systems. *Pages 306–320 of: Third International Conference on Graph Transformations*. ICGT'06. Springer-Verlag.

[12] Bensalem, Saddek, Ganesh, Vijay, Lakhnech, Yassine, noz, César Mu Owre, Sam, Rueß, Harald, Rushby, John, Rusu, Vlad, Saïdi, Hassen, Shankar, N., Singerman, Eli, & Tiwari, Ashish. 2000. An overview of SAL. *Pages 187–196 of:* Holloway, C. Michael (ed), *Fifth NASA Langley Formal Methods Workshop*.

[13] Bertot, Yves, & Castéran, Pierre. 2004. *Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions*. Texts in Theoretical Computer Science. Springer Verlag.

[14] Boolos, George S. 1975. On second-order logic. *The Journal of Philosophy*, **72**(16), 509–527.

[15] Born, Kristopher, Arendt, Thorsten, Heß, Florian, & Taentzer, Gabriele. 2015. Analyzing conflicts and dependencies of rule-based transformations in Henshin. *Pages 165–168 of:* Egyed, Alexander, & Schaefer, Ina (eds), *18th International Conference on Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science, vol. 9033. Springer-Verlag.

[16] Corradini, Andrea, Montanari, Ugo, Rossi, Francesca, Ehrig, Hartmut, Heckel, Reiko, & Löwe, Michael. 1997. Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. *In:* [71].

[17] Corradini, Andrea, Duval, Dominique, Echahed, Rachid, Prost, Frédéric, & Ribeiro, Leila. 2015. AGREE - algebraic graph rewriting with controlled embedding. *Pages 35–51 of: 8th International Conference on Graph Transformation*. Lecture Notes in Computer Science, vol. 9151. Springer-Verlag.

[18] Courcelle, Bruno. 1997. The expression of graph properties and graph transformations in monadic second-order logic. *In:* [71].

[19] Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., & Varr'o, D'aniel. 2002. VIATRA - visual automated transformations for formal verification and validation of UML models. *Pages 267–270 of: 17th IEEE International Conference on Automated Software Engineering*.

[20] da Costa, Simone André. 2010. *Relational approach of graph grammars*. Ph.D. thesis, UFRGS, Brazil.

[21] da Costa, Simone André, & Ribeiro, Leila. 2009. Formal verification of graph grammars using mathematical induction. *Electronic Notes in Theoretical Computer Science*, **240**, 43–60.

[22] da Costa, Simone André, & Ribeiro, Leila. 2012. Verification of graph grammars using a logical approach. *Science of Computer Programming*, **77**(4), 480 – 504.

[23] da Costa Cavalheiro, Simone André, Foss, Luciana, & Ribeiro, Leila. 2012. Specification patterns for properties over reachable states of graph grammars. *Pages 83–98 of:* Gheyi, Rohit, & Naumann, David A. (eds), *15th Brazilian Symposium on Formal Methods: Foundations and applications*. Lecture Notes in Computer Science, vol. 7498. Springer-Verlag.

[24] DEPLOY. 2008. *Event-B and the Rodin platform*. http://www.event-b.org/. Rodin Development is supported by European Union ICT Projects DEPLOY (2008 to 2012) and RODIN (2004 to 2007).

[25] Dotti, Fernando Luís, Foss, Luciana, Ribeiro, Leila, & dos Santos, Osmar Marchi. 2003. Verification of distributed object-based systems. *Pages 261–275 of: 6th International Conference on Formal Methods for Open Object-Based Distributed Systems*.

[26] Duval, Dominique, Echahed, Rachid, Prost, Frédéric, & Ribeiro, Leila. 2014. Transformation of attributed structures with cloning. *Pages 310–324 of: 17th International Conference on Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science, vol. 8411. Springer-Verlag.

[27] Ehrig, H., Engels, G., Kreowski, H.-J., & Rozenberg, G. (eds). 1999. *Handbook of graph grammars and computing by graph transformation, volume 2: applications, languages, and tools*. World Scientific Publishing Co., Inc.

[28] Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. 2006a. *Fundamentals of algebraic graph transformation*. Springer-Verlag.

[29] Ehrig, Hartmut. 1979. Introduction to the algebraic theory of graph grammars (A survey). *Pages 1–69 of:* Claus, Volker, Ehrig, Hartmut, & Rozenberg, Grzegorz (eds), *International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*. Lecture Notes in Computer Science, vol. 73. Springer-Verlag.

[30] Ehrig, Hartmut, Heckel, Reiko, Korff, Martin, Löwe, Michael, Ribeiro, Leila, Wagner, Annika, & Corradini, Andrea. 1997. Algebraic approaches to graph transformation - part II: Single pushout approach and comparison with double pushout approach. *In:* [71].

[31] Ehrig, Hartmut, Ehrig, Karsten, Prange, Ulrike, & Taentzer, Gabriele. 2006b. Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. *Fundamta informaticae*, **74**(1), 31–61.

[32] Ermel, C., Rudolf, M., & Taentzer, G. 1999. The AGG approach: Language and environment. *In:* [27].

[33] Ermel, Claudia, Biermann, Enrico, Schmidt, Johann, & Warning, Angeline. 2010. Visual modeling of controlled EMF model transformation using HENSHIN. *Electronic Communications of the EASST*, **32**.

[34] Galvão, Ismênia, Zambon, Eduardo, Rensink, Arend, Wevers, Lesley, & Aksit, Mehmet. 2011. Knowledge-based graph exploration analysis. *Pages 105–120 of:* Schürr, Andy, Varró, Dániel, & Varró, Gergely (eds), *4th International Symposium on Applications of Graph Transformations with Industrial Relevance*. Lecture Notes in Computer Science, vol. 7233. Springer-Verlag.

[35] Garavel, Hubert, Lang, Frédéric, Mateescu, Radu, & Serwe, Wendelin. 2013. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, **15**(2), 89–107.

[36] Ghamarian, Amir Hossein, de Mol, Maarten, Rensink, Arend, Zambon, Eduardo, & Zimakova, Maria. 2012. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, **14**(1), 15–40.

[37] Girard, Jean-Yves. 1987. Linear logic. *Theoretical computer science*, **50**(1), 1 – 101.

[38] Golas, Ulrike. 2012. A general attribution concept for models in $\mathcal{M}$-adhesive transformation systems. *Pages 187–202 of: 6th International Conference on Graph Transformations*. Lecture Notes in Computer Science, vol. 7562. Springer-Verlag.

[39] Groote, J.F., Keiren, J., Mathijssen, A., Ploeger, B., Stappers, F., Tankink, C., Usenko, Y., Weerdenburg, M. van, Wesselink, W., Willemse, T., & Wulp, J. van der. 2008. The mCRL2 toolset. *In: International Workshop on Advanced Software Development Tools and Techniques*.

[40] Habel, Annegret, & Pennemann, Karl-Heinz. 2009. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, **19**(4), 245–296.

[41] Habel, Annegret, & Plump, Detlef. 2002. Relabelling in graph transformation. *Pages 135–147 of: First International Conference on Graph Transformation*. Lecture Notes in Computer Science, vol. 2505. Springer-Verlag.

[42] Habel, Annegret, Heckel, Reiko, & Taentzer, Gabriele. 1996. Graph grammars with negative application conditions. *Fundamenta Informaticae*, **26**(3-4), 287–313.

[43] Habel, Annegret, Pennemann, Karl-Heinz, & Rensink, Arend. 2006. Weakest preconditions for high-level programs. *Pages 445–460 of:* Corradini, Andrea, Ehrig, Hartmut, Montanari, Ugo, Ribeiro, Leila, & Rozenberg, Grzegorz (eds), *Third International Conference on Graph Transformations*. Lecture Notes in Computer Science, vol. 4178. Springer-Verlag.

[44] Hermann, Frank, Kastenberg, Harmen, & Modica, Tony. 2006. Towards translating graph transformation approaches by model transformations. *Electronic Communications of the EASST*, **4**.

