McMillan Unfolding for Graph Transformation Systems

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, University of 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 a generalisation of the McMillan's complete prefix approach 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 for concurrent and distributed systems [11], generalising Petri nets. Along the years their concurrent behaviour has been deeply studied and a consolidated theory of concurrency is now available. In particular, several semantics of Petri nets, like process and unfolding semantics, have been extended to GTSs (see, e.g., [7, 22, 2, 3]). Building on this semantical foundations, recently some efforts have been devoted to the development of frameworks where behavioural properties of GTSs can be expressed and verified (see [13, 16, 14, 23, 21, 1]).

A powerful verification technique for systems modelled as Petri nets is the so-called *complete prefix approach*, originally introduced by McMillan [18], and based on the unfolding semantics of Petri nets. The unfolding construction produces a static structure which fully describes the concurrent behaviour of the system, including all possible steps and their mutual dependencies, as well as all reachable states. The unfolding is infinite for any non-trivial net, and cannot be used directly for model-checking purposes. However, when the system is finite-state, it is possible to identify a finite initial fragment of the unfolding, called *finite complete prefix* [18, 12], which provides an exact approximation of the system as far as reachability properties are concerned. To construct the finite prefix the idea consists of unfolding the given net, avoiding to insert transitions which do not contribute to generate new markings. More precisely the construction stops at the so-called *cut-off* events, which are transitions which produce the same marking as other transitions but with a larger causal history (i.e., in a bigger number of steps). The complete finite prefix provides a "compact" representation of the state space of a system, which avoids the combinatorial explosion arising when one explores all possible interleavings of events. Finite prefixes have been used for the verification of properties like absence of deadlocks or hazard-freedom in logic circuits [18, 25] and for the model checking of a simple but quite powerful branching temporal logic over Petri nets [12], where net properties like reachability, mutual exclusion, concurrency, liveness, cyclicity can be expressed.

In this paper we concentrate on *finite-state graph transformation systems* and we show how the complete prefix approach can be extended and fruitfully applied to this setting. Roughly, a *graph grammar* consists of a start graph and of a set of productions of the kind $q : L \to R$ specifying that once an occurrence of the left-hand side graph L is found in a graph G, then G can be rewritten replacing L with the right-hand side graph R. Say that a graph grammar $\mathcal{G}$ is *finite-state* if the number of graphs which can be reached from the start graph, considered *up to isomorphism* and *up to isolated nodes*, is finite. Then given any such graph grammar $\mathcal{G}$ we show that there exists a prefix of its unfolding which is *finite* and *complete*, i.e., which provides an exact approximation of the behaviour of $\mathcal{G}$ as far as reachable states are concerned.

Notably, in the case of graph transformation systems, the possibility of performing "contextual" rewritings, i.e., of preserving part of the state in a rewriting step, leads to a form of asymmetric conflict (or weak causality) between events, a phenomenon widely studied, e.g., in the setting of contextual nets [19, 24, 4]. Hence, differently from what happens for (ordinary) Petri nets, in graph grammar computations an event does not have a uniquely determined causal history (consisting of the set of its causes), but it may have instead a set of different local histories [4, 3]. The definition of cut-off event must be adapted consequently: an event is called a *generalised cut-off* if for *any* of its histories there exists another event with a smaller local history, producing the same final state.

Exploiting this definition of generalised cut-off, we are able to prove the *existence* of a finite complete prefix for any finite-state graph grammar. Unfortunately the proof is non-constructive in general. Therefore it is still an open problem to devise an algorithm for constructing such a finite complete prefix, the main difficulty being the non-monotonicity of the notion of generalised cut-off: an event which looks like a cut-off in a finite prefix of the unfolding may not be a cut-off in a larger prefix (because subsequent unfolding steps may add new local histories for the event itself). However, inspired by the work on contextual nets of [25], we can identify a significant subclass of graph grammars, called *read-persistent* graph grammars, where an easy adaptation of the existing algorithms for Petri nets is feasible.

In the second part of the paper we consider a logic $\mathcal{L}2$ where some 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 [9].) Then we show how formulae of such logic can be "translated" into formulae of a logic on Petri net markings which can then be verified on a net extracted from the prefix of the unfolding.

A technical, but important remark, is that in our setting each graph is considered "up-to isolated nodes", i.e., two graphs differing only for isolated nodes are considered equivalent. On the one hand, in this view, our restriction to graph productions which do not delete nodes becomes quite mild, since the deletion of a node can be simulated by leaving the node isolated. On the other hand, in a finite state GTS 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 like the π-calculus or the ambient calculus, where a process can generate an unbounded number of names, while being essentially finite-state since a finite number of them is actually used at each time. Existing model-checking tools, such as SPIN [15], 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 lead, for instance, to the introduction of so-called HD-automata [20].

