Explicit State Model Checking for Graph Grammars

Arend Rensink

Department of Computer Science, University of Twente, The Netherlands
rensink@cs.utwente.nl

Abstract. In this paper we present the philosophy behind the GROOVE project, in which graph transformation is used as a modelling formalism on top of which a model checking approach to software verification is being built. We describe the basic formalism, the current state of the project, and (current and future) challenges.

1 Introduction

Our primary interest in this paper is software model checking, in particular of object-oriented programs. Model checking has been quite successful as a hardware verification technique and its potential application to software is receiving wide research interest. Indeed, software model checkers are being developed and applied at several research institutes; we mention Bogor [32] and Java Pathfinder [17] as two well-known examples of model checkers for Java.

Despite these developments, we claim that there is an aspect of software that does not occur in this form in hardware, and which is only poorly covered by existing model checking theory: dynamic (de)allocation, both on the heap (due to object creation and garbage collection) and on the stack (due to mutual and recursive method calls and returns). Classical model checking approaches are based on propositional logic with a fixed number of propositions; this does not allow a straightforward representation of systems that may involve variable, possibly unbounded numbers of objects. Although there exist workarounds for this (as evidenced by the fact that, as we have already seen, there are working model checkers for Java) we strongly feel that a better theoretical understanding of the issues involved is needed.

Graphs are an obvious choice for modelling the structures involved, at least informally; direct evidence of this can be found in the fact that any textbook of object-oriented programming uses graphs (of some form) for illustrative purposes. Indeed, a graph model is a very straightforward way to visualise and reason about heap and stack structures, at least when they are of restricted size. In fact, there is no *a priori* reason why this connection cannot be exploited beyond the informal, given the existence of a rich theory of (in particular) graph transformation — see for instance the handbook [33], or the more recent textbook [8]. By adopting graph transformation, one can model the computation steps of object-oriented systems through rules working directly on the graphs, rather than through some intermediate modelling language, such as a process algebra.

This insight has been the inspiration for the GROOVE project and tool.[1] Though the idea is in itself not revolutionary or unique, the approach we have followed differs

[1] GROOVE stands for "GRaphs for Object-Oriented VErification.

P. Degano et al. (Eds.): Montanari Festschrift, LNCS 5065, pp. 114–132, 2008.

from others in the fact that it is based on state space generation directly from graph grammars; hence, neither do we use the input language of an existing model checker to translate the graph rules to, like in [35,12], nor do we attempt to prove properties on the level of graph grammars, like in [22,14,21]. In this paper, we present the elements of this approach, as well as the current state of the research. It is thus essentially the successor [25], where we first outlined the approach.

The paper is structured as follows: in Sect. 2 we introduce the formal notion of graphs and transformations, in a constructive way rather than relying on category theoretical notions. In Sect. 3 we define automata, so-called *graph transition systems*, on top of graph grammars; we also define bisimilarity and show that there exist minimal (reduced) automata. (This is a new result, achieved by abstracting away from symmetries in a somewhat different way than by Montanari and Pistore in [24].) In Sect. 4 we define first-order temporal logic on top of graph transition systems; we also present an equivalent temporal logic based on graph morphisms as core elements, along the lines of [28,6]. Finally, in Sect. 5 we give an evaluation and outlook.

2 Transformation of Simple Graphs

We model system states as graphs. Immediately, we are faced with the choice of graph formalism. In order to make optimal use of the existing theory on graph transformation, in particular the algebraic approach [8], it is preferable to select a definition that gives rise to a (weakly) adhesive HLR category (see [20,10]), such as multi-sorted graphs (with separate sorts for nodes and edges, and explicit source and target functions), or attributed graphs [7] built on top of those. On the other hand, in the GROOVE project and tool [27], in which the approach described in this paper was developed, we have chosen to use simple graphs, which do not fulfill these criteria (unless one also restricts the rules to regular left morphisms, which we have not done), and single-pushout transformation, as first defined by Löwe in [23]. There were two main reasons for this choice:

- In the envisaged domain of application (operational semantics of object-oriented systems) there is little use for edges with identities (see, e.g., [18]);
- The most straightforward connection to first-order logic is to interpret edges as binary predicates (see, e.g., [28]); again, this ignores edge identities.

In this paper, we stick to this choice and present the approach based on simple graphs; in Sect. 5 we will come back to this issue.

Throughout this paper we will assume the existence of a universe of labels Label, and a universe of node identities Node.

Definition 1 (simple graph)

- *A* simple graph *is a tuple* $\langle V, E\rangle$*, where* $V \subseteq \mathsf{Node}$ *is a set of nodes and* $E \subseteq V \times \mathsf{Label} \times V$ *a set of edges. Given* $e = (v, a, w) \in E_G$*, we denote* $src(e) = v$*,* $lab(e) = a$ *and* $tgt(e) = w$ *for its source, label and target, respectively.*

– *Given two simple graphs G, H, a (partial)* graph morphism $f: G \to H$ *is a pair of partial functions* $f_V: V_G \rightharpoonup V_H$ *and* $f_E: E_G \rightharpoonup E_H$*, such that for all* $e \in dom(f_E)$*,* $f_E(e) = (f_V(src(e)), lab(e), f_V(tgt(e)))$.

Some notation and terminology.