[45] Hermann, Frank, Corradini, Andrea, & Ehrig, Hartmut. 2014. Analysis of permutation equivalence in $\mathcal{M}$-adhesive transformation systems with negative application conditions. *Mathematical Structures in Computer Science*, **24**(4).

[46] Hristakiev, Ivaylo, & Plump, Detlef. 2014. A unification algorithm for GP 2. *Electronic Communications of the EASST*, **71**.

[47] Jr., Luiz Carlos Lemor, da Costa Cavalheiro, Simone André, & Foss, Luciana. 2015. Proof tactics for theorem proving graph grammars through Rodin. *Revista de Informática Teórica e Aplicada*, **22**(1), 190–241.

[48] Junior, Luiz Carlos Lemos, da Costa Cavalheiro, Simone André, & Foss, Luciana. 2013. Theorem proving graph grammars: Strategies for discharging proof obligations. *Pages 147–162 of:* Iyoda, Juliano, & de Moura, Leonardo Mendonça (eds), *16th Brazilian Symposium on Formal Methods: Foundations and applications*. Lecture Notes in Computer Science, vol. 8195. Springer-Verlag.

[49] Kastenberg, Harmen. 2006. Towards attributed graphs in groove: Work in progress. *Electronic Notes in Theoretical Computer Science*, **154**(2), 47 – 54.

[50] König, Barbara, & Kozioura, Vitali. 2008. Towards the verification of attributed graph transformation systems. *Pages 305–320 of: 4th International Conference on Graph Transformations*. Lecture Notes in Computer Science, vol. 5214. Springer-Verlag.

[51] Korff, Martin. 1995. *Generalized graph structures with application to concurrent object-oriented systems*. Ph.D. thesis, TU Berlin.

[52] Kwiatkowska, Marta, Norman, Gethin, & Parker, David. 2011. Prism 4.0: Verification of probabilistic real-time systems. *Pages 585–591 of: 23rd International Conference on Computer Aided Verification*. CAV'11. Springer-Verlag.

[53] Löwe, Michael. 1993. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, **109**(1&2), 181–224.

[54] Löwe, Michael, Korff, Martin, & Wagner, Annika. 1993. An algebraic framework for the transformation of attributed graphs. *Pages 185–199*

of: Term Graph Rewriting: theory and practice. John Wiley and Sons Ltda.
[55] Manna, Zohar, & Pnueli, Amir. 1992. *The temporal logic of reactive and concurrent systems - specification*. Springer-Verlag.
[56] Manning, Greg, & Plump, Detlef. 2008. The GP programming system. *Electronic Communications of the EASST*, **10**.
[57] Nipkow, Tobias, Paulson, Lawrence C., & Wenzel, Markus. 2002. *Isabelle/HOL: A proof assistant for higher-order logic*. Lecture Notes in Computer Science, vol. 2283. Springer-Verlag.
[58] Oliveira, Marcel, Cavalcanti, Ana, & Woodcock, Jim. 2006. Unifying theories in proofpower-Z. *Pages 123–140 of:* Dunne, Steve, & Stoddart, Bill (eds), *First International Symposium on Unifying Theories of Programming*. Lecture Notes in Computer Science, vol. 4010. Springer-Verlag.
[59] Orejas, Fernando. 2011. Symbolic graphs for attributed graph constraints. *Journal of Symbolic Computation*, **46**(3), 294–315.
[60] Orejas, Fernando, & Lambers, Leen. 2010. Symbolic attributed graphs for attributed graph transformation. *Electronic Communications of the EASST*, **30**.
[61] Pennemann, Karl-Heinz. 2008a. An algorithm for approximating the satisfiability problem of high-level conditions. *Electronic Notes in Theoretical Computer Science*, **213**(1), 75–94.
[62] Pennemann, Karl-Heinz. 2008b. Resolution-like theorem proving for high-level conditions. *Pages 289–304 of:* Ehrig, Hartmut, Heckel, Reiko, Rozenberg, Grzegorz, & Taentzer, Gabriele (eds), *International Conference on Graph Transformations*. Lecture Notes in Computer Science, vol. 5214. Springer-Verlag.
[63] Percebois, Christian, Strecker, Martin, & Tran, Hanh Nhi. 2013. Rule-level verification of graph transformations for invariants based on edges' transitive closure. *Pages 106–121 of:* Hierons, Robert M., Merayo, Mercedes G., & Bravetti, Mario (eds), *11th International Conference on Software Engineering and Formal Methods*. Lecture Notes in Computer Science, vol. 8137. Springer-Verlag.
[64] Peuser, Christoph, & Habel, Annegret. 2015. Attribution of graphs by composition of m, n-adhesive categories. *Pages 66–81 of: 6th international workshop on graph computation models*, vol. 1403. CEUR-WS.org.
[65] Plump, Detlef. 2011. The design of GP 2. *Pages 1–16 of:* Escobar, Santiago (ed), *10th International Workshop on Reduction Strategies in Rewriting and Programming*. EPTCS, vol. 82.
[66] Plump, Detlef, & Bak, Christopher. 2012. Rooted graph programs. *Electronic Communications of the EASST*, **54**.
[67] Poskitt, Christopher M., & Plump, Detlef. 2012. Hoare-style verification of graph programs. *Fundamenta Informaticae*, **118**(1-2), 135–175.
[68] Poskitt, Christopher M., & Plump, Detlef. 2013. Verifying total correctness of graph programs. *In: Revised selected papers, graph computation models*. Electronic Communications of the EASST, vol. 61.
[69] Rensink, Arend. 2004. The GROOVE simulator: A tool for state space generation. *Pages 479–485 of:* Pfalz, J., Nagl, M., & Böhlen, B. (eds), *Applications of graph transformations with industrial relevance*. Lecture Notes in Computer Science, vol. 3062. Springer-Verlag.
[70] Ribeiro, Leila, Dotti, Fernando Luís, da Costa, Simone André, & Dillenburg, Fabiane Cristine. 2010. Towards theorem proving graph grammars using event-B. *Electronic Communications of the EASST*, **30**.
[71] Rozenberg, Grzegorz (ed). 1997. *Handbook of graph grammars and computing by graph transformations, volume 1: Foundations*. World Scientific.
[72] Strecker, Martin. 2008. Modeling and verifying graph transformations in proof assistants. *Electronic Notes in Theoretical Computer Science*, **203**(1), 135–148.
[73] Strecker, Martin. 2012. Locality in reasoning about graph transformations. *Pages 169–181 of: 4th International Conference on Applications of Graph Transformations with Industrial Relevance*. AGTIVE'11. Springer-Verlag.
[74] Strecker, Martin. 2014. Interactive and automated proofs for graph transformations. *Mathematical Structures in Computer Science (MSCS)*, 31 pages.
[75] Tanenbaum, Andrew S., & Wetherall, David J. 2011. *Computer networks*. 5th edn. Prentice Hall.
[76] Tran, Hanh Nhi, & Percebois, Christian. 2012. Towards a rule-level verification framework for property-preserving graph transformations. *Pages 946–953 of:* Antoniol, Giuliano, Bertolino, Antonia, & Labiche, Yvan (eds), *Fifth International IEEE Conference on Software Testing, Verification and Validation*. IEEE Computer Society.
[77] Väänänen, Jouko. 2012. *Epistemology versus ontology: Essays on the philosophy and foundations of mathematics in honour of Per Martin-Löf*. Springer-Verlag. Chap. Second Order Logic, Set Theory and Foundations of Mathematics, pages 371–380.
[78] Varró, Dániel, Varró, Gergely, & Pataricza, András. 2002. Designing the automatic transformation of visual languages. *Science of Computer Programming*, **44**(2), 205–227.
[79] Warmer, Jos, & Kleppe, Anneke. 1999. *The object constraint language: Precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc.