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

2 Unfolding semantics of graph grammars

2.1 Graph Transformation Systems

This section briefly reviews typed graph grammars [7], a variation of classical double-pushout (DPO) graph grammars [10, 8] where the rewriting takes place on so-called *typed graphs*, namely graphs labelled over a structure (the *graph of types*) that is itself a graph.

In the following, given a set A we denote by A^* the set of finite strings of elements of A (i.e., the free monoid over A). Given $u \in A^*$ we write $|u|$ to indicate the length of u. If $u = a_0 \dots a_n$ and $0 \le i \le n$, 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.

Definition 1 (hypergraph). *A* hypergraph 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. Let* G, G' *be hypergraphs. A* hypergraph morphism $\phi : G \to G'$ *consists of 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* ${\phi_V}^*(c_G(e)) = c_{G'}(\phi_E(e))$*.*

Hereafter, "graph" is used as a short for "hypergraph". Moreover, when clear from the context, the subscripts V and E of the components of hypergraph morphisms are omitted.

Definition 2 (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 morphisms* $f : |G_1| \to |G_2|$ *consistent with the typing, i.e., such that* $type_{G_1} = type_{G_2} \circ f$*.*

A typed graph G is called *injective* if the typing morphism $type_G$ is injective. More generally, for a fixed $n \in \mathbb{N}$, the graph is called *n-injective* if for any item x in T, $|type_G^{-1}(x)| \le n$, namely if the number of instances of "resources" of any type x is bounded by n.

Definition 3 (isomorphism up to isolated nodes). *Let* G *and* G' *be two graphs. We will write* $G \simeq G'$ *to mean that* G *and* G' *are* isomorphic*, and* $G \ddot{\simeq} G'$ *to mean that* G *and* G' *are* isomorphic up to isolated nodes*.*

We use productions as in the double-pushout approach [10], with some restrictions.

Definition 4 (production). *Fixed a graph of types* T*, a* graph production *is a span of* injective T*-typed hypergraph morphisms* $q = (L_q \overset{l_q}{\hookleftarrow} K_q \overset{r_q}{\hookrightarrow} R_q)$*, where* L_q*,* K_q*,* R_q *are finite hypergraphs.*

To simplify the notation, without loss of generality, in the following we will assume that for any production $q = (L_q \overset{l_q}{\hookleftarrow} K_q \overset{r_q}{\hookrightarrow} R_q)$, the morphisms l_q and r_q are (set-theoretical) inclusions and that $|K_q| = |L_q| \cap |R_q|$ (componentwise). Moreover, unless stated otherwise, all the graphs are assumed to be typed over a fixed graph of types T.

Definition 5 (hypergraph rewriting). *Let* $q = (L_q \hookleftarrow K_q \hookrightarrow R_q)$ *be a production. A* match *of* q *in a graph* G *is a morphism* $\phi : L_q \to G$*, satisfying the* identification condition*, i.e., for any* $e, e' \in E_{|L_q|}$*, if* $\phi(e) = \phi(e')$ *then* $e, e' \in E_{|K_q|}$*. In this case we write* $G \Rightarrow_{q,\phi} H$ *or simply* $G \Rightarrow_q H$*, if there exists a double-pushout diagram*

$$\begin{tikzcd} L_q \arrow[d, "\phi"'] & K_q \arrow[l, hook', "l_q"'] \arrow[r, hook, "r_q"] \arrow[d] & R_q \arrow[d] \\ G & D \arrow[l, hook'] \arrow[r, hook] & H \end{tikzcd}$$

Intuitively, the target graph H of the rewriting step is obtained from G by removing the image of the items in $L - K$ and adding the items in $R - K$. The image of the items in K are, in a sense, preserved by the rewriting step.

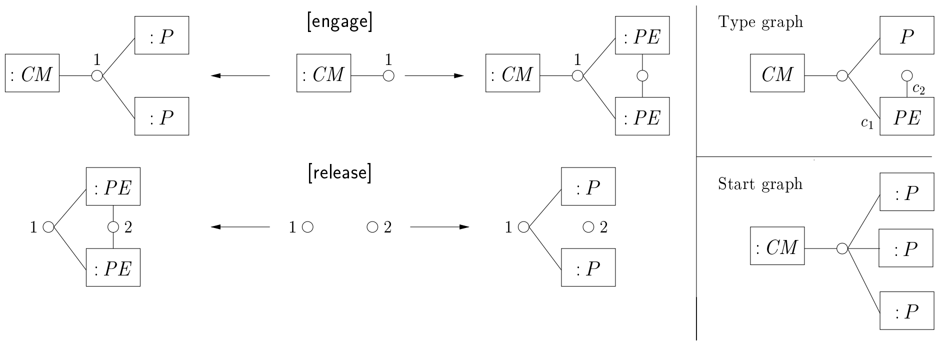

Fig. 1. The finite-state graph grammar $\mathcal{CP}$.

