UNIVERSITÀ CA' FOSCARI DI VENEZIA

Dipartimento di Informatica

Technical Report Series in Computer Science

Rapporto di Ricerca CS-2004-10

Novembre 2004

Paolo Baldan, Andrea Corradini, Barbara König

Verifying Finite-State Graph Grammars: an Unfolding-Based Approach

Dipartimento di Informatica, Università ca' Foscari di Venezia
via Torino, 155 – 30172 Mestre-Venezia, Italy.
TEL: +39 041 2348411 FAX: +39 041 2348419

Verifying Finite-State Graph Grammars: an Unfolding-Based Approach*

Paolo Baldan[1], Andrea Corradini[2], and Barbara König[3]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Institutsverbund Informatik, Universität Stuttgart, Germany

baldan@dsi.unive.it andrea@di.unipi.it koenigba@fmi.uni-stuttgart.de

Abstract. We propose a framework where behavioural properties of finite-state systems modelled as graph transformation systems can be expressed and verified. The technique is based on the unfolding semantics and it generalises McMillan's complete prefix approach, originally developed for Petri nets, to graph transformation systems. It allows to check properties of the graphs reachable in the system, expressed in a monadic second order logic.

1 Introduction

Graph transformation systems (GTSs) are recognised as an expressive specification formalism, especially suited for concurrent and distributed systems [9]: the (topo)logical distribution of a system can be naturally represented by using a graphical structure and the dynamics of the system, e.g., the reconfigurations of its topology, can be modelled by means of graph rewriting rules. Moreover GTSs can be seen as a proper generalisation of a classical model of concurrency, i.e., Petri nets, since the latter are essentially rewriting systems on (multi)sets, the rewriting rules being the transitions.

The concurrent behaviour of GTSs has been thoroughly studied and a consolidated theory of concurrency for GTSs is available, including the generalisation of several semantics of Petri nets, like process and unfolding semantics (see, e.g., [6, 21, 2]). However, only recently, building on these semantical foundations, some efforts have been devoted to the development of frameworks where behavioural properties of GTSs can be expressed and verified (see [12, 15, 13, 22, 20, 1]).

As witnessed, e.g., by the approaches in [17, 10] for Petri Nets, truly concurrent semantics are potentially useful in the verification of finite-state systems, in that they help to avoid the combinatorial explosion arising when one explores all possible interleavings of events. Still, to the best of our knowledge, no technique based on partial order (process or unfolding) semantics has been proposed for the verification of finite-state GTSs.

* Research partially supported by EU FET-GC Project AGILE, the EC RTN SEGRAVIS, DFG project SANDS and EPSRC grant R93346/01.

In this paper we contribute to this topic by proposing a verification framework for *finite-state graph transformation systems* based on their unfolding semantics. Our technique is inspired by the approach originally developed by McMillan for Petri nets [17] and further developed by many authors (see, e.g., [10, 11, 24]). More precisely, our technique applies to any *graph grammar*, i.e., any set of graph rewriting rules with a fixed start graph (the initial state of the system), which is *finite-state* in a liberal sense: the set of graphs which can be reached from the start graph, considered not only *up to isomorphism*, but also *up to isolated nodes*, is finite. Hence in a finite-state graph grammar in our sense there is not actually a bound to the number of nodes generated in a computation, but only to the nodes which are connected to some edge at each stage of the computation. Some similarities exist with name-based process calculi, where a process may generate an unbounded number of names still being essentially finite-state since a finite number of them is actually used at each time. Existing model-checking tools, such as SPIN [14], usually do not directly support the creation of an arbitrary number of objects while still maintaining a finite state space, making entirely non-trivial their use for checking finite-state GTSs or process calculi agents with name creation. In the field of process calculi this has led, for instance, to the introduction of so-called HD-automata [19].

As a first step we face the problem of identifying a finite, still useful fragment of the unfolding of a GTS. In fact, the unfolding construction for GTSs produces a structure which fully describes the concurrent behaviour of the system, including all possible steps and their mutual dependencies, as well as all reachable states. However, the unfolding is infinite for non-trivial systems, and cannot be used directly for model-checking purposes.

Following McMillan's approach, we show that given any finite-state graph grammar $\mathcal{G}$ a *finite* fragment of its unfolding which is *complete*, i.e., which provides full information about the system as far as reachability (and other) properties are concerned, can be characterised as the maximal prefix of the unfolding not including *cut-off events*. A cut-off event is intuitively an event which does not contribute to generating new states. For Petri nets this is formalised by defining a cut-off, roughly, as a transition which produces the same marking as other transitions but with a larger causal history. This notion turns out to be inappropriate for GTSs, because, due to the possibility of performing "contextual" rewritings (i.e., of preserving part of the state in a rewriting step), an event can have several different local histories: a problem widely studied, e.g., in the setting of contextual nets [18, 23, 3, 24]. Consequently, the generalised notion of cut-off for GTSs must take into account the multiplicity of the local histories.

Unfortunately the characterisation of the finite complete prefix is not constructive. Hence, while leaving as an open problem the definition of a general algorithm for constructing such a prefix, we identify a significant subclass of graph grammars where an adaptation of the existing algorithms for Petri nets is feasible. These are called *read-persistent* graph grammars by analogy with the terminology used in the work on contextual nets [24].

The complete finite prefix provides a "compact" representation of the state space of a system, not only because it avoids the combinatorial explosion arising from the possible interleavings of concurrent events, but also because several branches of computations are merged until a real choice point is reached. To fruitfully exploit the prefix, in the second part of the paper we consider a logic $\mathcal{L}2$ where graph properties of interest can be expressed, like the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (related to security properties) or cycles (related to deadlock-freedom). This is a monadic second-order logic over graphs where quantification is allowed over (sets of) edges. (Similar logics are considered in [8] and, in the field of verification, in [20, 4].) Then we show how a complete finite prefix of a graph grammar can be used to verify properties of the graphs reachable in $\mathcal{G}$, expressed in the logic $\mathcal{L}2$. This is done by exploiting both the graphical structure underlying the prefix and the concurrency information it provides.

The rest of the paper is organised as follows. Section 2 introduces graph transformation systems and their unfolding semantics. Section 3 characterises a finite complete prefix for finite-state GTSs and discusses the problem of constructing such a prefix. Section 4 introduces a logic for GTSs, showing how it can be checked over a finite complete prefix. Finally, Section 5 draws some conclusions and indicates directions of further research. Two Appendices contain a detailed definition of the unfolding construction and the proofs for Section 3, respectively.

This technical report is an extended version of [5] which was published in the proceedings of CONCUR'04.

2 Unfolding semantics of graph grammars

This section presents the notion of graph rewriting used in the paper. Rewriting takes place on so-called *typed graphs*, namely graphs labelled over a structure that is itself a graph [6]. It can be seen as a set-theoretical presentation of an instance of algebraic (single- or double-pushout) rewriting (see, e.g., [7]). Next we review the notion of occurrence grammar, which is instrumental in defining the unfolding of a graph grammar [2, 21].

2.1 Graph Transformation Systems

In the following, given a set A we denote by A^* the set of finite strings of elements of A. Given $u \in A^*$ we write $|u|$ to indicate the length of u. If $u = a_0 \dots a_n$ and $0 \leq i \leq n$, by $[u]_i$ we denote the i-th element a_i of u. Furthermore, if $f : A \to B$ is a function then we denote by $f^* : A^* \to B^*$ its extension to strings.

A *(hyper)graph* G is a tuple (V_G, E_G, c_G), where V_G is a set of nodes, E_G is a set of edges and $c_G : E_G \to {V_G}^*$ is a connection function. A node $v \in V_G$ is called *isolated* if it is not connected to any edge. Given two graphs G, G', a *graph morphism* $\phi : G \to G'$ is a pair of total functions $\langle \phi_V : V_G \to V_{G'},\ \phi_E : E_G \to E_{G'} \rangle$ such that for every $e \in E_G$ it holds that ${\phi_V}^*(c_G(e)) = c_{G'}(\phi_E(e))$.

When clear from the context, the subscripts V and E of the components of graph morphisms will be omitted.

Definition 1 (typed graph). *Given a graph (of types) T, a* typed graph G *over T is a graph $|G|$, together with a morphism $type_G : |G| \to T$. A* morphism *between T-typed graphs $f : G_1 \to G_2$ is a graph morphism $f : |G_1| \to |G_2|$ consistent with the typing, i.e., such that $type_{G_1} = type_{G_2} \circ f$.*

The graph of types T can be thought of as a set of labels (with some additional structure) and for any x in a T-typed graph G, its type $type_G(x)$ can be read as the label of x.

A typed graph G is called *injective* if the typing morphism $type_G$ is injective. More generally, given $n \in \mathbb{N}$, the graph is called *n-injective* if for any item x in T, $|type_G^{-1}(x)| \leq n$, namely if the number of "instances of resources" of any type x is bounded by n. Given two (typed) graphs G and G' we will write $G \simeq G'$ to mean that G and G' are *isomorphic*, and $G \ddot{\simeq} G'$ when G and G' are isomorphic once their isolated nodes have been removed. In the last case we say that G and G' are *isomorphic up to isolated nodes*.

In the sequel we extensively use the fact that given a graph G, any subgraph of G without isolated nodes is identified by the set of its edges. Precisely, given a subset of edges $X \subseteq E_G$, we denote by $graph(X)$ the least subgraph of G (actually the unique subgraph, up to isolated nodes) having X as set of edges.