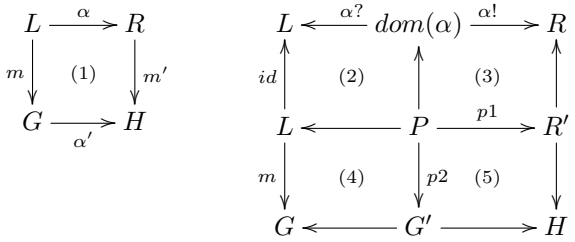
Appendix A. Pushouts Constructions

Definition 36. *Given a pair of morphisms $f : A \to B$ and $g : A \to C$ in a category* $\mathbf{Cat}$ *the pushout of f and g is (up to isomorphism) a triple $(D, f' : C \to D, g' : B \to D)$ such that for any other triple $(X, f^X : C \to X, g^X : B \to X)$ there is a unique morphism $u : D \to X$ with $u \circ f' = f^X$ and $u \circ g' = g^X$.*

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & (PO) & \downarrow{\scriptstyle g'} \\ C & \xrightarrow[f']{} & D \end{array}$$

In the following definition we give the constructions of pushout in some concrete categories used in the paper. We consider only a particular kind of pushout where one of the morphisms in a monomorphism and the other a total morphism.

Definition 37. *Consider diagram (1), where α is a monomorphism and m is total. The concrete construction of the object H as a pushout in some categories is given by*[12]*:*



Set *: In the category of sets and total functions, if α is an injective function, H can be constructed as follows [29, 16, 51]*

$$H = G \uplus (R \backslash rng(\alpha))$$

SetP *: In the category of sets and partial functions, if α is injective and m is total, the construction of H can be done by following the 3 steps below (this is a special case of the construction proposed in [54]):*

(i) *Construction of the gluing object: this step construct the object P that contains all elements of $dom(\alpha)$ that will actually be in the resulting set. The elements that are in $dom(\alpha)$ and are not in P are the ones for which a definition analogous to the identification condition in Def. 13 for sets is true.*

P is the greatest subset of L satisfying $P \subseteq dom(\alpha)$ and for all $x \in P, y \in L$, if $m(x) = m(y)$ then $y \in P$.

(ii) *Construction of the definedness areas of α' and m': R' resp. G' are constructed by removing from R resp. G all elements that should not be in the result, that is, elements in the image of $\alpha!$ resp. m that do not have pre-images in P. Note that $L \backslash P$ denotes all elements that can not have images in the resulting set H.*

$$R' = R \backslash (\alpha[(L \backslash P)]) \text{ and } G' = G \backslash (m[(L \backslash P)])$$

(iii) *Gluing: Now H is obtained by gluing G' and R' along P. This step ((5) in the diagram above) is a pushout in the category* **Set***, where $p1$ and $p2$ are $\alpha!$ and m, respectively, restricted to the elements of P.*

$$H = G' \uplus (R' \backslash rng(p))$$

Graph: *In the category of graphs and total graph homomorphisms, pushouts are constructed componentwise in* **Set***, where the source and target functions are obtained as the universal morphism induced by the pushout of edges [29, 16].*

GraphP: *In the category of graphs and partial graph homomorphisms, the pushout object D is the largest subgraph of $D' = (Vert^{D'}, Edge^{D'}, source^{D'}, target^{D'})$ where the sets of vertices and edges are constructed componentwise in* **SetP** *[51, 30] to be equivalent to the definition in [54], which follows essentially the steps presented above for the category* **SetP***: step (i) results in the largest graph that contains all items that should be in the result; step (ii) removes dangling edges from R (R' in diagram (3) is the largest subgraph of R that contains images of all elements in P) and from G (G' is the largest subgraph of G that contains images of all elements of P); finally, step (iii) performs the gluing of G' and R' as a pushout in* **Graph***. Deletion due to identification condition is performed in step (i) whereas deletion of dangling edges in step (ii). In the construction proposed in [51], deletion due to identification condition is performed componentwise by the pushouts in* **SetP***, and deletion of dangling edges by finding the largest graph that contains all the remaining elements (formally, this was defined by a free construction from a category of "quasi graphs" to the category* **GraphP***).*

[12]The object $dom(\alpha)$ is the domain of definition of the partial morphism α, morphisms $\alpha?$ and $\alpha!$ are the corresponding domain inclusion and domain restriction, respectively.

$\mathbf{TGraph}(\mathbf{T})$: *The category of typed graphs and total graph homomorphisms can be seen as a comma category $(\mathbf{Graph} \downarrow \mathbf{T})$, where $\mathbf{T}$ is the category having just graph T as element. Pushouts in comma categories are constructed componentwise.*

$\mathbf{TGraphP}(\mathbf{T})$: *The category of typed graphs and partial graph homomorphisms is very similar to a comma category, where commutativity is substituted by a weak commutativity requirement (commutativity is required in the domain of definition only). [51] have shown that pushouts in this category can be constructed componentwise in $\mathbf{GraphP}$ and $\mathbf{T}$.*

Proposition 38. *Given the setting of Theorem 16, the pushout of α_V and m_V in $\mathbf{SetP}$ yields, up to isomorphism, $VertH = (VertG \backslash Del_V^{\alpha,m}) \uplus New_V^{\alpha,m}$.*

PROOF. *According to the construction of pushouts in $\mathbf{SetP}$ (Def. 37), we have:*

$$VertP \text{ has all vertices that are preserved by the rule not identified with deleted vertices by the match.} \tag{A.1}$$

$$VertG' = VertG \backslash (m^V[(VertL \backslash VertP)] \tag{A.2}$$

$$VertR' = VertR \backslash (\alpha_V[(VertL \backslash VertP)]) \tag{A.3}$$

$$VertH = VertG' \uplus (VertR' \backslash rng(p1^V)) \tag{A.4}$$

Using Defs. 12 and 6 we deduce that $Del_V^{\alpha,m} = m_V[RuleDel_V^{\alpha}] = m_V[VertL \setminus dom(\alpha_V)]$. Note that by definition of $VertP$, the only possibility of a vertex v to be in $dom(\alpha_V)$ and not in $VertP$ is when the image of v under m is the same of another vertex that is not in $dom(\alpha_V)$. Therefore we conclude that $m_V[VertL \backslash VertP] = m_V[VertL \setminus dom(\alpha_V)] = m_V[Del_V^{\alpha,m}]$, and thus

$$VertG' = VertG \backslash (m^V[Del_V^{\alpha,m}]) \tag{A.5}$$

Using Defs. 12 and 6, $New_V^{\alpha,m} = RuleCreate_V^{\alpha} = VertR \setminus rng(\alpha_V)$. Obviously, since α_V is a function, $\alpha_V[(VertL \backslash VertP)] \subseteq rng(\alpha_V)$ and therefore $VertR \setminus rng(\alpha_V) \subseteq VertR \backslash (\alpha_V[(VertL \backslash VertP)])$ and therefore, using A.3 we conclude that $VertR \setminus rng(\alpha_V) \subseteq VertR'$.

$$VertR' = VertR \setminus rng(\alpha_V) = New_V^{\alpha,m} \tag{A.6}$$

But by construction (3) in the diagram of Def. 37, $VertR' \subseteq VertR$, and $p1_V$ is a restriction of α_V!, and therefore the image of $p1_V$ must be contained in the image of α_V leading to the fact that $(VertR \setminus rng(\alpha_V)) \backslash rng(p1_V) = VertR \setminus rng(\alpha_V)$ and

$$VertR' \backslash rng(p1_V) = VertR \setminus rng(\alpha_V) = New_V^{\alpha,m} \tag{A.7}$$

Using A.4, A.5 and A.7 we finally conclude that

$$VertH = VertG' \uplus (VertR' \backslash rng(p1^V)) = VertG \backslash (m^V[Del_V^{\alpha,m}]) \uplus New_V^{\alpha,m} \tag{A.8}$$

Proposition 39. *Given the setting of Theorem 16, the set of edges of the pushout object of α and m in $\mathbf{TGraphP}(\mathbf{T})$ is, up to isomorphism, $EdgeH = (EdgeG \backslash (Del_E^{\alpha,m} \cup Dangling^{\alpha,m})) \uplus New_E^{\alpha,m}$.*