Definition 6 (simple productions). *A production* $q = (L_q \hookleftarrow K_q \hookrightarrow R_q)$ *is called* simple *if (i)* $V_{|L_q|} = V_{|K_q|}$*, (ii)* $|L_q|$ *does not contain isolated nodes, (iii) each isolated node in* $|R_q|$ *belongs to* $|K_q|$*, (iv)* $|K_q|$ *is* properly *included both in* $|L_q|$ *and in* $|K_q|$.

Hereafter we will limit our attention to simple productions. The motivation for Conditions (i)–(iv) above is the following: Condition (i) ensures that productions cannot delete nodes. Deletion of nodes could be managed at the price of introducing heavy technical machineries, which is, in our view, unnecessary, since the deletion of a node can be simulated by leaving that node isolated. Indeed, conditions (ii) and (iii) essentially state that we are interested only in rewriting up to isolated nodes. By (iii) no node is isolated when created and by (ii) nodes that become isolated have no influence on further reductions and hence one can safely assume that they are removed by some kind of garbage collection. Finally Condition (iv) makes sure that every production consumes and produces at least one edge: a requirement corresponding to T-restrictness in the theory of Petri nets.

Definition 7 (graph transformation system and graph grammar). *A* (hyper)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* $\pi(q) = L_q \hookleftarrow K_q \hookrightarrow R_q$*. 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 a graph G is *reachable* in $\mathcal{G}$ if $G_s \Rightarrow^*_{\mathcal{G}} G$, where $\Rightarrow^*_{\mathcal{G}}$ denotes the transitive closure of the rewriting relation induced by the productions of $\mathcal{G}$.

As an example we 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. 1). 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. 1: 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 marking each edge $e \in E_{|G|}$ with "$e : type_G(e)$", or simply by "$: type_G(e)$" if the identity is not relevant (as in the start graph and in all graphs of the productions of Fig. 1). Graph grammar $\mathcal{CP}$ is clearly finite-state, but only up to isolated nodes.

2.2 Nondeterministic Occurrence Grammars

Analogously to what happens for Petri nets, occurrence grammars are "safe" grammars, where the dependency relations among productions satisfy some acyclicity and well-foundedness requirements. Nondeterministic occurrence grammars will be used to provide a static description of the computations of a given grammar, 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.

The notion of safe graph grammar [7] 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-safety, which amounts to the property of being finite-state (up to isomorphism and up to isolated nodes).

Definition 8 (safe grammar). *For a fixed $n \in \mathbb{N}$, we say that a graph grammar $\mathcal{G}$ is* n-safe *if for all graphs H reachable in $\mathcal{G}$ there is an n-injective graph H' such that $H' \ddot{\simeq} H$. A 1-safe grammar will be called simply* safe*.*

The definition can be understood by thinking of nodes and edges of the graph of types as a generalisation of places in Petri nets. In this view the number of different items of a graph which are typed on a given item of the graph of types corresponds to the number of tokens contained in a place.

Since in a safe grammar each graph G reachable from the start graph is injectively typed, we can identify it with the corresponding subgraph $type_G(|G|)$ of the graph of types. With this identification, a production can only be applied to the subgraph of the graph of types which is the image via the typing morphism of its left-hand side. Thus, according to its typing, we can safely think that a production *produces*, *preserves* or *consumes* items of the graph of types. Furthermore, by conditions (ii) and (iii) of Definition 6, for any production q, the graph $L_q \cup R_q$ has no isolated nodes, and thus the image of such a graph in the graph of types T is completely determined by the image of the edges. Hence, we may identify each production of a safe grammar with the sets of edges (of the graph of types) which it consumes, preserves and produces, without loss of information.

Using a net-like language, we speak of *pre-set* ${}^\bullet q$, *context* $\underline{q}$ and *post-set* $q^\bullet$ of a production q, defined as follows. Although these notions are meaningful only for safe grammars, it is technically convenient to define them for general grammars. The same holds for the causal relation and asymmetric conflict relation introduced below.