We will use some set-theoretical operations on (typed) graphs with "componentwise" meaning. Let G and G' be T-typed graphs. We say that G and G' are *consistent* if the structure $G \cup G'$ defined as $(V_{|G|} \cup V_{|G'|}, E_{|G|} \cup E_{|G'|}, c_G \cup c_{G'})$, typed by $type_G \cup type_{G'}$, is a well-defined T-typed graph. In this case also the intersection $G \cap G'$, constructed in a similar way, is well-defined. Given a graph G and a set (of edges) E we denote by $G - E$ the graph obtained from G by removing the edges in E (formally, $|G - E| = (V_{|G|}, E_{|G|} - E, c_{G'})$, where $c_{G'}$ is the restriction of c_G, typed by $type_{G-E}$ which is the restriction of $type_G$). Sometimes we will also refer to the items (nodes and edges) in $G - G'$, where G and G' are graphs, although the structure resulting as the componentwise set-difference of G and G' might not be a well-defined graph.

Definition 2 (production). *Given a graph of types T, a* T-typed production *is a pair of finite consistent T-typed graphs $q = (L, R)$, often written $L \to R$, such that*

1. *$L \cup R$ and L do not include isolated nodes;*
2. *$V_{|L|} \subseteq V_{|R|}$;*
3. *$E_{|L|} - E_{|R|}$ and $E_{|R|} - E_{|L|}$ are non-empty.*

Informally, a rule $L \to R$ specifies that, once an occurrence of L is found in a graph G, then G can be rewritten by removing (the image in G of) the items in $L - R$ and adding the items in $R - L$. The (image in G of the) items in $L \cap R$ instead are left unchanged: they are, in a sense, preserved or read by the rewriting step.

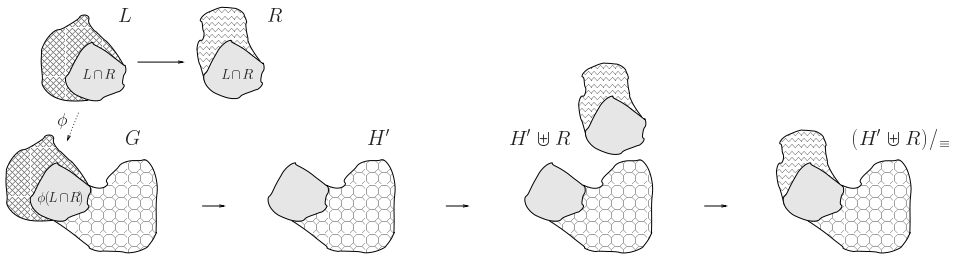

Fig. 1. A rewriting step, schematically.

This informal explanation should also motivate Conditions 1–3 above. Condition 1 essentially states that we are interested only in rewriting up to isolated nodes: by the requirement on $L \cup R$, no node is isolated when created and, by the requirement on L, nodes that become isolated have no influence on further reductions. Thus one can safely assume that isolated nodes are removed by some kind of garbage collection. Consistently with this view, by Condition 2 productions cannot delete nodes. Deletion of nodes could be managed at the price of introducing heavy technical machinery, which is, in our view, unnecessary, since the deletion of a node can be simulated by leaving that node isolated. Finally Condition 3 ensures that every production consumes and produces at least one edge: a requirement corresponding to T-restrictedness in the theory of Petri nets.

Definition 3 (graph rewriting). *Let $q = L \to R$ be a T-typed production. A* match *of q in a T-typed graph G is a morphism $\phi : L \to G$, satisfying the* identification condition*, i.e., for any $e, e' \in E_{|L|}$, if $\phi(e) = \phi(e')$ then $e, e' \in E_{|R|}$. In this case G rewrites to the graph H, obtained as $H = ((G - \phi(E_{|L|} - E_{|R|})) \uplus R)/_{\equiv}$, where $\equiv$ is the least equivalence on the items of the graph such that $x \equiv \phi(x)$. We write $G \Rightarrow_{q,\phi} H$ or simply $G \Rightarrow_q H$.*

A rewriting step is schematically represented in Fig. 1. Intuitively, in the graph $H' = G - \phi(E_{|L|} - E_{|R|})$ the images of all the edges in $L - R$ have been removed. Then in order to get the resulting graph, merge R to H' along the image through ϕ of the preserved subgraph $L \cap R$. Formally the resulting graph H is obtained by first taking $H' \uplus R$ and then by identifying, via the equivalence relation $\equiv$, the image through ϕ of each item in $L \cap R$ with the corresponding item in R.

Note that by the *identification condition* if a production requires the deletion of two edges, then two *distinct* edges must be actually deleted in G. Instead, two edges in L which are preserved can be mapped to the same item in G.

Definition 4 (graph transformation system and graph grammar). *A* graph transformation system (GTS) *is a triple $\mathcal{R} = \langle T, P, \pi \rangle$, where T is a graph of types, P is a set of production names and π is a function which assigns to each production name $q \in P$ a T-typed production $\pi(q) = L_q \to R_q$. A* graph grammar *is a tuple $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ where $\langle T, P, \pi \rangle$ is a GTS and G_s is a*

finite T-typed graph, without isolated nodes, called the start graph. *We denote by $Elem(\mathcal{G})$ the (disjoint) union $E_T \uplus P$, i.e., the set of edges in the graph of types and the production names. We say that $\mathcal{G}$ is* finite *if the set $Elem(\mathcal{G})$ is finite.*

A T-typed graph G is *reachable* in $\mathcal{G}$ if for some graph $G' \simeq G$, we have $G_s \Rightarrow^*_{\mathcal{G}} G'$, where $\Rightarrow^*_{\mathcal{G}}$ is the transitive closure of the rewriting relation induced by the productions of $\mathcal{G}$.

We remark that Place/Transition Petri nets can be viewed as a very special subclass of typed graph grammars. Say that a graph G is *edge-discrete* if its set of nodes is empty and thus for any edge e in G, $c_G(e)$ is the empty string, i.e., the edge has no connection. Given a P/T net P, let T_P be the edge-discrete graph having the set of places of P as edges. Then any finite edge-discrete graph typed over T_P can be seen as a marking of P: an edge typed over s represents a token in place s. Using this correspondence, a production $L_t \to R_t$ faithfully represents a transition t of P if L_t encodes the marking *pre-set*(t), R_t encodes *post-set*(t), and $L_t \cap R_t = \emptyset$. The graph grammar corresponding to a Petri net is finite iff the original net has finitely many places and transitions. It is worth stressing that the generalisation from edge-discrete to proper graphs radically changes the expressive power of the formalism. For instance, it can be easily shown that, unlike P/T Petri nets, the class of grammars considered in this paper is Turing complete.

Example 1. Consider the graph grammar $\mathcal{CP}$, modeling a system where three *processes* of type P are connected to a *communication manager* of type CM (see the start graph in Fig. 2, where edges are represented as rectangles and nodes as small circles). Two processes may establish a new connection with each other via the communication manager, becoming *processes engaged* in communication (typed PE, the only edge with more than one connection). This transformation is modelled by the production [engage] in Fig. 2: observe that a new node connecting the two processes is created. The second production [release] describes the termination of the communication between the two partners. A typed graph G over $T_{\mathcal{CP}}$ is drawn by labeling each edge or node x of G with "$: type_G(x)$". Only when the same graphical item x belongs to both the left- and the right-hand side of a production we include its identity in the label (which becomes "$x : type_G(e)$"): in this case we also shade the item, to stress that it is preserved by the production. This example is not meant to be meaningful or realistic, but rather it has been chosen to illustrate, in a simple situation, the notions and concepts introduced in the paper.

We next discuss the notion of safety for graph grammars [6]. It generalises the one for P/T nets which requires that each place contains at most one token in any reachable marking. More generally, we extend to graph grammars the notion of n-boundedness.

Definition 5 (safe grammar). *For a fixed $n \in \mathbb{N}$, we say that a graph grammar $\mathcal{G}$ is* n-bounded *if for all graphs H reachable in $\mathcal{G}$ there is an n-injective graph H' such that $H' \ddot{\simeq} H$. A 1-bounded grammar will be called* safe.

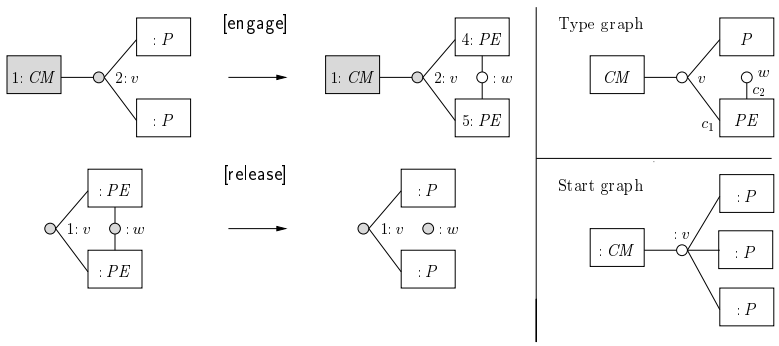

Fig. 2. The finite-state graph grammar $\mathcal{CP}$.

The definition can be understood by thinking of edges of the graph of types T as a generalisation of places in Petri nets. In this view the number of different edges of a graph which are typed on the same item of T corresponds to the number of tokens contained in a place. Observe that for *finite* graph grammar, n-boundedness amounts to the property of being finite-state (up to isomorphism and up to isolated nodes). In the sequel when considering a finite-state graph grammar we will (often implicitly) assume that it is also finite.