PROOF. *According to Def. 37, pushouts in $\mathbf{TGraphP}(\mathbf{T})$ is done componentwise, and the set of edges of the resulting graph is obtained by a pushout in $\mathbf{GraphP}$, which, in turn, is obtained componentwise for vertices and edges in $\mathbf{SetP}$, followed by a free construction that basically deletes the dangling edges such that H is the largest graph that contains all vertices and edges obtained in the corresponding pushouts in $\mathbf{SetP}$. Let $EdgeH'$ be the set of edges obtained by the pushout of m^E and α^E in $\mathbf{SetP}$. Analogoulsy to the proof of Prop. 38,*

$$EdgeH' = EdgeG \backslash (m^E[Del_E^{\alpha,m}]) \uplus (EdgeR' \backslash rng(p1^E)) \tag{A.9}$$

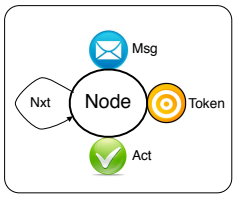
Note that here, differently from the definition of $New_V^{\alpha,m}$, $New_E^{\alpha,m}$ may contain less than the edges of $RuleCreate_E^{\alpha}$ because only vertices whose source and target are preserved or created can be in $New_E^{\alpha,m}$, that is $New_E^{\alpha,m} \subseteq$

$(EdgeR'\backslash rng(p1^E))$. *Now, to construct* $EdgeH$ *we have to remove from* $EdgeH'$ *exactly all edges that prevent* H' *from being a graph, these are exactly all edges that are dangling (whose source or target does not belong to* $VertH$*). These dangling edges may be in* G*, or in* R*. By Def. 12,* $Dangling^{\alpha,m}$ *contains exactly the edges from* G *that are dangling, and the set* $New_E^{\alpha,m}$ *contains only the edges of* R *that are not dangling. Thus, we conclude that the set* $EdgeH$ *below contains the largest set of edges that turns* H *a graph.*

$$EdgeH = (EdgeG\backslash(Del_E^{\alpha,m} \cup Dangling^{\alpha,m})) \uplus New_E^{\alpha,m} \tag{A.10}$$

Appendix B. Summary of the Translation

In this appendix we present a summary of the translation of the Token Ring graph grammar to the event-B model. Figures B.11 to B.13 illustrate the steps translating the structural part of a GG (a GG without attributes) to event-B, whereas Figures B.14 to B.17 show the steps of the translation of the attribute layer.

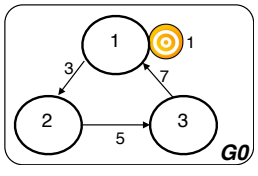


(a) Type graph

```
sets
     VertT // (Type Graph T) Vertices
     EdgeT // (Type Graph T) Types of Edges
constants Node // Type of Nodes
          Nxt  Tok  Msg  Act // Types of Edges
          sourceT // // function sourceT -- returns the source of an Edge of T
          targetT // // function targetT -- returns the target of an Edge of T
axioms
  @axm_VertT partition(VertT, {Node}) // (Type Graph T)
  @axm_EdgeT partition(EdgeT, {Nxt}, {Tok}, {Msg}, {Act}) // EdgeT = {Nxt, Tok, Msg, Stb, Act}
  @axm_srcTtype sourceT ∈ EdgeT → VertT
  @axn_srcTdef partition(sourceT,{Nxt↦Node},{Tok↦Node},{Msg↦Node},{Act↦Node})
  @axm_tgtTtype targetT ∈ EdgeT → VertT
  @axn_tgtTdef partition(targetT,{Nxt↦Node},{Tok↦Node},{Msg↦Node},{Act↦Node})
```

(b) Type graph in event-B

Figure B.11: Step T1 (part of AT1) - Translation of the type graph (Def. 17)

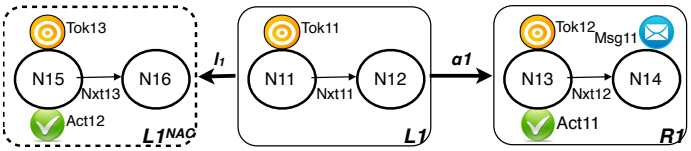


(a) Initial graph G_0

```
variables
     VertG   // (Graph) Vertices
     EdgeG   // (Graph) Edges
     sourceG // (Graph) function sourceG
     targetG // (Graph) function targetG
     tG_V    // (Graph) Typing vertices
     tG_E    // (Graph) Typing edges
invariants
  @inv_vertG VertG ∈ ℙ(ℕ) // (Graph) Vertices are natural numbers.
  @inv_edgeG EdgeG ∈  ℙ(ℕ) // (Graph) Edges are natural numbers.
  @inv_srcGtype sourceG ∈ EdgeG → VertG // (Graph) function sourceG
  @inv_tgtGtype targetG ∈ EdgeG → VertG // (Graph) function targetG
  @inv_tG_V tG_V ∈ VertG → VertT // (Graph) function tG_V
  @inv_tG_E tG_E ∈ EdgeG → EdgeT // (Graph) function tG_E
events
   event INITIALISATION
     then
       @act_vertG VertG ≔ {1, 2, 3} // // (G = G0) Vertices
       @act_edgeG EdgeG ≔ {1,3,5,7} // // (G = G0) Edges
       @act_srcG sourceG ≔  {1↦1, 3↦1, 5↦2, 7↦3} // // (G = G0) function sourceG
       @act_tgtG targetG ≔ {1↦1, 3↦2, 5↦3, 7↦1} // // (G = G0) function targetG
       @act_tG_V tG_V ≔ {1↦Node, 2↦Node, 3↦Node} // // (G = G0) Typing vertices
       @act_tG_E tG_E ≔ {1↦Tok,  3↦Nxt, 5↦Nxt, 7↦Nxt} // // (G = G0) Typing edges
   end
```

(b) State graph and initial graph in event-B

Figure B.12: Step T2 (part of AT1) - Translation of state graph and initial graph (Def. 18)



(a) Rule r_1

```
sets
    VertL1 // (Rule 1) Left Graph L1 -- Vertices
    EdgeL1 // (Rule 1) Left Graph L1 -- Edges
constants
    N11 N12 // (Rule 1) Vertex names
    Tok11 Nxt11 // (Rule 1) Edge names
    sourceL1 // (Rule 1) function sourceL1 -- returns the source of an Edge
    targetL1 // (Rule 1) function targetL1 -- returns the target of an Edge
    tL1_V // (Rule 1) Typing Vertices
    tL1_E // (Rule 1) Typing Edges
axioms
  @axm_VertL1 partition(VertL1, {N11}, {N12}) // (Rule 1) Left Graph L1 -- Vertices
  @axm_EdgeL1 partition(EdgeL1, {Tok11},{Nxt11}) // (Rule 1) Left Graph L1 -- Edges
  @axm_srcL1type sourceL1 ∈ EdgeL1 → VertL1 // (Rule 1) function sourceL1
  @axn_srcL1def partition(sourceL1,{Tok11↦N11},{Nxt11↦N11})
  @axm_tgtL1type targetL1 ∈ EdgeL1 → VertL1 // (Rule 1) function targetL1
  @axn_tgtL1def partition(targetL1,{Tok11↦N11},{Nxt11↦N12})
  @axm_tL1_V tL1_V ∈  VertL1↦VertT // (Rule 1) Typing Vertices
  @axm_tL1_V_def partition(tL1_V , {N11↦Node}, {N12↦Node})
  @axm_tL1_E tL1_E ∈  EdgeL1↦EdgeT // (Rule 1) Typing Edges
  @axm_tL1_E_def partition(tL1_E , {Tok11↦Tok}, {Nxt11↦Nxt})
```