Definition 9. *For any production q of a graph grammar $\mathcal{G} = \langle T, G_s, P, \pi \rangle$, we define*

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

Similarly, 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 ${}^\bullet v = \{q \in P : v \in type_{R_q}(V_{|R_q|} - V_{|K_q|})\}$.

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

Any subgraph of T without isolated nodes can be identified by the set of its edges.

Definition 10. *Given a graph of types T and a subset of edges $X \subseteq E_T$, we denote by $graph(X)$ the least subgraph of T (actually the unique subgraph, up to isolated nodes) having X as set of edges.*

Observe that, according to the definition, for any production q of a safe grammar we have $type_{L_q}(|L_q|) \ddot{\simeq} graph({}^\bullet q)$, $type_{K_q}(|K_q|) \ddot{\simeq} graph(\underline{q})$, $type_{R_q}(|R_q|) \ddot{\simeq} graph(q^\bullet)$.

Definition 11 (causal relation). *The* causal relation *of a 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* $\lfloor x \rfloor = \{q \in P : q \leq x\}$.

Notice 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 [4, 19, 17, 25]): The application of q' prevents q from being applied, so that q can never follow q' in a derivation (or, equivalently, when both q and q' occur in a derivation then q must precede q'). But the converse is not true, since q *can* be applied before q'.

Definition 12 (asymmetric conflict). *The* asymmetric conflict relation *of a grammar* $\mathcal{G}$ *is the binary relation* $\nearrow$ *over the set of productions* P, *defined by:*

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

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. 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 13 ((nondeterministic) occurrence grammar). *A* (nondeterministic) occurrence grammar *is a grammar* $\mathcal{O} = \langle T, G_s, P, \pi \rangle$ *such that*

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 graph generated by the set* $Min(\mathcal{O})$ *of minimal elements of* $\langle Elem(\mathcal{O}), \leq \rangle$, *typed by the inclusion, i.e.,* $|G_s| = graph(Min(\mathcal{O}))$.[1]
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 part"* $|L_q| - |K_q|$, *and* $type_{R_q}$ *is injective on the "produced part"* $|R_q| - |K_q|$.

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

One can show that, by the defining conditions, each occurrence grammar is safe.

Intuitively, conditions (1)–(3) recast in the framework of graph grammars the analogous conditions of occurrence nets (actually of occurrence contextual nets [4]). 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. 2.

[1] Notice that $Min(\mathcal{O}) \subseteq E_T$, i.e., it does not contain productions by condition (iv) of Definition 6.

2.3 Concurrent Subgraphs, Configurations and Histories

The finite computations of an occurrence grammar are characterised by special subsets of productions where all causal dependencies are satisfied and there are no conflicts.

Definition 14 (configuration). *Let $\mathcal{O}$ be an occurrence grammar. A configuration of $\mathcal{O}$ is a finite subset of productions $C \subseteq P_{\mathcal{O}}$ 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 $C_2 - C_1$ to be applied.

Definition 15 (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. *$C_1 \subseteq C_2$;*
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. *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 $\nearrow$.

Observe that, 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 16. *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., $\{C[\![q]\!] : C \in Conf(\mathcal{O})\}$.*

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

Definition 17 (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 DPO approach to graph rewriting has been proposed in [3]: the construction sketched here is slightly simpler because the productions cannot delete nodes and thus there is no need of taking into account the dangling condition (see also the work on the single-pushout approach to graph rewriting [5]). A more detailed description of the construction can be found in Appendix A.

Intuitively, the construction starts 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 18 (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 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 $m(|L_q|)$* concurrent*.*

Then occurrence grammar $\mathcal{U}(\mathcal{G})^{[n+1]}$ is obtained by extending $\mathcal{U}(\mathcal{G})^{[n]}$ with the application of q to the match m. The new production $q' = \langle q, m\rangle$, which intuitively represents the occurrence of q at match m, includes the match m in the 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, without deleting any item.

As for Petri nets the unfolding is mapped over the original grammar by the so-called folding morphism $\chi : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$. This consists of a pair of mappings, $\chi_T : T' \to T$ mapping each graph item in the (graph of types of) the unfolding to the correspoding 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 $\mathcal{G}$.

The occurrence grammar in Fig. 2 is an initial part of the unfolding of the grammar $\mathcal{CP}$ in Fig. 1. For instance, production engage1 is an occurrence of production engage in $\mathcal{CP}$, applied at the match consisting of the edges $:CM$, $1{:}P$, $2{:}P$. Unfolding such match, two new graph items $1{:}PE$, $2{:}PE$ 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 $T_{\mathcal{CP}}$ (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 $\mathcal{G}$ be a graph grammar. Then a 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 show here that given a graph grammar which is finite-state (up to isomorphism and up to isolated nodes) there is a finite fragment of its unfolding which is complete, i.e., which represents all and only the reachable states.