For instance, the graph grammar $\mathcal{CP}$ in Fig. 2 is clearly 3-bounded and thus finite-state (but only up to isolated nodes).

2.2 Nondeterministic Occurrence Grammars

In this subsection we introduce nondeterministic occurrence grammars, a special class of grammars which are used as semantical model for general grammars: the unfolding construction maps every grammar $\mathcal{G}$ into an occurrence grammar which provides a static description of the computations of $\mathcal{G}$, recording the events (production applications) which appear in all possible derivations and the dependency relations among them. While for nets it suffices to take into account only the causality and conflict relations, for grammars the fact that a production application not only consumes and produces, but also preserves a part of the state leads to a form of asymmetric conflict between productions.

When a graph grammar $\mathcal{G}$ is safe, and thus reachable graphs are injectively typed, at every step, for any item t in the type graph every production can consume, preserve and produce a single item typed t. Hence we can safely think that a production, according to its typing, *consumes*, *preserves* and *produces* items of the graph of types. Using a net-like language, we speak of *pre-set* ${}^\bullet q$, *context* $\underline{q}$ and *post-set* $q^\bullet$ of a production q. Since we work with graphs considered up to isolated nodes, we will record in these sets only edges. The proper graphical structure will always be recoverable from the graph of types.

Definition 6 (pre-set, context, post-set of productions). *For any production q of a graph grammar $\mathcal{G} = \langle T, G_s, P, \pi \rangle$, we define*

$^\bullet q = type_{L_q}(E_{|L_q|} - E_{|R_q|})$ $\quad \underline{q} = type_{L_q}(E_{|L_q \cap R_q|})$ $\quad q^\bullet = type_{R_q}(E_{|R_q|} - E_{|L_q|})$

Furthermore, for any edge e in T we define $^\bullet e = \{q \in P : e \in q^\bullet\}$, $\underline{e} = \{q \in P : e \in \underline{q}\}$, $e^\bullet = \{q \in P : e \in {}^\bullet q\}$. *This notation is extended also to nodes in the obvious way, e.g., for $v \in V_T$ we define* $^\bullet v = \{q \in P : v \in type_{R_q}(V_{|R_q|} - V_{|L_q|})\}$.

An example of safe grammar can be found in Fig. 3 (for the moment ignore its relation to grammar $\mathcal{CP}$ in Fig. 2). For this grammar, $^\bullet\mathsf{engage1} = \{2{:}P, 3{:}P\}$, $\underline{\mathsf{engage1}} = \{1{:}CM\}$ and $\mathsf{engage1}^\bullet = \{5{:}PE, 6{:}PE\}$, while $^\bullet 1{:}CM = \emptyset$, $\underline{1{:}CM} = \{\mathsf{engage1}, \mathsf{engage2}, \mathsf{engage3}\}$ and $3{:}P^\bullet = \{\mathsf{engage1}, \mathsf{engage3}\}$.

Definition 7 (causal relation). *The* causal relation *of a safe grammar $\mathcal{G}$ is the least transitive relation $<$ over $Elem(\mathcal{G})$ satisfying: for any edge e in the graph of types T, and for productions $q, q' \in P$*

1. *if $e \in {}^\bullet q$ then $e < q$;*
2. *if $e \in q^\bullet$ then $q < e$;*
3. *if $q^\bullet \cap \underline{q'} \neq \emptyset$ then $q < q'$.*

As usual $\leq$ is the reflexive closure of $<$. Moreover, for $x \in Elem(\mathcal{G})$ we denote by $\lfloor x \rfloor$ the set of causes of x in P, namely $\lfloor x \rfloor = \{q \in P : q \leq x\}$.

Observe that the fact that an item is preserved by q and consumed by q', i.e., $\underline{q} \cap {}^\bullet q' \neq \emptyset$ does not imply $q < q'$. In this case, the dependency between the two productions is a kind of *asymmetric conflict* (see [3, 18, 16, 24]): The application of q' prevents q from being applied, so that q can never follow q' in a derivation (or, equivalently, when both productions q and q' occur in a derivation then q must precede q').

Definition 8 (asymmetric conflict). *The* asymmetric conflict relation *of a safe grammar $\mathcal{G}$ is the binary relation $\nearrow$ over the set of productions P, defined by $q \nearrow q'$ if:*

1. $\underline{q} \cap {}^\bullet q' \neq \emptyset$;
2. $^\bullet q \cap {}^\bullet q' \neq \emptyset$ *and* $q \neq q'$;
3. $q < q'$.

Condition 1 is justified by the discussion above. Condition 2 essentially expresses the fact that the ordinary symmetric conflict is encoded, in this setting, as an asymmetric conflict in both directions. More generally, we will write $q \# q'$ and say that q and q' are in *conflict* when the causes of q and q', i.e., $\lfloor q \rfloor \cup \lfloor q' \rfloor$, includes a cycle of asymmetric conflict. Finally, since $<$ represents a global order of execution, while $\nearrow$ determines an order of execution only locally to each computation, it is natural to impose $\nearrow$ to be an extension of $<$ (Condition 3).

A *nondeterministic occurrence grammar* is an acyclic grammar which represents, in a branching structure, several possible computations beginning from its start graph and using each production at most once.

Definition 9 ((nondeterministic) occurrence grammar). *A safe grammar $\mathcal{O} = \langle T, G_s, P, \pi \rangle$ is called a* (nondeterministic) occurrence grammar *if*

1. *its causal relation $\leq$ is a partial order, and, for any $q \in P$, the set $\lfloor q \rfloor$ is finite and the asymmetric conflict $\nearrow$ is acyclic on $\lfloor q \rfloor$;*
2. *the start graph G_s is the generated by the set $Min(\mathcal{O})$ of minimal elements of $\langle Elem(\mathcal{O}), \leq \rangle$, typed over T by the inclusion, i.e., $|G_s| = graph(Min(\mathcal{O}))$;*[4]
3. *any edge or node x in T is created by at most one production in P, namely $|\ {}^\bullet x\ | \leq 1$;*
4. *for each $q \in P$, the typing $type_{L_q}$ is injective on the "consumed" items in $|L_q| - |R_q|$, and $type_{R_q}$ is injective on the "produced" items in $|R_q| - |L_q|$.*

Since the start graph of an occurrence grammar $\mathcal{O}$ is determined by $Min(\mathcal{O})$, we often do not mention it explicitly.

Intuitively, Conditions 1–3 recast in the framework of graph grammars the analogous conditions of occurrence nets (actually of occurrence contextual nets [3, 24]). In particular, in Condition 1, the acyclicity of asymmetric conflict on $\lfloor q \rfloor$ corresponds to the requirement of irreflexivity for the conflict relation in occurrence nets. In fact, notice that if a set of productions forms an asymmetric conflict cycle $q_0 \nearrow q_1 \nearrow \ldots \nearrow q_n \nearrow q_0$, then such productions cannot appear in the same computation, otherwise the application of each production should precede the application of the production itself; this fact can be naturally interpreted as a form of n-ary conflict. Condition 4, instead, is closely related to safety and requires that each production consumes and produces items with multiplicity one. An example of occurrence grammar is given in Fig. 3.

2.3 Concurrent Subgraphs, Configurations and Histories

The finite computations of an occurrence grammar are characterised by special subsets of productions closed under causal dependencies and with no conflicts (i.e., cycles of asymmetric conflict).

Definition 10 (configuration). *Let $\mathcal{O} = \langle T, P, \pi \rangle$ be an occurrence grammar. A configuration of $\mathcal{O}$ is a finite subset of productions $C \subseteq P$ such that*

1. *$\nearrow_C$, the asymmetric conflict restricted to C, is acyclic;*
2. *for any $q \in C$, $\lfloor q \rfloor \subseteq C$.*

We define a computational order on configurations so that C_1 precedes C_2 when C_1 can be extended to C_2 by applying the productions in $C_2 - C_1$. Due to the presence of asymmetric conflicts this order is not simply subset inclusion. In fact one must be sure that productions in C_1 do not prevent the productions in $C_2 - C_1$ from being applied.

Definition 11 (poset of configurations). *Given two configurations C_1, C_2 of an occurrence grammar $\mathcal{O}$ we write $C_1 \sqsubseteq C_2$ if*

1. *$C_1 \subseteq C_2$;*

[4] Notice that $Min(\mathcal{O}) \subseteq E_T$, i.e., it does not contain productions by Condition 3 of Definition 2.

2. *for any* $q_1 \in C_1$, $q_2 \in C_2$, *if* $q_2 \nearrow q_1$ *then* $q_2 \in C_1$.

The set of all configurations of $\mathcal{O}$, *ordered by* $\sqsubseteq$, *is denoted by* $Conf(\mathcal{O})$.

Proposition 1 (rechability of graphs generated by configurations). *Let* $\mathcal{O}$ *be an occurrence grammar,* $C \in Conf(\mathcal{O})$ *be a configuration and*

$$\mathbf{G}(C) = graph((Min(\mathcal{O}) \cup \bigcup_{q \in C} q^\bullet) - \bigcup_{q \in C} {}^\bullet q).$$

Then a graph G *such that* $G \ddot{\simeq} \mathbf{G}(C)$ *can be obtained from the start graph of* $\mathcal{O}$, *by applying all the productions in* C *in any order compatible with* $\nearrow$.