(b) LHS of r_1 rule pattern in event-B

```
event r1
  any mV // mV component of a match
      mE // mE component of a match
      newMsg11 // new fresh name for an edge
      newAct11 // new fresh name for an edge
  where
    @grd_mV mV ∈ VertL1 → VertG // mV is total
    @grd_mE mE ∈ EdgeL1 → EdgeG // mE is total and injective
    @grd_new_newMsg11 newMsg11 ∈  N\EdgeG // newEmsg is a fresh name
    @grd_new_newAct11 newAct11 ∈  N\EdgeG // newEact is a fresh name
    @grd_diffnewMsg11newAct11 newMsg11 ≠ newAct11 // new edges are different
    @grd_tV ∀v· v ∈ VertL1 ⇒ tL1_V(v) = tG_V(mV(v)) // vertex type compatibility
    @grd_tE ∀e· e ∈ EdgeL1 ⇒ tL1_E(e) = tG_E(mE(e)) // edge type compatibility
    @grd_srctgt ∀e· e ∈ EdgeL1 ⇒ mV(sourceL1(e))=sourceG(mE(e)) ∧ mV(targetL1(e))=targetG(mE(e))
                // source/target compatibility
    @grd_NAC1 ¬(∃forbAct12·
                {forbAct12} ⊆ EdgeG \ mE[EdgeL1] ∧  tG_E(forbAct12) = Act ∧
                              sourceG(forbAct12) = mV(N11) ∧ targetG(forbAct12) = mV(N11))
            // there is no Act edge on the image of vertex N11 (the vertex that has the Tok edge)
  then
    @act_E EdgeG ≔  EdgeG ∪ {newAct11,newMsg11}
    @act_src sourceG ≔ sourceG ∪ {newAct11 ↦ mV(N11),newMsg11 ↦ mV(N12)}
    @act_tgt targetG ≔ targetG ∪ {newAct11 ↦ mV(N11),newMsg11 ↦ mV(N12)}
    @act_tE tG_E ≔   tG_E ∪  {newAct11↦Act,newMsg11↦Msg}
end
```

(c) Rule application of r_1 in event-B

Figure B.13: Step T3 (part of AT1) - Translation of each rule (Defs. 19 and 21)

sorts Bool, Nat, Stack

opns

true : $\rightarrow$ Bool
false: $\rightarrow$ Bool
0 : $\rightarrow$ Nat
succ : Nat $\rightarrow$ Nat
$\neq$: Nat $\times$ Nat $\rightarrow$ Bool
empty : $\rightarrow$ Stack
push : Nat $\times$ Stack $\rightarrow$ Stack
pop : Stack $\rightarrow$ Stack
top : Stack $\rightarrow$ Nat
notTwice : Stack $\rightarrow$ Bool

eqns

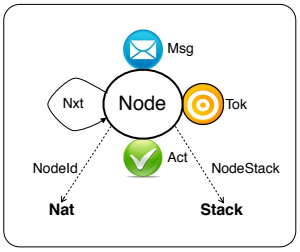
$\forall n \in$ Nat, $s \in$ Stack:
$\neq$(0,0)=false
$\neq$(0,succ(n))=true
$\neq$(succ(n),0)=true
$\neq$(succ($n1$),succ($n2$))=$\neq$($n1$,$n2$)
pop(push(n, s))= s
pop(empty)=empty
top(push(n,s))= n
top(empty)=0
notTwice(empty)=true
notTwice(push(n, empty))= true
($top(s) \neq n \vee n = 0$) $\Rightarrow$
notTwice(push(n,s))= notTwice(s)
notTwice(push(n,push(n,s)))= false

(a) Algebraic specification

```
sets
    Stack
constants
    empty
    pop
    push
    top
    notTwice
axioms
  @axm_empty empty ∈ Stack
  @axm_pop pop  ∈ Stack → Stack
  @axm_push push  ∈ N × Stack → Stack
  @axm_top top   ∈ Stack → N
  @axm_notTwice notTwice ∈ Stack →  BOOL
  @axm_eq1 ∀s,n· s ∈ Stack  ∧ n∈ N  ⇒ pop(push(n↦s))= s
  @axm_eq2 pop(empty)=empty
  @axm_eq3 ∀s,n· s ∈ Stack ∧  n∈ N  ⇒ top(push(n↦s))= n
  @axm_eq4 top(empty)=0
  @axm_eq5 notTwice(empty)=TRUE
  @axm_eq6 ∀n· n∈ N  ⇒ notTwice(push(n↦empty))= TRUE
  @axm_eq7 ∀s,n· s ∈ Stack  ∧ n∈ N  ∧ (top(s)≠n ∨ n=0) ⇒ notTwice(push(n↦s))= notTwice(s)
  @axm_eq8 ∀s,n· s ∈ Stack  ∧ n∈ N  ⇒ notTwice(push(n↦push(n↦s)))= FALSE
```

(b) Data types specification in event-B

Figure B.14: Step AT2 - Translation of the data types (Def. 31)

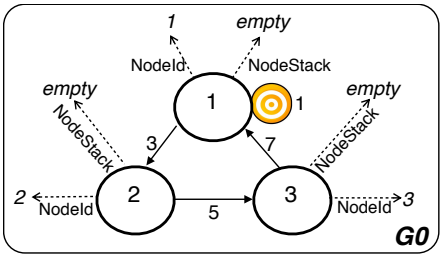


(a) Attributed type graph

```
sets
    AttrT
    DataType
constants
    attrvT
    valT
    BoolSort
    NatSort
    StackSort
    NodeStack
    NodeId
axioms
  @axm_AttrT partition(AttrT,{NodeStack},{NodeId})
  @axm_attrTDiffNodeStackNodeId NodeStack ≠ NodeId
  @axm_data partition(DataType,{NatSort},{StackSort},{BoolSort})
  @axm_dataDiffNatSortStackSort NatSort≠StackSort
  @axm_dataDiffNatSortBoolSort NatSort≠BoolSort
  @axm_dataDiffBoolSortStackSort BoolSort≠StackSort
  @axm_attrvT attrvT ∈ AttrT → VertT
  @axm_attrvTdef partition(attrvT,{NodeStack↦Node},{NodeId↦Node})
  @axm_valT valT ∈  AttrT → DataType
  @axm_valTdef partition(valT,{NodeStack↦StackSort},{NodeId↦NatSort})
```

(b) Attributes of the type graph in event-B

Figure B.15: Step AT3 - Translation of the type graph attributes (Def. 32)



(a) Attributed initial graph

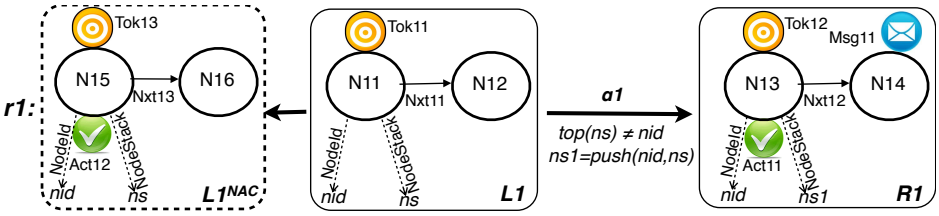
```
variables
  AttrG // attribute vertices
  attrvG// assigns graph vertices to attribute vertices
  tG_A // assigns a type to each attribute vertex
  valG_NodeStack // assigns values to attribute vertices
  valG_NodeId // assigns values to attribute vertices
invariants
  @inv_AttrG AttrG ∈ ℙ(N) // (Attribute) Vertices are natural numbers.
  @inv_elemG attrvG ∈ AttrG → VertG
  @inv_tGA tG_A ∈   AttrG → AttrT
  @inv_valGNodeStack valG_NodeStack ∈ AttrG ⇸ Stack
  @inv_valGNodeId valG_NodeId ∈ AttrG ⇸ N
  @inv_DiffNodeStackNodeId dom(valG_NodeStack)ndom(valG_NodeId)=∅
  @inv_typeNodeStack ∀a· a∈ AttrG ∧  a ∈ dom(tG_A⩥{NodeStack}) ⇒ a ∈ dom(valG_NodeStack)
              // assures type compatibility of stack attribute vertices
  @inv_typeNodeId ∀a· a∈ AttrG ∧  a ∈ dom(tG_A⩥{NodeId}) ⇒ a ∈ dom(valG_NodeId)
              // assures type compatibility of nat attribute vertices
```