Let us fix a graph grammar $\mathcal{G} = \langle T, G_s, P, \pi\rangle$, 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 18. 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 $\mathit{Reach}(C)$ the same graph, seen as a graph typed over T by the restriction of the folding morphism, i.e., $\mathit{Reach}(C) = \langle \mathbf{G}(C), \chi_{T|\mathbf{G}(C)} \rangle$.

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

Definition 19 (cut-off). *A production $q \in P'$ of the unfolding $\mathcal{U}(\mathcal{G})$ is a* (generalised) cut-off *if for all $C_q \in \mathit{Hist}(q)$ there exists a $q' \in P'$ and $C_{q'}$ in $\mathit{Hist}(q')$ such that*

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

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

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

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

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 $\mathit{Conf}(\mathcal{T}(\mathcal{G}))$ such that $\mathit{Reach}(C) \ddot{\simeq} G$.*

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

Theorem 3 (finiteness). *If grammar $\mathcal{G}$ is n-safe, then the truncation $\mathcal{T}(\mathcal{G})$ is finite.*

Unfortunately, the proof of the existence of a finite complete prefix for finite-state grammars does not suggest immediately a way of constructing the prefix itself. While leaving this 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 with the notion of generalised cut-off is that it is not "monotone", i.e., an event which is a cut-off in some prefix of the unfolding, could cease to be a cut-off in a larger one. More precisely, consider a production q which satisfies the defining properties of a cut-off in a given prefix of the unfolding, as given in Definition 19. Then the prefix may 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. Observe that the naive idea of generating a production q in the prefix only after all the productions q' such that $q' \nearrow q$ have been inserted does not work since the latter class of productions might be infinite.

The mentioned problem disappears whenever the asymmetric conflict becomes inessential, in the sense that it is subsumed by the other relations.

Definition 20 (read-persistence). *An occurrence graph grammar $\mathcal{O}$ 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.*

Although space limitations prevents us to give the details, it can be shown that an adaptation of the algorithm originally proposed in [18] for ordinary nets and extended in [25] to the class of read-persistent contextual nets, works for read-persistent graph grammars. 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 $q' \in M$ and let $M := M - \{q'\}$;
   - If q' is not a cut-off[2], then unfold q', adding it to the prefix (see Definition 18);
   - 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 interface graph K_q is discrete. 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. Viewing graph grammars as generalised Petri nets (see, e.g., [6]), this appears as the graph grammar theoretical version of the read-persistence for contextual nets as formulated in [25].

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 transitions, is never consumed. Its truncation is the graph grammar $\mathcal{T}(\mathcal{CP})$ depicted in Fig. 2. 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.

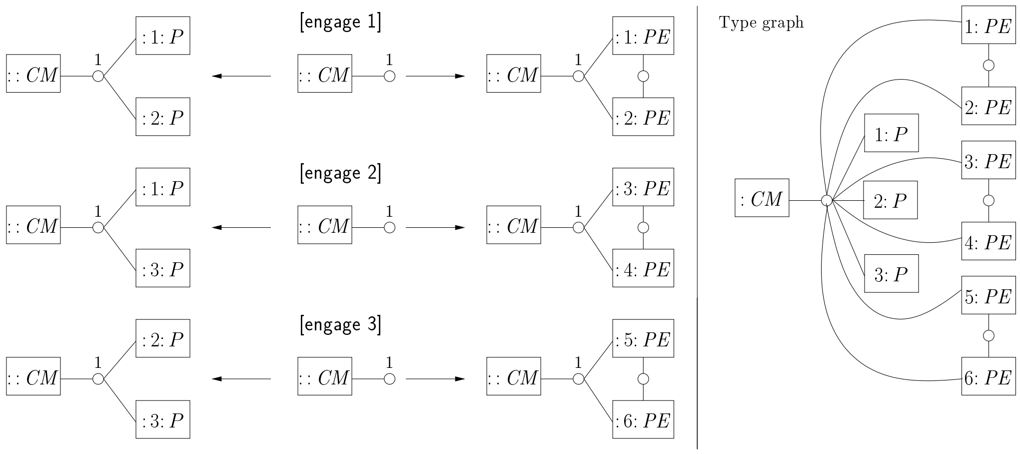

Fig. 2. The truncation $\mathcal{T}(\mathcal{CP})$ of the graph grammar in Fig. 1.

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.