Due to the presence of asymmetric conflicts, given a production q, the history of q, i.e., the set of events which must precede q in a computation is not uniquely determined by q, but it depends also on the particular computation. Essentially, the history of q can include or not the productions which are in asymmetric conflict with q.

Definition 12 (history). *Let* $\mathcal{O}$ *be an occurrence grammar, let* $C \in Conf(\mathcal{O})$ *be a configuration and let* $q \in C$. *The* history *of* q *in* C *is the set of events* $C[\![q]\!] = \{q' \in C : q' \nearrow_C^* q\}$. *We denote by* $Hist(q)$ *the set of histories of* q, *i.e.,* $Hist(q) = \{C[\![q]\!] : C \in Conf(\mathcal{O})\}$.

As in the case of nets, reachable states are characterised in terms of a concurrency relation.

Definition 13 (concurrent graph). *Let* $\mathcal{O} = \langle T, P, \pi \rangle$ *be an occurrence grammar. A finite subset of edges* $E \subseteq E_T$ *is called* concurrent, *written* $co(E)$, *if*

1. $\nearrow_E$, *the asymmetric conflict restricted to* $\bigcup_{x \in E} \lfloor x \rfloor$, *is acyclic;*
2. $\neg(x < y)$ *for all* $x, y \in E$.

A subgraph G *of* T *is called concurrent, written* $co(G)$, *if* $co(E_G)$.

The maximal concurrent subgraphs G of T corresponds exactly (up to isolated nodes) to the graphs reachable from the start graph (by means of a derivation which applies all the productions in $\bigcup_{x \in E_G} \lfloor x \rfloor$ exactly once in any order compatible with $\nearrow$).

2.4 Unfolding of graph grammars

The unfolding construction, when applied to a grammar $\mathcal{G}$, produces a nondeterministic occurrence grammar $\mathcal{U}(\mathcal{G})$ describing the behaviour of $\mathcal{G}$. An unfolding construction for the double-pushout algebraic approach to graph rewriting has been proposed in [2]: the construction sketched here is slightly simpler because productions cannot delete nodes and thus the so-called dangling condition does not play a role (see also the work on the single-pushout approach to graph rewriting [21]). A more detailed description of the construction can be found in Appendix A.

Intuitively, the construction begins from the start graph of $\mathcal{G}$, and then applies in all possible ways its productions to concurrent subgraphs, recording in the unfolding each occurrence of production and each new graph item generated in the rewriting process.

Definition 14 (unfolding - sketch). *Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a graph grammar. The unfolding $\mathcal{U}(\mathcal{G}) = \langle T', G'_s, P', \pi' \rangle$ is defined as the "componentwise" union of the following inductively defined sequence of occurrence graph grammars $\mathcal{U}(\mathcal{G})^{[n]}$.*

($\mathbf{n = 0}$) The grammar $\mathcal{U}(\mathcal{G})^{[0]}$ consists of the start graph $|G_s|$, with no productions.

($\mathbf{n \to n + 1}$) Let $q \in P$ be a production of $\mathcal{G}$ and let m be a match of q in the graph of types of $\mathcal{U}(\mathcal{G})^{[n]}$, satisfying the identification condition and such that $m(|L_q|)$ is concurrent.

Then the occurrence grammar $\mathcal{U}(\mathcal{G})^{[n+1]}$ is obtained by "recording" in $\mathcal{U}(\mathcal{G})^{[n]}$ the application of q at the match m. More precisely, a new production $q' = \langle q, m \rangle$ is added and the graph of types $T^{[n]}$ is extended by adding to it a copy of each item generated by the application q, without deleting any item.

The unfolding is mapped over the original grammar by the so-called folding morphism $\chi = \langle \chi_T, \chi_P \rangle : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$. The first component $\chi_T : T' \to T$ is a graph morphism mapping each graph item in the (graph of types of) the unfolding to the corresponding item in the (graph of types of) the original grammar $\mathcal{G}$. The second component $\chi_P : P' \to P$ maps any production occurrence $\langle q, m \rangle$ in the unfolding to the corresponding production q of $\mathcal{G}$.

Keeping in mind that productions and items of the graph of types for GTSs play the role of transitions and place names for Petri nets, we have a clear analogy between the unfolding constructions for GTSs and Petri nets: in fact, for Petri nets the unfolding is again a Petri net (taken from a special class) and the folding morphism maps transitions and places of the unfolding to the corresponding transition and places of the original net.

The occurrence grammar in Fig. 3 is an initial part of the (infinite) unfolding of the grammar $\mathcal{CP}$ in Fig. 2. For instance, production engage1 is an occurrence of production engage in $\mathcal{CP}$, applied at the match consisting of the edges 1:CM, 2:P, 3:P. Unfolding such a match, three new graph items, two edges 5:PE, 6:PE and a node, are added to the graph of types of the unfolding. Note that the graph of types of the (partial) unfolding (call it $T_{\mathcal{T}}$) is typed over the graph of types $T_{\mathcal{CP}}$ of the original grammar (via the folding morphism $\chi_T : T_{\mathcal{T}} \to T_{\mathcal{CP}}$). This explains why the edges of the graphs in the productions of the unfolding, which are typed over $T_{\mathcal{T}}$, are marked with names including two colons.

The unfolding provides a compact representation of the behaviour of $\mathcal{G}$, and in particular it represents all the graphs reachable in $\mathcal{G}$, in the following sense.

Theorem 1 (completeness of the unfolding). *Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a graph grammar. Then a T-typed graph G is reachable in $\mathcal{G}$ iff there exists a maximal concurrent subgraph X' of the graph of types of $\mathcal{U}(\mathcal{G})$ such that $G \simeq \langle X', \chi_{T|X'} \rangle$.*

3 Finite Prefix for Graph Grammars

We first introduce a generalised notion of cut-off which, for finite-state (up to isomorphism and up to isolated nodes) graph grammars, allows us to characterise a finite and complete prefix of the unfolding.

Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ denote a graph grammar, fixed throughout the section, and let $\mathcal{U}(\mathcal{G}) = \langle T', P', \pi' \rangle$ be its unfolding with $\chi : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$ the folding morphism, as in Definition 14. Given a configuration C of $\mathcal{U}(\mathcal{G})$, recall from Proposition 1 that $\mathbf{G}(C)$ denotes the subgraph of T' reached after the execution of the productions in C (up to isolated nodes). We shall denote by $Reach(C)$ the same graph, seen as a graph typed over T by the restriction of the folding morphism, i.e., $Reach(C) = \langle \mathbf{G}(C), \chi_{T\,|\mathbf{G}(C)} \rangle$.

To identify a finite prefix of the unfolding the idea consists of avoiding to keep in the unfolding useless productions, i.e., productions which do not contribute to generating new graphs. The definition of "cut-off event" introduced by McMillan for Petri nets in order to formalise such a notion has to be adapted to this context, since for graph grammars a production may have different histories.

Definition 15 (cut-off). *A production $q \in P'$ of the unfolding $\mathcal{U}(\mathcal{G})$ is called a* cut-off *if there exists $q' \in P'$ such that $Reach(\lfloor q \rfloor) = Reach(\lfloor q' \rfloor)$ and $|\lfloor q' \rfloor| < |\lfloor q \rfloor|$.*

A production q is called a strong cut-off *if for all $C_q \in Hist(q)$ there exists a $q' \in P'$ and $C_{q'}$ in $Hist(q')$ such that*

1. *$Reach(C_q) \simeq Reach(C_{q'})$ and*
2. *$|C_{q'}| < |C_q|$.*

The truncation *of $\mathcal{U}(\mathcal{G})$ is the greatest prefix $\mathcal{T}(\mathcal{G})$ of $\mathcal{U}(\mathcal{G})$ not containing strong cut-offs.*

The proofs of the following theorems can be found in Appendix B.

Theorem 2 (completeness of the truncation). *The truncation $\mathcal{T}(\mathcal{G})$ is a complete prefix of the unfolding, i.e., for any reachable graph G of $\mathcal{G}$ there is a configuration C in $Conf(\mathcal{T}(\mathcal{G}))$ such that $Reach(C) \dddot{\simeq} G$.*

For finite n-bounded grammars, where the number of possible states (up to isomorphism and up to isolated nodes) is finite, the truncation of the unfolding of the grammar is finite.

Theorem 3 (finiteness). *Let $\mathcal{G}$ be a finite graph grammar. If $\mathcal{G}$ is n-bounded then the truncation $\mathcal{T}(\mathcal{G})$ is finite.*

Unfortunately, the proof of the above theorems does not suggest immediately a way of constructing the truncation for finite-state graph grammars. While leaving the solution for the general case as an open problem, we next discuss how a finite complete prefix can be derived for still interesting classes of graph grammars.

Let $\mathcal{G}$ be a finite-state graph grammar. The problem essentially resides in the difficulty of finding an order of generation for the productions which ensures that, whenever a production q' is classified as a strong cut-off by looking only at the *partial* informations provided by the prefix under construction, then q' is really a strong cut-off. More precisely, consider a production q which satisfies the defining properties of a strong cut-off, but only with respect to a given prefix of the unfolding (i.e., in Definition 15, q', $Hist(q)$ and $Hist(q')$ are taken in the prefix and not in the full unfolding). Later the prefix can be extended with a new production q' which is in asymmetric conflict with q, i.e., such that $q' \nearrow q$. But in this way q gains new possible histories containing q', and such histories may generate new graphs, not generated by other productions.