(b) State attributes in event-B

```
event INITIALISATION extends INITIALISATION
 then
     @act_AttrG AttrG ≔  {11, 12, 21, 22, 31, 32} // attribute vertices
     @act_attrvG attrvG ≔  {11↦1,12↦1,21↦2,22↦2,31↦3,32↦3} // assigns graph vertices to attribute vertices
     @act_tGA tG_A ≔ {11↦NodeStack,12↦NodeId,21↦NodeStack,22↦NodeId,31↦NodeStack,32↦NodeId}
                  // assigns a type to each attribute vertex
     @act_valGNodeStack valG_NodeStack ≔ {11↦empty,21↦empty,31↦empty} // assigns values to attribute vertices
     @act_valGNodeId valG_NodeId ≔ {12↦1,22↦2,32↦3} // assigns values to attribute vertices
 end
```

(c) Initial graph attributes in event-B

Figure B.16: Step AT4 - Translation of state graph and initial graph attributes (Def. 33)



(a) Attributed rule r_1

```
sets
    AttrL1  // set of attribute vertices of L1
constants
    attrvL1 // assigns a graph vertex to each attibute vertex of L1
    tL1_A   // assigns a type to each of attribute vertex nof L1
    NodeId1 NodeStack1 // names of attribute vertices of L1
axioms
  @axm_AttrL1 partition(AttrL1,{NodeId1},{NodeStack1})
  @axm_attrvL1 attrvL1 ∈ AttrL1 → VertL1
  @axm_attrvL1def partition(attrvL1,{NodeId1↦N11},{NodeStack1↦N11})
  @axm_tL1_A tL1_A ∈  AttrL1 →  AttrT
  @axm_tL1_Adef partition(tL1_A,{NodeId1↦NodeId},{NodeStack1↦NodeStack})
```

(b) Attributes of the typed graph L_1 in event-B

```
event r1 extends r1
  any
      mA
      Del_A
      newNodeStack
      nid
      ns
      ns1
  where
    @grd_mA mA ∈ AttrL1 → AttrG
    @grd_DelA Del_A = mA[{NodeStack1}]
    @grd_newNodeStack newNodeStack ∈ N \ AttrG
    @grd_nid nid ∈ N
    @grd_ns ns ∈ Stack
    @grd_ns1 ns1 ∈ Stack
    @grd_attrvNodeId1 mV(attrvL1(NodeId1)) = attrvG(mA(NodeId1))
    @grd_attrvNodeStack1 mV(attrvL1(NodeStack1)) = attrvG(mA(NodeStack1))
    @grd_tANodeId1 tL1_A(NodeId1) = tG_A(mA(NodeId1))
    @grd_tANodeStack1 tL1_A(NodeStack1) = tG_A(mA(NodeStack1))
    @grd_val_NodeStack1 ns = valG_NodeStack(mA(NodeStack1)) // ns is the value of attribute NodeStack
    @grd_val_NodeId1 nid = valG_NodeId(mA(NodeId1)) // nid is the value of attribute NodeId
    @grd_eqn1 top(ns) ≠ nid         // rule equation 1
    @grd_eqn2 ns1 = push(nid↦ns)    // rule equation 2
  then
    @act_A AttrG ≔ (AttrG \ Del_A) ∪ {newNodeStack}
    @act_valGNodeStack valG_NodeStack ≔  ( Del_A ⩤ valG_NodeStack) ∪ {newNodeStack ↦ ns1}
    @act_attrv attrvG ≔  ( Del_A ⩤ attrvG) ∪ {newNodeStack ↦ mV(N11)}
    @act_tGA tG_A ≔  ( Del_A ⩤ tG_A) ∪ {newNodeStack ↦ NodeStack}
end
```

(c) Extension of rule application r_1 in event-B

Figure B.17: Step AT5 - Translation of each attributed rule (Def. 34)

Appendix C. Rodin Proofs

This Appendix contains the goals and commented proof trees generated in the Rodin environment for some properties. The main points where help was necessary in the proofs are marked with red circles and comments. The most difficult kind of help is add hypothesis (ah), that are the points where the user enters a new hypothesis to aid the proof process (this hypothesis, once proven, can be used to prove the goal). Another kind of help that needs attention is instantiation (when we instantiate some hypothesis with concrete values, indicated by the circled $\forall$ sign, followed by an instantiation). Proof by cases is indicated by the circled $\vee$ (when we click this symbol, the Rodin environment offers the possibility to split the proof in corresponding cases). Figures C.18, C.19, C.20 and C.21 are examples of proofs of `uniqueTok`, whereas Figs. C.22 and C.23 are examples of proofs of property `fairness`.