[2] Here one can take a simplified notion of cut-off referring to the least history only, i.e., to the set of causes.

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 such logic can be "translated" into a propositional logic on markings of a Petri net extracted from a finite prefix of the unfolding. Thus 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 the 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 [9]).

Definition 21 (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 $\ell \in \Lambda$, $i, j \in \mathbb{N}$:*

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

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 fixed two 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. Notice that even if arbitrary quantification over nodes is not allowed, it is possible to refer to *non-isolated* nodes using the term $c_i(x)$.

A simple, but fundamental observation is that any occurrence grammar $\mathcal{O}$ can be viewed as occurrence contextual net (i.e., a Petri net with read arcs, see, e.g., [4, 25]) $\mathsf{Net}(\mathcal{O})$ by simply forgetting the graphical nature of the states.

Definition 22 (Petri net underlying a graph 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. 2) is depicted in Fig. 3. 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 a finite complete prefix of the unfolding $\mathcal{T}(\mathcal{G})$. 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

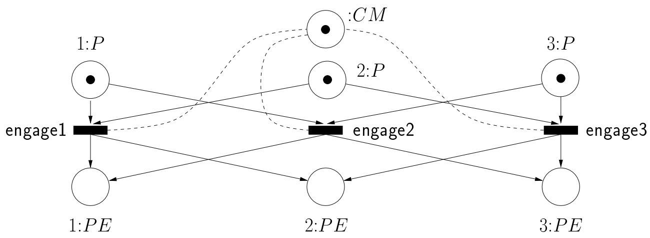

Fig. 3. The Petri net underlying the truncation $\mathcal{T}(\mathcal{CP})$ in Fig. 2

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., $\mathit{graph}(m)$ according to the notation in Definition 10, 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

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

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 23. *Let $\mathcal{T}(\mathcal{G})$ be a complete prefix of a graph grammar $\mathcal{G}$, as above. Let F be graph formula in $\mathcal{L}2$, let $\sigma : \mathit{free}(F) \rightarrow E_{T'}$ and $\Sigma : \mathit{Free}(F) \rightarrow P_{\mathit{fin}}(E_{T'})$. The* encoding *$M$ is defined as follows:*

$$
\begin{aligned}
M[x = y, \sigma, \Sigma] &= \begin{cases} \mathit{true} & \mathit{if}\ \sigma(x) = \sigma(y) \\ \mathit{false} & \mathit{otherwise} \end{cases} \\
M[c_i(x) = c_j(y), \sigma, \Sigma] &= \begin{cases} \mathit{true} & \mathit{if}\ |c_{T'}(\sigma(x))| \geq i \ \wedge\ |c_{T'}(\sigma(y))| \geq j \\ & \qquad \wedge\ [c_{T'}(\sigma(x))]_i = [c_{T'}(\sigma(y))]_j \\ \mathit{false} & \mathit{otherwise} \end{cases} \\
M[\mathit{type}(x) = \ell, \sigma, \Sigma] &= \begin{cases} \mathit{true} & \mathit{if}\ \chi_T(\sigma(x)) = \ell \\ \mathit{false} & \mathit{otherwise} \end{cases} \\
M[F_1 \vee F_2, \sigma, \Sigma] &= M[F_1, \sigma, \Sigma] \vee M[F_2, \sigma, \Sigma] \\
M[F_1 \wedge F_2, \sigma, \Sigma] &= M[F_1, \sigma, \Sigma] \wedge M[F_2, \sigma, \Sigma] \\
M[\neg F, \sigma, \Sigma] &= \neg M[F, \sigma, \Sigma] \\
M[\exists x.F, \sigma, \Sigma] &= \bigvee_{e \in E_{T'}} (e \ \wedge\ M[F, \sigma \cup \{x \mapsto e\}, \Sigma]) \\
M[\forall x.F, \sigma, \Sigma] &= \bigwedge_{e \in E_{T'}} (e \ \rightarrow M[F, \sigma \cup \{x \mapsto e\}, \Sigma] \\
M[\exists X.F, \sigma, \Sigma] &= \bigvee_{E \subseteq_{\mathit{fin}} E_{T'},\ \mathit{co}(E)} \Big(\bigwedge E \ \wedge\ M[F, \sigma, \Sigma \cup \{X \mapsto E\}]\Big) \\
M[\forall X.F, \sigma, \Sigma] &= \bigwedge_{E \subseteq_{\mathit{fin}} E_{T'},\ \mathit{co}(E)} \Big(\bigwedge E \rightarrow M[F, \sigma, \Sigma \cup \{X \mapsto E\}]\Big)
\end{aligned}
$$

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 sugraph 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 set must be covered. Observe that in this case E is limited to range only over concurrent subset of edges. In fact, if E is a non-concurrent set, then any reachable marking m will not include E, i.e., $m \not\models \bigwedge E$.