The obvious idea of generating a production q in the prefix only after all the productions q' such that $q' \nearrow q$ (and q' not in conflict with q) have been inserted does not work, in general, since $\nearrow$ might not be finitary and thus the set of productions which should be generated before a given q can be infinite. In other words, the set of possible histories for a production can be infinite.

The prefix can be constructed if we limit our attention to classes of graph grammars where the set of possible histories for each production is finite. In particular, this holds whenever the asymmetric conflict becomes inessential, in the sense that it is subsumed by the other relations. We call this property "read-persistence" since, viewing graph grammars as generalised Petri nets, it appears as the graph grammar theoretical version of the read-persistence for contextual nets as formulated in [24].

Definition 16 (read-persistence). *An occurrence grammar $\mathcal{O} = \langle T, P, \pi \rangle$ is called* read-persistent *if for any $q_1, q_2 \in P$, if $q_1 \nearrow q_2$ then $q_1 \leq q_2$ or $q_1 \# q_2$. A graph grammar $\mathcal{G}$ is called* read-persistent *if its unfolding $\mathcal{U}(\mathcal{G})$ is read-persistent.*

It can be shown that an adaptation of the algorithm originally proposed in [17] for ordinary nets and extended in [24] to read-persistent contextual nets, works for read-persistent graph grammars. In particular, the notion of strong cut-off can be safely replaced by the weaker notion of (ordinary) cut-off. Roughly, the algorithm works on a graph grammar $\mathcal{G}$ as follows:

1. Start from the initial graph of $\mathcal{G}$;
2. Let M be the set of possible new productions (pairs $q' = \langle q, m \rangle$, where q is a production in $\mathcal{G}$ and m is a concurrent match of the left-hand side of q in the type graph of the current prefix).
3. While $M \neq \emptyset$
   - Take $q' \in M$ (with minimal $|\lfloor q' \rfloor|$) and let $M := M - \{q'\}$;
   - If q' is not a cut-off, then unfold q', adding it to the prefix (see Definition 14);
   - Update M;

An obvious class of read-persistent grammars consists of all the grammars $\mathcal{G}$ where productions do not preserve any edge, i.e., where for any production q the intersection $L_q \cap R_q$ is a discrete graph. Note that, as far as reachable graphs

are concerned, such grammars are as expressive as the general ones, since the preservation of an edge can be simulated by deleting and creating again the edge with the same connections. Instead, in this way the degree of concurrency of the system might decrease since while two rewriting steps "preserving" the same item can run concurrently, their encodings using productions that delete and create again the item are forced to be executed sequentially. Technically, this encoding might increase the number of causal dependencies and thus may make the complete prefix larger. More generally, a grammar is read-persistent whenever any two productions q_1 and q_2 such that $\underline{q_1} \cap {}^\bullet q_2 \neq \emptyset$ cannot be applicable at the same time.

Observe that, for instance, the grammar $\mathcal{CP}$ in our running example is read-persistent, since the communication manager CM, the only edge preserved by productions, is never consumed. Its truncation is the graph grammar $\mathcal{T}(\mathcal{CP})$ depicted in Fig. 3.

Denote by $T_{\mathcal{T}}$ the type graph of the truncation. Note that applying the production [release] to any subgraph of $T_{\mathcal{T}}$ matching its left-hand side would result in a cut-off: this is the reason why $\mathcal{T}(\mathcal{CP})$ does not include any instance of production [release]. The start graph of the truncation is isomorphic to the start graph of grammar $\mathcal{CP}$ and it is mapped injectively to the graph of types $T_{\mathcal{T}}$ in the obvious way.

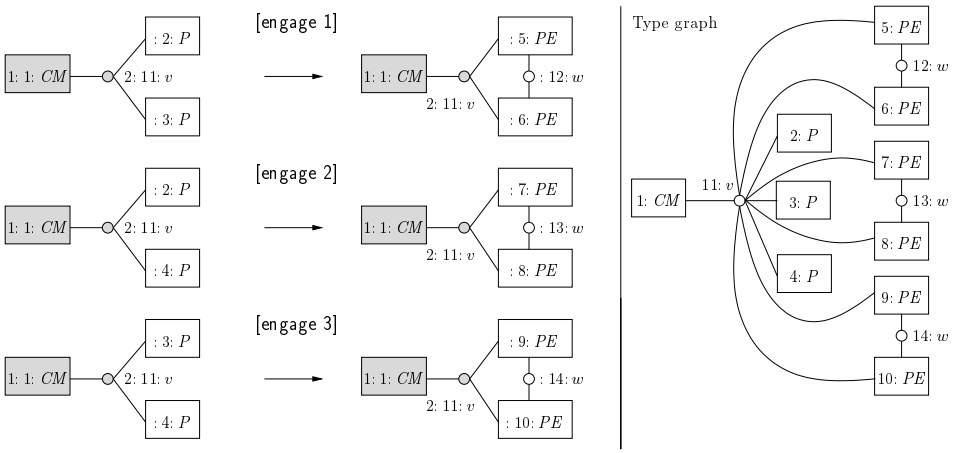

Fig. 3. The truncation $\mathcal{T}(\mathcal{CP})$ of the graph grammar in Fig. 2.

In general, the truncation of a grammar like $\mathcal{CP}$ but where n processes are connected to CM in the start graph, will contain $\frac{n(n-1)}{2}$ productions. If we would consider all possible interleavings we would end up with an exponential number of productions.

4 Exploiting the prefix

In this section we propose a monadic second-order logic $\mathcal{L}2$ where some graph properties of interest can be expressed. Then we show how the validity of a property in $\mathcal{L}2$ over all the reachable graphs of a finite-state grammar $\mathcal{G}$ can be verified by exploiting a complete finite prefix.

4.1 A logic on graphs

Let us first introduce the monadic second order logic $\mathcal{L}2$ for specifying graph properties. Quantification is allowed over edges, but not over nodes (as, e.g., in [8]).

Definition 17 (Graph formulae). *Let $\mathcal{X}_1 = \{x, y, z, \ldots\}$ be a set of (first-order) edge variables and let $\mathcal{X}_2 = \{X, Y, Z, \ldots\}$ be a set of (second-order) variables representing edge sets. The set of* graph formulae *of the logic $\mathcal{L}2$ is defined as follows, where $\ell \in \Lambda$, $i, j \in \mathbb{N}$:*

$$\begin{array}{rcll} F & ::= & x = y \mid c_i(x) = c_j(y) \mid \mathit{type}(x) = \ell \mid x \in X & \textit{(Predicates)} \\ & & F \vee F \mid F \wedge F \mid \neg F & \textit{(Connectives)} \\ & & \forall x.F \mid \exists x.F \mid \forall X.F \mid \exists X.F & \textit{(Quantifiers)} \end{array}$$

We denote by $\mathit{free}(F)$ and $\mathit{Free}(F)$ the sets of first-order and second-order variables, respectively, occurring free in F, defined in the obvious way.

Given a T-typed graph G, a formula F in $\mathcal{L}2$, and two fixed valuations $\sigma : \mathit{free}(F) \to E_{|G|}$ and $\Sigma : \mathit{Free}(F) \to \mathcal{P}(E_{|G|})$ for the free first- and second-order variables of F, respectively, the *satisfaction relation* $G \models_{\sigma,\Sigma} F$ is defined inductively, in the usual way; for instance:

$$\begin{array}{rcl} G \models_{\sigma,\Sigma} x = y & \iff & \sigma(x) = \sigma(y) \\ G \models_{\sigma,\Sigma} c_i(x) = c_j(y) & \iff & |c_G(\sigma(x))| \geq i \wedge |c_G(\sigma(y))| \geq j \wedge [c_G(\sigma(x))]_i = [c_G(\sigma(y))]_j \\ G \models_{\sigma,\Sigma} \mathit{type}(x) = \ell & \iff & \mathit{type}_G(\sigma(x)) = \ell \\ G \models_{\sigma,\Sigma} x \in X & \iff & \sigma(x) \in \Sigma(X) \\ G \models_{\sigma,\Sigma} \forall X.F & \iff & G \models_{\sigma,\Sigma'} F \text{ for any } \Sigma' \text{ such that } \Sigma'(Y) = \Sigma(Y) \\ & & \text{for } Y \in \mathcal{X}_2 - \{X\}, \text{ and } \Sigma'(X) \in \mathcal{P}(E_G) \end{array}$$

The logic $\mathcal{L}2$ can be used to express quite interesting properties of graphs. The presence of second-order quantifiers allows, for example, to write a closed formula that is satisfied by all and only acyclic graphs, a property not expressible in the first-order fragment. Note that, although arbitrary quantification over nodes is not allowed, if the graph of types T is fixed, it is possible to refer to a *non-isolated* node using a term of the form $c_i(x)$. Note also that the maximum arity of edges in T is a bound also for the arity of edges in T-typed graphs.

It can be shown that the logical equivalence induced by $\mathcal{L}2$ on finite graphs is graph isomorphism up to isolated nodes, i.e., two (finite) graphs are isomorphic up to isolated nodes iff they satisfy the same formulae in $\mathcal{L}2$.