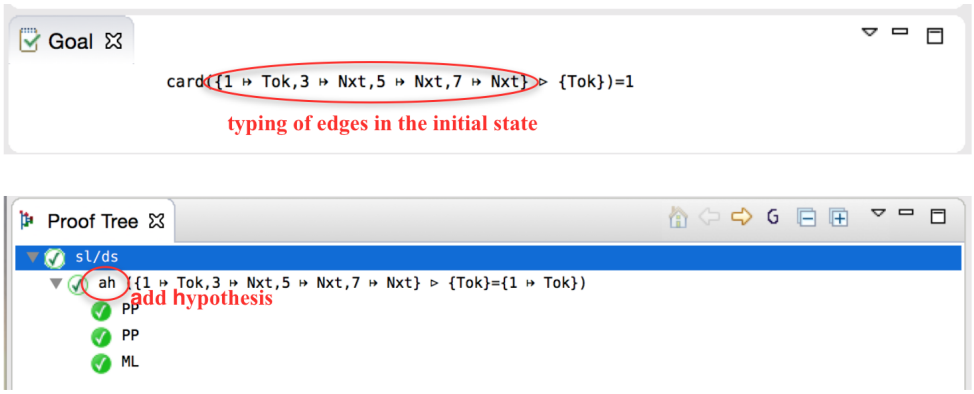


Figure C.18: Proof that the Initialization Event establishes `uniqueTok`

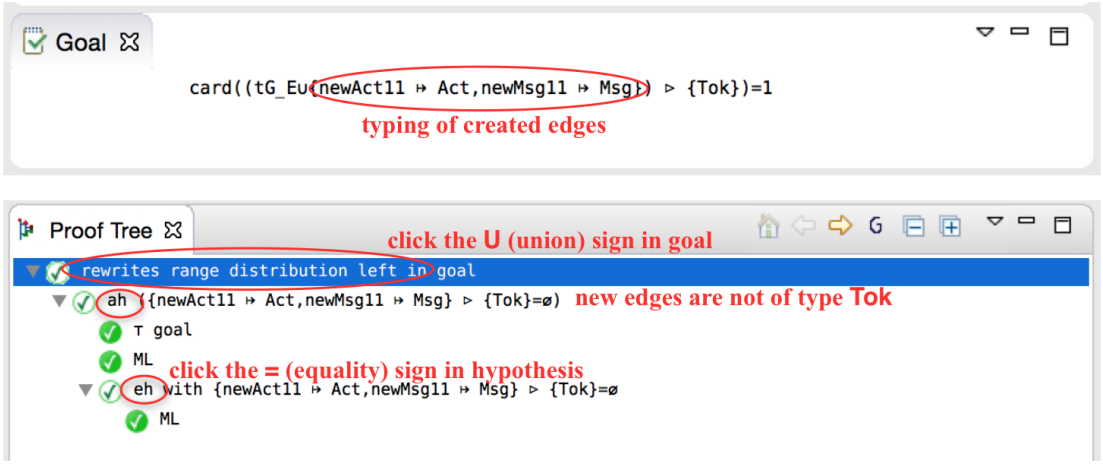


Figure C.19: Proof that rule $r1$ preserves `uniqueTok`

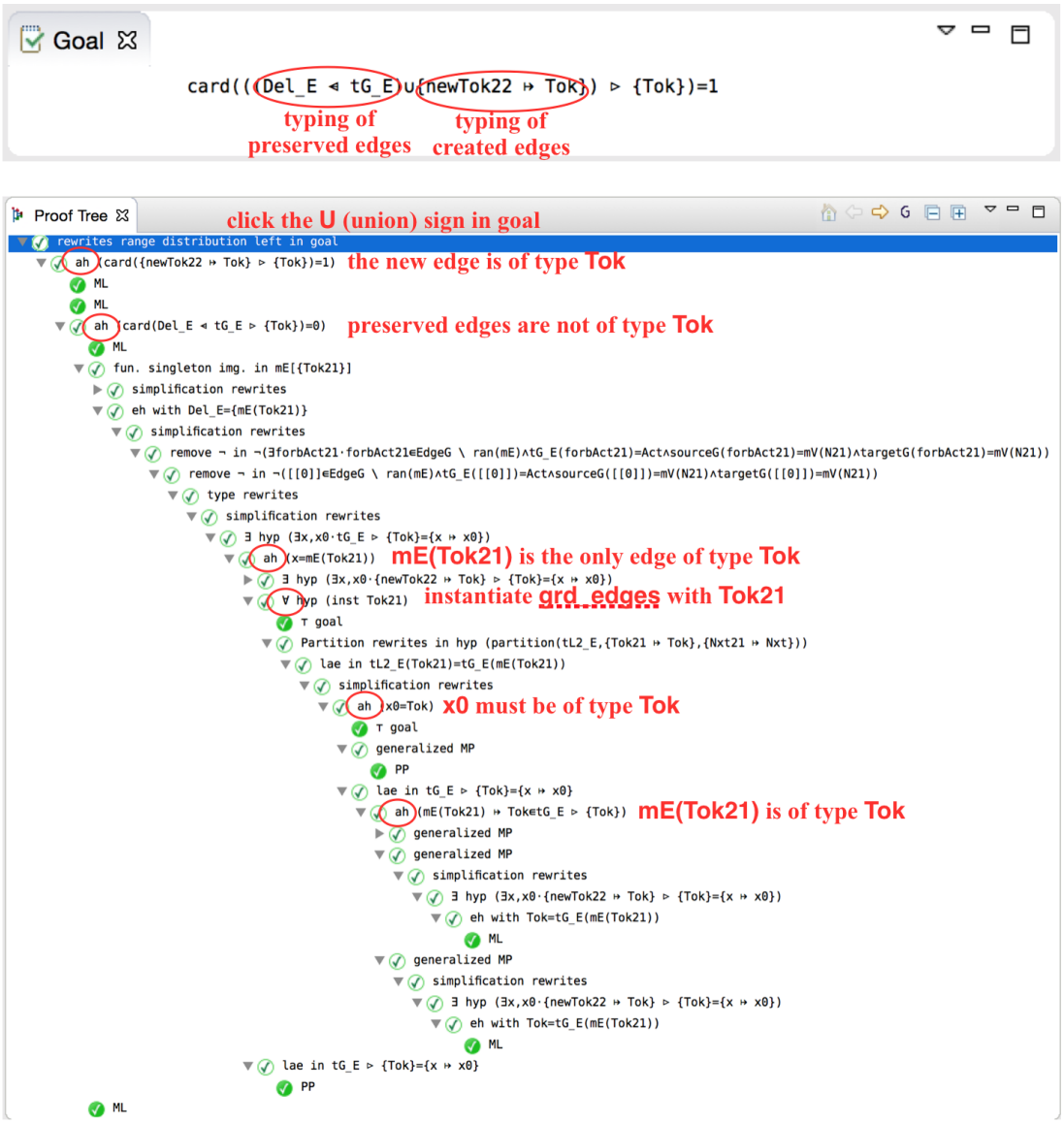


Figure C.20: Proof that rule $r2$ preserves `uniqueTok`