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

Proposition 2. *Let $\mathcal{T}(\mathcal{G})$ be a complete prefix 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

Consider the simple temporal logic over graph grammars where we can express the fact that a graph property in $\mathcal{L}2$ always / eventually holds in the system, i.e., the logic $\mathbf{T}\mathcal{L}2$ whose formulae are defined as follows:

$$\phi ::= \Box F \mid \Diamond F \mid \neg\phi$$

where F ranges over closed formula of $\mathcal{L}2$. The formulae in $\mathbf{T}\mathcal{L}2$ are interpreted over grammars. Intuitively, given a grammar $\mathcal{G} = \langle T, G_s, P, \pi\rangle$, then

- $\mathcal{G} \models \Diamond F$ if there exists some reachable graph G which satisfies the property F, i.e., if there exists G such that $G_s \Rightarrow^*_{\mathcal{G}} G$ and $G \models F$;
- $\mathcal{G} \models \Box F$ if the property F holds for any reachable graph, i.e., for all G such that $G_s \Rightarrow^*_{\mathcal{G}} G$ it holds that $G \models F$.

Let $\mathcal{G} = \langle G_s, T, P, \pi\rangle$ be a fixed finite-state graph grammar and let ϕ be a $\mathbf{T}\mathcal{L}2$ formula. We next show how we can verify whether $\mathcal{G} \models \phi$.

Let $\mathcal{T}(\mathcal{G}) = \langle T', P', \pi'\rangle$ be a 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 23 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 $S \subseteq_{fin} E_{T'}$ and $\neg co(S)$ then replace the subformula $\bigwedge S$ by false.

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

Observe that, since $\Box F$ and $\neg\Diamond\neg F$ are equivalent, we can always think that our formulae are of the kind $\Diamond F$ or $\neg\Diamond F$ for some F. And since checking whether $\mathcal{G} \models \neg\Diamond F$ amount to check whether $\mathcal{G} \not\models \Diamond F$, we can restrict to the first case only.

A formula in $\mathbf{T}\mathcal{L}2$ of the kind $\Diamond F$ can be verified over $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ as follows:

- Consider the formula over markings $M[F, \emptyset, \emptyset]$;
- Express $M[F, \emptyset, \emptyset]$ in disjunctive normal form, i.e., as a disjunction of conjunctions of atoms (literals or negation of literals)

$$\eta = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{k_i} a_{i,j};$$

  where $a_{i,j}$ can be e or $\neg e$ for $e \in E_{T'}$.

– Apply the Simplification Rule in Proposition 3, as far as possible, thus obtaining a formula η';
– For any conjunct in η' of the kind $e_1 \wedge \ldots \wedge e_h \wedge \neg e'_1 \wedge \ldots \wedge \neg e'_l$ verify if it holds 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., $m_C = \Sigma_{t \in C} t^\bullet - \Sigma_{t \in C} {}^\bullet t$. 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 an extension of C with transitions which consume the e'_j but not the e_i.
– The formula $\Diamond F$ holds if and only if this check succeeds for at least one conjunct.

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 graph to a formula on markings, and, from a practical point of view, 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 the temporal formula $\Box F$, 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.,

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

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

$$\phi \equiv (1{:}\,PE \iff 2{:}\,PE) \wedge (3{:}\,PE \iff 4{:}\,PE) \wedge (5{:}\,PE \iff 6{:}\,PE)$$

and we have to check that the truncation does not satisfy

$$\begin{aligned} \Diamond\neg\phi = \Diamond[(1{:}\,PE \wedge \neg 2{:}\,PE) \vee (\neg 1{:}\,PE \wedge 2{:}\,PE) \vee (3{:}\,PE \wedge \neg 4{:}\,PE) \vee (\neg 3{:}\,PE \wedge 4{:}\,PE) \\ \vee (5{:}\,PE \wedge \neg 6{:}\,PE) \vee (\neg 5{:}\,PE \wedge 6{:}\,PE)], \end{aligned}$$

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. For any graph grammar which is finite-state, up to isomorphism and up to isolated nodes, there exists a finite prefix of the unfolding which provides the same information of the unfolding as far as reachability properties are concerned. A complete finite prefix can be constructed for some classes of graph transformation systems, 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 will be interesting to try to determine an upper bound on the size of the prefix, with respect to the number of reachable graphs. As in the case of Petri nets we expect that a refined cut-off condition will help to decrease the size of the prefix.