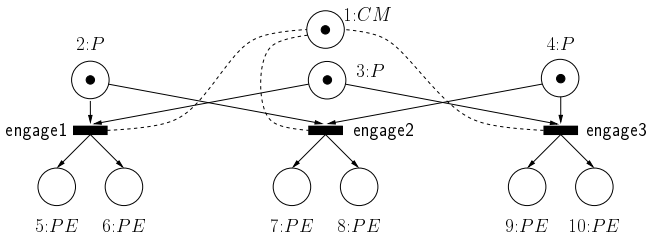

Fig. 4. The Petri net underlying the truncation $\mathcal{T}(\mathcal{CP})$ in Fig. 3

A simple, but fundamental observation is that, while for n-bounded graph grammars the graphical nature of the state (nodes, edges and their connections) plays a basic role, for any occurrence grammar $\mathcal{O}$ we can can forget about the graphical nature of the states and view $\mathcal{O}$ as an occurrence contextual net (i.e., a Petri net with read arcs, see, e.g., [3, 24]).

Definition 18 (Petri net underlying an occurrence grammar). *Let $\mathcal{O} = \langle T', P', \pi' \rangle$ be an occurrence grammar. The* contextual Petri net underlying $\mathcal{O}$, *denoted by* $\mathsf{Net}(\mathcal{O})$, *is the Petri net having the set of edges $E_{T'}$ as places and a transition for every production $q \in P'$, with pre-set ${}^\bullet q$, post-set $q^\bullet$ and context $\underline{q}$.*

For instance, the Petri net $\mathsf{Net}(\mathcal{T}(\mathcal{CP}))$ underlying the truncation of $\mathcal{CP}$ (see Fig. 3) is depicted in Fig. 4. Read arcs are represented as dotted undirected lines.

Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a fixed finite-state graph grammar and consider the truncation $\mathcal{T}(\mathcal{G}) = \langle T', P', \pi' \rangle$ (actually, all the results hold for any complete finite prefix of the unfolding). Notice that, by completeness of $\mathcal{T}(\mathcal{G})$, any graph reachable in $\mathcal{G}$ is (up to isolated nodes) a subgraph of the graph of types T' of $\mathcal{T}(\mathcal{G})$, typed over T by the restriction of the folding morphism $\chi : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$. Also observe that a safe marking m of $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ can be seen as a graph typed over the type graph T of the original grammar $\mathcal{G}$: take the least subgraph of T' having m as set of edges, i.e., $\mathit{graph}(m)$, and type it over T by the restriction of the folding morphism. With a slight abuse of notation this typed graph will be denoted simply as $\mathit{graph}(m)$.

We show how any formula ϕ in $\mathcal{L}2$ can be translated to a formula $M(\phi)$ over the safe markings of $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ such that, for any marking m *reachable* in $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$

$$\mathit{graph}(m) \models \phi \quad \text{iff} \quad m \models M(\phi).$$

The syntax of the formulae over markings is

$$\phi ::= e \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \to \phi,$$

where the basic formulae e are place (edge) names, meaning that the place is marked, i.e., $m \models e$ if $e \in m$. Logical connectives are treated as usual.

Definition 19 (Encoding graph formulae into multiset formulae). *Let $\mathcal{T}(\mathcal{G})$ be the truncation of a graph grammar $\mathcal{G}$, as above. Let F be graph formula in $\mathcal{L}2$, let $\sigma : free(F) \to E_{T'}$ and $\Sigma : Free(F) \to P(E_{T'})$. The* encoding M *is defined as follows:*

$$M[x = y, \sigma, \Sigma] = \begin{cases} true \ \ if\ \sigma(x) = \sigma(y) \\ false \ \ otherwise \end{cases}$$

$$M[c_i(x) = c_j(y), \sigma, \Sigma] = \begin{cases} true \ \ if\ |c_{T'}(\sigma(x))| \geq i \ \wedge \ |c_{T'}(\sigma(y))| \geq j \\ \qquad\qquad \wedge \ [c_{T'}(\sigma(x))]_i = [c_{T'}(\sigma(y))]_j \\ false \ otherwise \end{cases}$$

$$M[type(x) = \ell, \sigma, \Sigma] = \begin{cases} true \ \ if\ \chi_T(\sigma(x)) = \ell \\ false \ \ otherwise \end{cases}$$

$$M[x \in X, \sigma, \Sigma] = \begin{cases} true \ \ if\ \sigma(x) \in \Sigma(X) \\ false \ \ otherwise \end{cases}$$

$$M[F_1 \vee F_2, \sigma, \Sigma] = M[F_1, \sigma, \Sigma] \vee M[F_2, \sigma, \Sigma]$$

$$M[F_1 \wedge F_2, \sigma, \Sigma] = M[F_1, \sigma, \Sigma] \wedge M[F_2, \sigma, \Sigma]$$

$$M[\neg F, \sigma, \Sigma] = \neg M[F, \sigma, \Sigma]$$

$$M[\exists x.F, \sigma, \Sigma] = \bigvee_{e \in E_{T'}} (e \ \wedge \ M[F, \sigma \cup \{x \mapsto e\}, \Sigma])$$

$$M[\forall x.F, \sigma, \Sigma] = \bigwedge_{e \in E_{T'}} (e \ \to M[F, \sigma \cup \{x \mapsto e\}, \Sigma]$$

$$M[\exists X.F, \sigma, \Sigma] = \bigvee_{E \subseteq E_{T'},\ co(E)} \Big(\bigwedge E \ \wedge \ M[F, \sigma, \Sigma \cup \{X \mapsto E\}] \Big)$$

$$M[\forall X.F, \sigma, \Sigma] = \bigwedge_{E \subseteq E_{T'},\ co(E)} \Big(\bigwedge E \to M[F, \sigma, \Sigma \cup \{X \mapsto E\}] \Big)$$

where, for $E = \{e_1, \ldots, e_n\}$, the symbol $\bigwedge E$ stands for $e_1 \wedge \ldots \wedge e_n$. If F is closed formula (i.e., without free variables), we define $M[F] = M[F, \emptyset, \emptyset]$.

Note that, since every reachable graph in $\mathcal{G}$ is isomorphic to a subgraph of T', typed by the restriction of χ_T, the encoding resolves the basic predicates by exploiting the structural information of T'. When a first-order variable x in a formula is mapped to an edge e, we take care that the edge is marked, and, similarly, when a second-order variable X in a formula is mapped to a set of edges E, such a set must be covered. Observe that in this case E is limited to range only over concurrent subsets of edges. In fact, if E is a non-concurrent set, then no reachable marking m will include E, i.e., $m \not\models \bigwedge E$.

It is possible to show that the above encoding is correct.

Proposition 2 (correctness of the encoding). *Let $\mathcal{T}(\mathcal{G})$ be the truncation of $\mathcal{G}$. Let ϕ be a formula in $\mathcal{L}2$. Then for any pair of valuations $\sigma : \mathcal{X}_1 \to E_{T'}$ and $\Sigma : \mathcal{X}_2 \to \mathcal{P}(E_{T'})$, and for any safe marking m over $E_{T'}$*

$$graph(m) \models_{\sigma,\Sigma} \phi \quad \Leftrightarrow \quad m \models M[\phi, \sigma, \Sigma]$$

4.2 Checking properties of reachable graphs

Let $\mathcal{G} = \langle G_s, T, P, \pi \rangle$ be a finite-state graph grammar. We next show how a complete finite prefix of $\mathcal{G}$ can be used to check whether, given a formula $F \in \mathcal{L}2$, there exists some reachable graph which satisfies F. In this case we will write $\mathcal{G} \models \Diamond F$. Note that the same algorithm allows to check "invariants" of a graph grammars, i.e., to verify whether a property $F \in \mathcal{L}2$ is satisfied by all graphs reachable in $\mathcal{G}$, written $\mathcal{G} \models \Box F$. In fact, it trivially holds that $\mathcal{G} \models \Box F$ iff $\mathcal{G} \not\models \Diamond \neg F$.

Let $\mathcal{T}(\mathcal{G}) = \langle T', P', \pi' \rangle$ be the truncation of $\mathcal{G}$ (or any complete prefix of the unfolding) and let $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ be the underlying Petri net. The formula produced by the encoding in Definition 19 can be simplified by exploiting the mutual relationships between items as expressed by the causality, (asymmetric) conflict and concurrency relation.

Proposition 3 (simplification). *Let F be any formula in $\mathcal{L}2$, let $\sigma : free(F) \to E_{T'}$ and $\Sigma : Free(F) \to \mathcal{P}(E_{T'})$ be valuations. If m is a marking reachable in $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ and η is a marking formula obtained by simplifying $M[F, \sigma, \Sigma]$ with the* Simplification Rule *below:*

If $S \subseteq E_{T'}$ and $\neg co(S)$ then replace the subformula $\bigwedge S$ by false.

then $graph(m) \models_{\sigma,\Sigma} F$ iff $m \models \eta$.

Algorithm. The question "$\mathcal{G} \models \Diamond F$?" is answered by working over $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ as follows:

- Consider the formula over markings $M[F]$ (see Definition 19);
- Express $M[F]$ in disjunctive normal form, i.e., as a disjunction of conjunctions of atoms (literals or negation of literals) as below, where $a_{i,j}$ can be e or $\neg e$ for $e \in E_{T'}$.
$$\eta = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{k_i} a_{i,j};.$$
- Apply the Simplification Rule in Proposition 3, as far as possible, thus obtaining a formula η';
- For any conjunct in η' of the kind $e_1 \wedge \ldots \wedge e_h \wedge \neg e'_1 \wedge \ldots \wedge \neg e'_l$ proceed as follows:
  - Take the configuration $C = \lfloor \{e_1, \ldots, e_h\} \rfloor$ (observe that $co(\{e_1, \ldots, e_h\})$, otherwise we could remove the conjunct by the Simplification Rule of Proposition 3).
  - Consider the safe marking reached after C, i.e., $m_C = (m_0 \cup \bigcup_{t \in C} t^\bullet) - \bigcup_{t \in C} {}^\bullet t$, where m_0 is the initial marking of $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ (minimal places). Surely m_C includes $\{e_1, \ldots, e_h\}$. Hence, the only reason why the conjunct can not be true is that m_C includes some of the $\{e'_1, \ldots, e'_l\}$. In this case look for a configuration $C' \supseteq C$, which enriches C with transitions which consume the e'_j but not the e_i.
    To this aim take all the conflict free sets of the following form

$$S' = \{t_1, \ldots, t_l : \forall i \in \{1, \ldots, l\}.\ t_i \in e_i'^{\bullet}\}$$

Conclude with success iff for some such S'

1. for any $t \in \lfloor S' \rfloor$, for any $i \in \{1, \ldots, h\}$. $e_i \not< t$
2. $C \cup \lfloor S' \rfloor$ is a configuration.