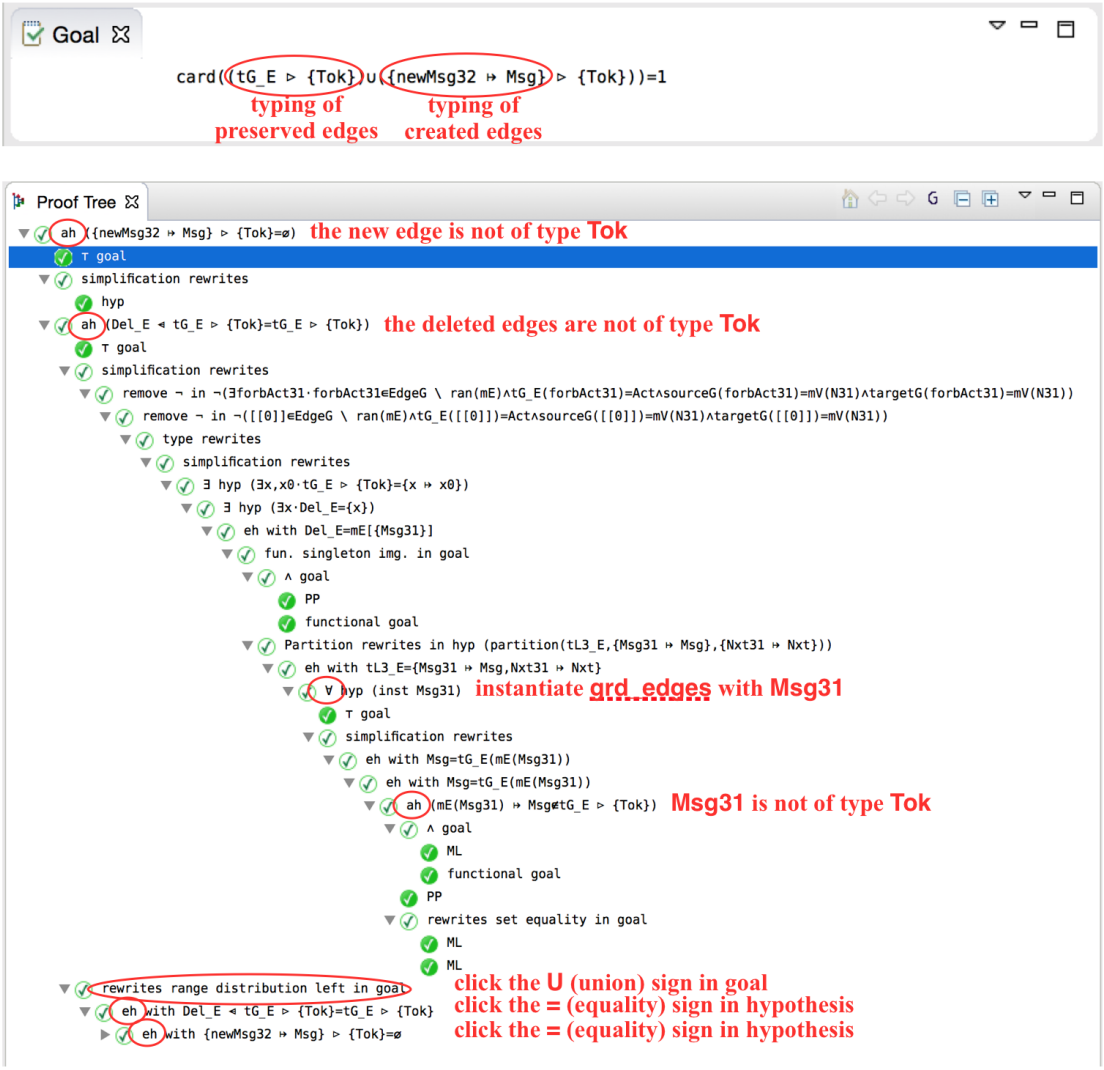


Figure C.21: Proof that rule $r3$ preserves `uniqueTok`

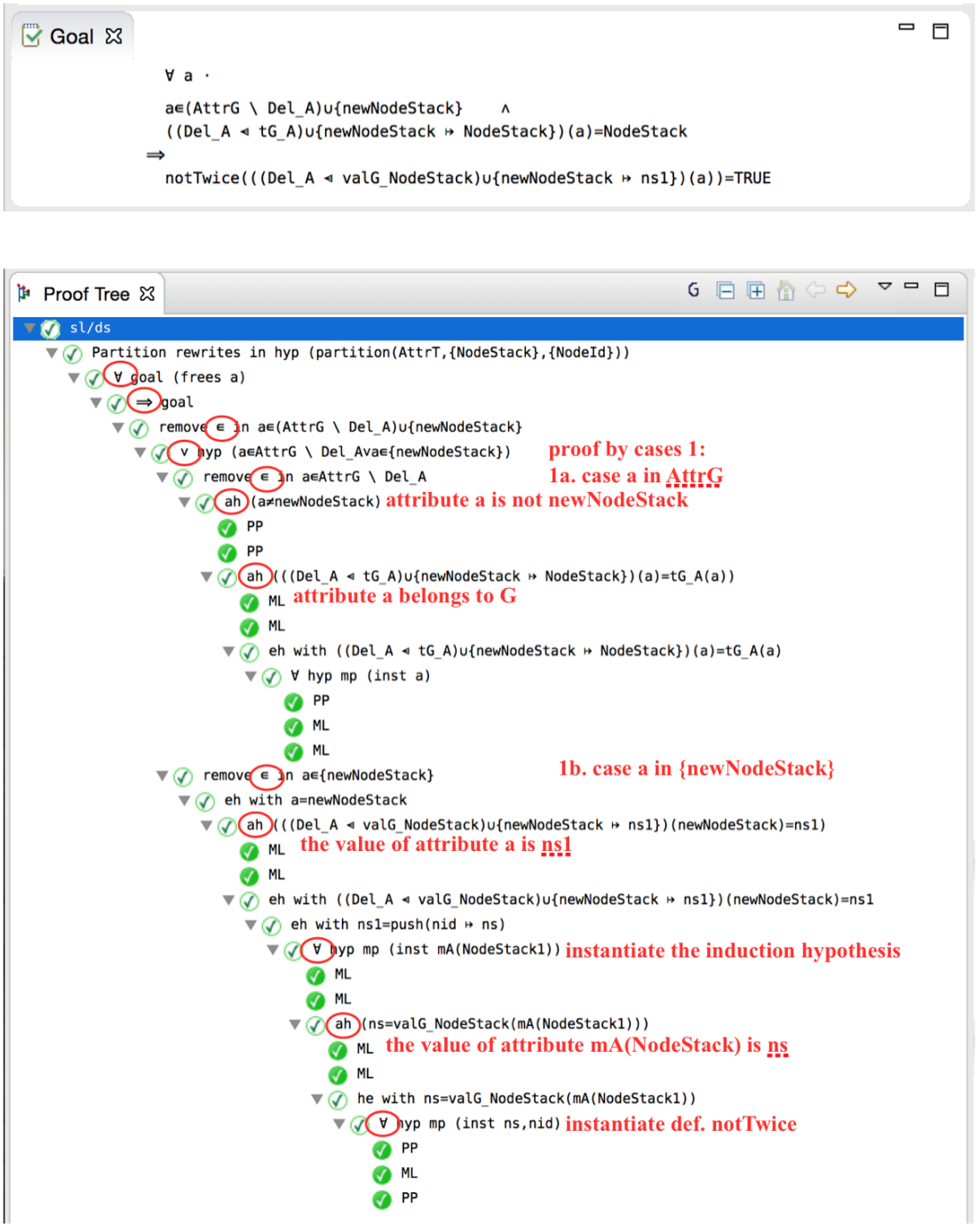


Figure C.22: Proof that rule $r1$ preserves `fairness`

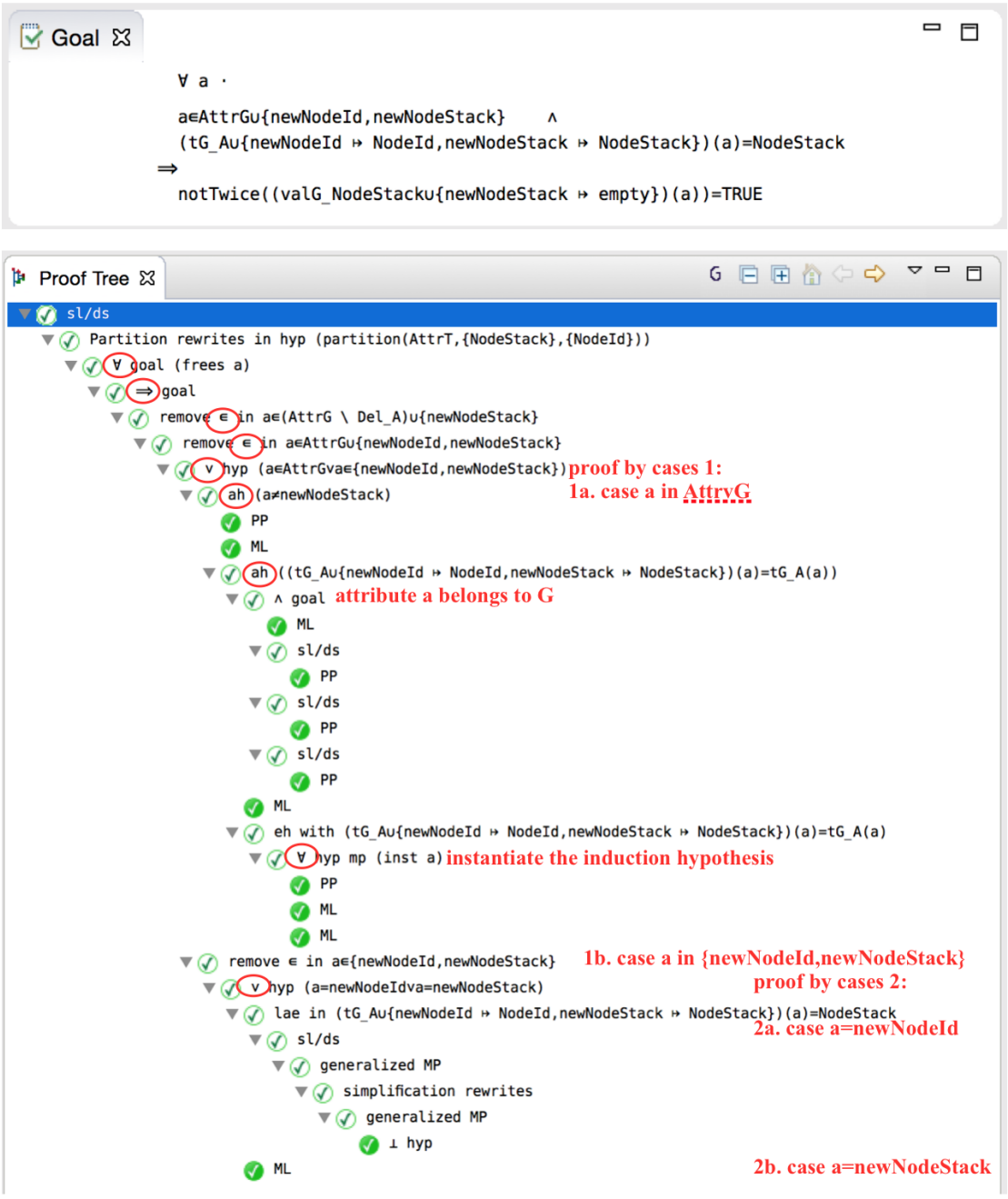


Figure C.23: Proof that rule $r5$ preserves `fairness`