We have shown how the complete finite prefix can be used to model-check a simple temporal logic over graph transformation systems. Note that since the definition of cut-off considers graphs up to isomorphism, our technique already incorporates ideas similar to that of symmetry reduction for Petri nets. We plan to generalise the verification technique proposed here to allow the model-checking of more expressive logics where temporal modalities can be arbitraly nested, like the one studied in [12] for Petri nets.

The conceptual relation with HD-automata mentioned in the introduction 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 [13, 14] present a general theory of verification without providing directly applicable techniques. In [16, 1] 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 [23, 21] concentrates on finite-state GTSs. They both generate a suitable labelled transition system out of a given finite-state GTS and then [23] resorts to model-checkers like SPIN, while [21] shows the decidability of 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. Concatenable graph processes: relating processes and derivation traces. In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 283–295. Springer Verlag, 1998.
3. 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.
4. P. Baldan, A. Corradini, and U. Montanari. Contextual Petri nets, asymmetric event structures and processes. *Information and Computation*, 171(1):1–49, 2001.
5. P. Baldan, A. Corradini, U. Montanari, and L. Ribeiro. Coreflective concurrent semantics for single-pushout graph grammars. Proceedings of WADT'02. To appear.
6. A. Corradini. Concurrent graph and term graph rewriting. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR'96*, volume 1119 of *LNCS*, pages 438–464. Springer Verlag, 1996.
7. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.
8. 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.
9. 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.
10. H. Ehrig. Tutorial introduction to the algebraic approach of graph-grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 3–14. Springer Verlag, 1987.
11. 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.
12. J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2–3):151–195, 1994.

13. 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.
14. 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.
15. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
16. 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.
17. R. Langerak. *Transformation and Semantics for LOTOS*. PhD thesis, Department of Computer Science, University of Twente, 1992.
18. K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
19. G.M. Pinna and A. Poigné. On the nature of events: another perspective in concurrency. *Theoretical Computer Science*, 138(2):425–454, 1995.
20. M. Pistore. *History Dependent Automata*. PhD thesis, Department of Computer Science, University of Pisa, 1999.
21. 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.
22. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, Technische Universität Berlin, 1996.
23. 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.
24. 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.
25. 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 to 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 interface), 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 24 (gluing). *Let q be a production, G a 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 $\langle V, E, c\rangle$, where:*

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

with m_* defined by:

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

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 $x \in q^{\bullet}$.

Definition 25 (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, with $m(|L_q|)$* concurrent *subgraph of $T^{[n]}$. Assume that $m(|L_q|)$ has not been unfolded yet and that it has minimal depth among the non-unfolded matches. 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 interface 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 to 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 the 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

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

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

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

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

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

$$\mathit{Reach}(C_1) \simeq \mathit{Reach}(C'_1) \quad \text{and} \quad |C'_1| - |C_{q'}| = |C_1| - |C_1[\![q]\!]|.$$

Since $\mathit{Reach}(C'_1) \simeq \mathit{Reach}(C_1)$, the production $\chi_P(q_1)$, executable in $\mathit{Reach}(C_1)$, is still executable in $\mathit{Reach}(C'_1)$ 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 $\mathit{Conf}(\mathcal{T}(\mathcal{G}))$ such that $\mathit{Reach}(C) \mathrel{\ddot{\simeq}} G$.*

Proof. We know, by the "completeness" of the (full) unfolding (Theorem 1) that surely we can find a finite configuration $C \in \mathit{Conf}(\mathcal{U}(\mathcal{G}))$ such that $\mathit{Reach}(C) \mathrel{\ddot{\simeq}} G$. By Lemma 1, there exists a finite configuration C' in $\mathit{Conf}(\mathcal{U}(\mathcal{G}))$ such that $\mathit{Reach}(C') \simeq \mathit{Reach}(C)$, which does not contain cut-offs. Such configuration must belong to $\mathit{Conf}(\mathcal{T}(\mathcal{G}))$, otherwise we could construct a 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 n-safe graph grammar. Then the truncation $\mathcal{T}(\mathcal{G})$ is finite.*

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

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

Let $G(T,n)$ be the set of n-safe 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 $\tau(q) = \mathit{Reach}(C_q)$. 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 $C \in \mathit{Hist}(q)$, we have $|C| \geq \mathit{depth}(q)$, one concludes that for any $G \in G(T,n)$, $\tau^{-1}(G)$ is finite, since it is entirely contained into a the prefix of the unfolding generated in a finite number of steps. Therefore $\mathcal{T}(\mathcal{G})$ is finite. □