– The formula $\Diamond F$ holds if and only if this check succeeds for at least one conjunct.

It can be shown that the above algorithm is correct. In particular, note that in point 2 above we can disregard the fact that an event has multiple histories. In fact, if $C \cup \lfloor S' \rfloor$ includes a conflict (cycle of asymmetric conflicts) we will surely not be able to get rid of it by considering larger histories for the events involved.

Observe that the Simplification Rule can be applied at any stage of the above process, in particular also during or immediately after the translation from a graph formula to a formula on markings. In practice, this might be quite convenient since it can allow to avoid the transformation into disjunctive normal form of relevant fragments of the formula.

As an example, suppose that we want to check that our sample graph grammar $\mathcal{CP}$ satisfies $\Box F$, where F is a $\mathcal{L}2$ formula specifying that every engaged process is connected through connection c_2 to exactly one other engaged process, i.e.,

$$F = \forall x.(\mathit{type}(x) = PE \ \Rightarrow\ \exists y.(x \neq y \ \wedge\ \mathit{type}(y) = PE \ \wedge\ c_2(x) = c_2(y) \\ \wedge\ \forall z.(\mathit{type}(z) = PE \ \wedge\ x \neq z \ \wedge\ c_2(x) = c_2(z) \ \Rightarrow\ y = z))).$$

The encoding $\phi = M[F]$ simplifies to

$$\phi \equiv (5{:}\,PE \iff 6{:}\,PE) \wedge (7{:}\,PE \iff 8{:}\,PE) \wedge (9{:}\,PE \iff 10{:}\,PE)$$

and we have to check that the truncation does not satisfy

$$\Diamond\neg\phi = \Diamond[(5{:}\,PE \wedge \neg 6{:}\,PE) \vee (\neg 5{:}\,PE \wedge 6{:}\,PE) \vee (7{:}\,PE \wedge \neg 8{:}\,PE) \\ \vee (\neg 7{:}\,PE \wedge 8{:}\,PE) \vee (9{:}\,PE \wedge \neg 10{:}\,PE) \vee (\neg 9{:}\,PE \wedge 10{:}\,PE)],$$

which can be done by using the described verification procedure.

5 Conclusions

We have discussed how the finite prefix approach, originally introduced by McMillan for Petri nets, can be generalised to graph transformation systems. A generalised notion of cut-off has been introduced which allows to characterise, for any graph grammar which is finite-state, up to isomorphism and up to isolated nodes, a finite complete prefix of the unfolding as (the largest) cut-off free prefix. A complete finite prefix can be constructed for some classes of graph grammars, but the problem of constructing it for general, possibly non-read-persistent grammars remains open and represents an interesting direction

of further research. Also, it would be interesting to try to determine an upper bound on the size of the prefix, with respect to the number of reachable graphs.

We have shown how the complete finite prefix can be used to model-check some properties of interest for graph transformation systems. We plan to generalise the verification technique proposed here to allow the model-checking of more expressive logics where temporal modalities can be arbitrarily nested, like the one studied in [10] for Petri nets. We intend to implement the model-checking procedure described in the paper and, as in the case of Petri nets, we expect that its efficiency could be improved by refined cut-off conditions (see, e.g., [11]) which help to decrease the size of the prefix.

An alternative approach could be based on the encoding of finite-state graph grammars into simpler formalisms, where a theory of verification is already available. Some attempts for the translation of finite-state graph grammars into Petri nets are not encouraging, suggesting that a direct approach is preferable. In fact the degree of concurrency drastically decreases in the encoding and, since all the possible graphical configurations of the system must be represented by the structure of the net, one gets an enormous explosion in the number of transitions and places. Further complications relate to the fact that the grammars of interest are finite-state only up to isolated nodes.

The conceptual relation with HD-automata and the existence of several encodings of process calculi for mobility into graph rewriting systems, also suggest the possibility of exploiting the complete finite prefix to verify behavioural properties of mobile systems. Checking our technique against concrete case studies coming from this field appears as a stimulating direction of further research.

As mentioned in the introduction, some effort has been devoted recently to the development of suitable verification techniques for GTSs. The papers [12, 13] present a general theory of verification without providing directly applicable techniques. In [15, 1, 4] one can find techniques which are applicable to infinite-state systems: the first defines a general framework based on types for graph rewriting, while the second is based on the construction of suitable approximations of the behaviour of the GTS. Instead, the papers [22, 20] concentrate on finite-state GTSs. They both generate a suitable labelled transition system out of a given finite-state GTS and then [22] resorts to model-checkers like SPIN, while [20] discusses the decidability of the model-checking problem for a logic, based on regular path expressions, allowing to talk about the history of nodes along computations. The main difference with respect to our work is that they do not exploit a partial order semantics, with an explicit representation of concurrency, and thus considering the possible interleavings of concurrent events these techniques may suffer of the state-explosion problem.

References

1. P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In K.G. Larsen and M. Nielsen, editors, *Proceedings of CONCUR 2001*, volume 2154 of *LNCS*, pages 381–395. Springer Verlag, 2001.

2. P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer Verlag, 1999.
3. P. Baldan, A. Corradini, and U. Montanari. Contextual Petri nets, asymmetric event structures and processes. *Information and Computation*, 171(1):1–49, 2001.
4. P. Baldan, B. König, and B. König. A logic for analyzing abstractions of graph transformation systems. In R. Cousot, editor, *Proceedings of SAS'03*, volume 2694 of *LNCS*, pages 255–272. Springer Verlag, 2003.
5. Paolo Baldan, Andrea Corradini, and Barbara König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR '04*, pages 83–98. Springer-Verlag, 2004. LNCS 3170.
6. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.
7. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations.* World Scientific, 1997.
8. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, chapter 5. World Scientific, 1997.
9. H. Ehrig, J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism and Distribution.* World Scientific, 1999.
10. J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2–3):151–195, 1994.
11. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(20):285–310, 2002.
12. F. Gadducci, R. Heckel, and M. Koch. A fully abstract model for graph-interpreted temporal logic. In H. Ehrig, G. Engels, H.J. Kreowski, and G. Rozenberg, editors, *Proceedings of TAGT'98*, volume 1764 of *LNCS*, pages 310–322. Springer Verlag, 2000.
13. R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Proceedings of FASE'98*, volume 1382 of *LNCS*, pages 138–153. Springer Verlag, 1998.
14. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
15. B. König. A general framework for types in graph rewriting. In *Proc. of FST TCS 2000*, volume 1974 of *LNCS*, pages 373–384. Springer Verlag, 2000.
16. R. Langerak. *Transformation and Semantics for LOTOS.* PhD thesis, Department of Computer Science, University of Twente, 1992.
17. K.L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.
18. G.M. Pinna and A. Poigné. On the nature of events: another perspective in concurrency. *Theoretical Computer Science*, 138(2):425–454, 1995.
19. M. Pistore. *History Dependent Automata.* PhD thesis, Department of Computer Science, University of Pisa, 1999.
20. A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. Lo Presti, editors, *Proceedings of the* 3rd *Workshop on Automated Verification of Critical Systems*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

21. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars.* PhD thesis, Technische Universität Berlin, 1996.
22. D. Varró. Towards symbolic analysis of visual modelling languages. In P. Bottoni and M. Minas, editors, *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 of *Electronic Notes in Computer Science*, pages 57–70. Elsevier, 2002.
23. W. Vogler. Efficiency of asynchronous systems and read arcs in Petri nets. In *Proceedings of ICALP'97*, volume 1256 of *LNCS*, pages 538–548. Springer Verlag, 1997.
24. W. Vogler, A. Semenov, and A. Yakovlev. Unfolding and finite prefix for nets with read arcs. In *Proceedings of CONCUR'98*, volume 1466 of *LNCS*, pages 501–516. Springer-Verlag, 1998.

A Unfolding construction for graph grammars

A basic ingredient of the unfolding is the *gluing* operation. It can be seen as a "partial application" of a production at a given match, in the sense that it generates the new items as specified by the production (i.e., items of right-hand side not in the left-hand side), but items that should have been deleted are not affected: intuitively, this is because such items may still be used by another production in the nondeterministic unfolding.

Definition 20 (gluing). *Let $q : L_q \to R_q$ be a T-typed production, G a T-typed graph and $m : L_q \to G$ a graph morphism. We define, for any symbol $*$, the* gluing of G and R_q, according to m and marked by $*$, *denoted by $glue_*(q, m, G)$, as the graph with untyped component $\langle V, E, c\rangle$, where:*

$$V = V_{|G|} \cup m_*(V_{|R_q|}) \qquad\qquad E = E_{|G|} \cup m_*(E_{|R_q|})$$

with m_* defined by:

$$m_*(x) = \begin{cases} m(x) \text{ if } x \in |L_q| \cap |R_q|; \\ \langle x, * \rangle \text{ otherwise.} \end{cases}$$

The connection function and the typing are inherited from G and R_q.

Note that the graph R_q is glued to G, by keeping unchanged the identity of the items already in G and recording in each newly added item from R_q the given symbol $*$.

The unfolding of a graph grammar is obtained as the limit of a chain of occurrence grammars, each approximating the unfolding up to a certain causal depth. Roughly, the causal depth of a production q is the length of the longest chain of productions $q_1 < \ldots q_n < q$. For any graph item x in T, if x is in the start graph it has causal depth 0, otherwise it has the same causal depth as the (unique) production q such that $x \in q^\bullet$.

The unfolding construction here is slightly simplified with respect to the original one in [2], since rules never delete nodes and thus one can avoid to check the dangling condition. A second change is that all matches which differ only for an automorphism of the left-hand side not affecting the preserved graph are considered indistinguishable. More precisely, given a production q, we say that two matches $m, m' : L_q \to G$ are equivalent, if there is automorphism $\alpha : L_q \to L_q$ such that $\alpha_{|L_q \cap R_q}$ is the identity and $m \circ \alpha = m'$. The idea is that two equivalent matches contribute to producing new reachable graphs exactly in the same way and thus only one of them can be considered. Formally, considering only non-equivalent matches we do not affect the completeness of the full unfolding.

Definition 21 (unfolding). *Let $\mathcal{G} = \langle T, G_s, P, \pi\rangle$ be a graph grammar. We inductively define, for each n, an occurrence grammar $\mathcal{U}(\mathcal{G})^{[n]} = \langle T^{[n]}, P^{[n]}, \pi^{[n]}\rangle$ and a pair of mappings $\chi^{[n]} = \langle \chi_T{}^{[n]} : T^{[n]} \to T, \chi_P{}^{[n]} : P^{[n]} \to P\rangle$. Then the* unfolding $\mathcal{U}(\mathcal{G})$ *and the* folding morphism $\chi_{\mathcal{G}} : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$ *are defined as the componentwise union of $\mathcal{U}(\mathcal{G})^{[n]}$ and $\chi^{[n]}$, respectively.*

($\mathbf{n = 0}$) *The components of the grammar* $\mathcal{U}(\mathcal{G})^{[0]}$ *are* $T^{[0]} = |G_s|$, $P^{[0]} = \pi^{[0]} = \emptyset$. *Moreover* $\chi_T{}^{[0]} = type_{G_s}$, $\chi_P{}^{[0]} = \emptyset$.

($\mathbf{n \to n+1}$) *Let* $q \in P$ *be a production in* $\mathcal{G}$ *and* $m : L_q \to \langle T^{[n]}, \chi_T^{R^{[n]}} \rangle$ *be a match satisfying the identification condition, where* $m(|L_q|)$ *is a* concurrent *subgraph of* $T^{[n]}$. *Assume that no match equivalent to* m *has been unfolded yet and that* m *consumes an item of minimal depth among those consumed by matches which have not yet been unfolded. Then* $\mathcal{U}(\mathcal{G})^{[n+1]}$ *is defined by:*

- $P^{[n+1]} = P^{[n]} \cup \{\langle q, m \rangle\}$ *and* $\chi_P{}^{[n+1]} = \chi_P{}^{[n]} \cup \{(\langle q, m \rangle, q)\}$
- *The* T*-typed graph* $\langle T^{[n+1]}, \chi_T^{R^{[n+1]}} \rangle$ *is defined as* $glue_{\langle q,m \rangle}(q, m, \langle T^{[n]}, \chi_T^{R^{[n]}} \rangle)$.
- *The production* $\pi^{[n]}(\langle q, m \rangle)$ *has the same untyped components as* $\pi(q)$. *The typing of the left-hand side is determined by* m, *and each item* x *of the right-hand side which is not in the left-hand side is typed over the corresponding new item* $\langle x, \langle q, m \rangle \rangle$ *of the graph of types.*

The occurrence grammar $\mathcal{U}(\mathcal{G})^{[n+1]}$ is obtained by extending $\mathcal{U}(\mathcal{G})^{[n]}$ with a possible application of a production q at a match m which is a concurrent subgraph of the graph of types. The new production $\langle q, m \rangle$, which intuitively represents the occurrence of q at match m, includes the match m in its name to record the "history" of the occurrence. The graph of types $T^{[n]}$ is extended by adding to it a copy of each item generated by the application q, marked by $\langle q, m \rangle$ (in order to keep trace of the history), as formally expressed via the gluing operation. The morphism $\chi^{[n]}$ is extended consequently.

B Proofs of the completeness and finiteness of the truncation

We first need a preliminary lemma.

Lemma 1 (strong cut-off elimination). *For any given configuration* $C \in Conf(\mathcal{U}(\mathcal{G}))$ *there exists a configuration* C' *without strong cut-offs such that* $Reach(C) \simeq Reach(C')$.

Proof. We show that if $q \in C$ is a strong cut-off then we can obtain a configuration C' such that $Reach(C) \simeq Reach(C')$ and $|C'| < |C|$.

In fact, let $q \in C$ be a strong cut-off. Therefore there exists a production q' in the unfolding and $C_{q'} \in Hist(q')$ such that

$$Reach(C[\![q]\!]) \simeq Reach(C_{q'}) \quad \text{and} \quad |C_{q'}| < |C[\![q]\!]|. \tag{1}$$

We show by induction on $k = |C| - |C[\![q]\!]|$ that we can find a configuration C', with $C_{q'} \sqsubseteq C'$, such that $Reach(C) \simeq Reach(C')$ and $|C'| - |C_{q'}| = |C| - |C[\![q]\!]|$, thus, by (1), $|C'| < |C|$.

($k = 0$) Obvious, since $C = C[\![q]\!]$ one can just choose $C' = C_{q'}$.

$(k \to k+1)$ In this case $C \setminus C[\![q]\!] \neq \emptyset$. Let $q_1 \in C \setminus C[\![q]\!]$, maximal w.r.t. $(\nearrow_C)^*$. Therefore $C_1 = C \setminus \{q_1\}$ is a configuration and $C_1[\![q]\!] = C[\![q]\!]$, by the choice of q_1. Thus by inductive hypothesis there exists a configuration C_1' s.t. $C_{q'} \sqsubseteq C_1'$ and

$$Reach(C_1) \simeq Reach(C_1') \quad \text{and} \quad |C_1'| - |C_{q'}| = |C_1| - |C_1[\![q]\!]|.$$

Since $Reach(C_1') \simeq Reach(C_1)$, the production $\chi_P(q_1)$, which was executable in $Reach(C_1)$, is still executable in $Reach(C_1')$ and thus C_1' can be extended with a production q_1' in such a way that $C' = C_1' \cup \{q_1'\}$ satisfies all the requirements. □

Theorem 2 (completeness of the truncation) *The truncation $\mathcal{T}(\mathcal{G})$ is a complete prefix of the unfolding, i.e., for any reachable graph G of $\mathcal{G}$ there is a configuration C in $Conf(\mathcal{T}(\mathcal{G}))$ such that $Reach(C) \ddot{\simeq} G$.*

Proof. We know, by the "completeness" of the (full) unfolding (Theorem 1) that there exists a finite configuration $C \in Conf(\mathcal{U}(\mathcal{G}))$ such that $Reach(C) \ddot{\simeq} G$. By Lemma 1, there exists a finite configuration C' in $Conf(\mathcal{U}(\mathcal{G}))$ such that $Reach(C') \simeq Reach(C)$, which does not contain strong cut-offs. Such configuration must belong to $Conf(\mathcal{T}(\mathcal{G}))$, otherwise we could construct a strong cut-off-free prefix of the unfolding greater than $\mathcal{T}(\mathcal{G})$. □

Theorem 3 (finiteness of the truncation) *Let $\mathcal{G}$ be a finite graph grammar. If $\mathcal{G}$ is n-bounded then the truncation $\mathcal{T}(\mathcal{G})$ is finite.*

Proof. Take $\mathcal{T}(\mathcal{G})$. By definition of strong cut-off, being $\mathcal{T}(\mathcal{G})$ strong cut-off free, for any production q in $\mathcal{T}(\mathcal{G})$ we can find a local configuration (history) $C_q \in Hist(q)$ such that

$$\text{for any } q' \text{ in } \mathcal{U}(\mathcal{G}), \text{ for any } C_{q'} \in Hist(q'), \text{ if } Reach(C_q) \simeq Reach(C_{q'}) \text{ then } |C_{q'}| \geq |C_q|.$$

Let $G(T,n)$ be the set of n-bounded graphs typed over T, let P be the set of productions in the truncation and consider the function $\tau : P \to G(T,n)$, defined by $\tau(q) = Reach(C_q)$. By the condition above, it is easy to see that $\tau(q_1) = \tau(q_2)$ implies $|C_{q_1}| = |C_{q_2}|$. Since given a production q of the unfolding clearly for any $C \in Hist(q)$, we have $|C| \geq depth(q)$, one concludes that for any $G \in G(T,n)$, $\tau^{-1}(G)$ is finite, since the number of productions in the unfolding up to a fixed causal depth is finite. Therefore $\mathcal{T}(\mathcal{G})$ is finite. □