– A morphism f is called *total* if f_V and f_E are total functions, *injective* if they are injective functions, and an *isomorphism* if they are bijective functions. The total, injective morphisms are sometimes called *monos*.[2]
– For $f: G \to H$, we call G and H the *source* and *target* of f, denoted $src(f)$ and $tgt(f)$, respectively. A pair of morphisms with a common source is called a *span*; with a common target, a *co-span*.
– We write $G \cong H$ to denote that there is an isomorphism from G to H, and $\phi: G \cong H$ to denote that ϕ is such an isomorphism. This is extended to the individual morphisms of spans $\xleftarrow{f}\xrightarrow{g}$: we write $f \cong g$ for such morphisms if there is an isomorphism $\phi: tgt(f) \to tgt(g)$ such that $g = \phi \circ f$.
– We use Morph to denote the set of all (partial) graph morphisms.

Some example graphs will follow below. As a further convention, in figures we will use node labels to represent self-edges; for instance, in Fig. 1, the labels Buffer, Cell and Object are used in this way.

Graph morphisms are used for many different purposes, but the following uses stand out in the context of graph transformation:

– Isomorphisms are used to capture the fact that two graphs are essentially the same. The whole theory of graph transformation is set up to be insensitive to isomorphism, meaning that it is all right to pick the most convenient isomorphic representative.
– Total morphisms describe an embedding of one graph into another, possibly while merging nodes. If a total morphism is also injective, then there is no merging, and the source graph is sometimes called an (isomorphic) *subgraph* of the target.
– Arbitrary (partial) morphisms are used to capture the *difference* between graphs, in terms of the exact change from the source graph to the target graph. To be precise, the change consists of deletion of those source graph elements on which the morphism is not defined, merging of those elements on which it is not injective, and addition of those target graph elements that are not in the image of the morphism.

A core construction in graph transformation is the so-called *pushout*. This is used as a way to combine, or glue together, different changes to the same graph; or in particular, an embedding on the one hand and a change on the other — where both the embedding and the change are captured by morphisms, as in the second and third items above. In the following, we define pushouts constructively.

Definition 2 (pushout). *For* $i = 1, 2$*, let* $f_i: G \to H_i$ *be morphisms in* Morph *such that* $H_i = \langle V_i, E_i \rangle$ *with* $V_1 \cap V_2 = \emptyset$*; let* $*$ *be arbitrary such that* $* \notin V_1 \cup V_2$.

[2] This name stems from category theory, where monos are arrows satisfying a particular decomposition property. We do not elaborate on this issue here.

1. *For $i = 1, 2$, let $\bar{f}_i \colon G \to \bar{H}_i$ be a total extension of f_i, defined by adding a distinct fresh node v' to H_i and setting $\bar{f}_i(v) = v'$ for each $v \in V_G \setminus dom(f_{V,i})$. Hence, $\bar{H}_i = \langle \bar{V}_i, \bar{E}_i \rangle$ such that $\bar{V}_i$ extends V_i with the fresh nodes, and $\bar{E}_i$ extends E_i with the fresh edges implied by the totality of $\bar{f}_i$.*
2. *Let $\bar{V} = \bar{V}_1 \cup \bar{V}_2$ be the union of the extended node sets, and $\bar{E} = E_1 \cup E_2$ that of the extended edge sets. Let $\simeq \subseteq \bar{V} \times \bar{V}$ be the smallest equivalence relation such that $\bar{f}_{V,1}(v) \simeq \bar{f}_{V,2}(v)$ for all $v \in V_G$; and likewise for edges.*
3. *Let $W = \{X \in \bar{V}/{\simeq} \mid X \subseteq V_1 \cup V_2\}$, and for $i = 1, 2$, define $g_{V,i} \colon V_i \to \bar{W}$ such that for all $v \in V_i$*

$$g_{V,i} \colon v \mapsto [v]_\simeq \quad \textit{if } [v]_\simeq \subseteq V_1 \cup V_2 \ .$$

   Let $F = \{([v]_\simeq, a, [w]_\simeq) \mid [(v, a, w)]_\simeq \subseteq E_1 \cup E_2\}$; moreover, for $i = 1, 2$, define $g_{E,i} \colon E_i \to F$ such that for all $(v, a, w) \in E_i$

$$g_{E,i} \colon (v, a, w) \mapsto ([v]_\simeq, a, [w]_\simeq) \quad \textit{if } [(v, a, w)]_\simeq \subseteq E_1 \cup E_2 \ .$$

4. *Let $K = \langle W, F \rangle$; then $g_i \colon H_i \to K$ are morphisms for $i = 1, 2$.*

K together with the morphisms g_i is called the pushout *of the span f_1, f_2; together they form the following* pushout diagram.

$$\begin{tikzcd} G \arrow[r, "f_1"] \arrow[d, "f_2"'] & H_1 \arrow[d, "g_1"'] \\ H_2 \arrow[r, "g_2"] & K \end{tikzcd}$$

The intuition behind the construction is as follows: first (step 1) we (essentially) construct the disjoint union of the two target graphs H_1 and H_2, augmented with images for those elements of G for which the morphisms f_1 resp. f_2 were originally undefined. These fresh images later work like little "time bombs" that obliterate themselves, together with all associated (i.e., equivalent) elements. In the resulting extended disjoint union, we call two elements equivalent (step 2) if they have a common source in G, under the (extended) morphisms $\bar{f}_1$ or $\bar{f}_2$. Then (step 3), we construct the quotient with respect to this equivalence, omitting however the (equivalence classes of the) fresh nodes and edges added earlier — in other words, this is when the bombs go off.

The name "pushout" for the object constructed in the previous definition is justified by the fact that it satisfies a particular co-limit property, stated formally in the following proposition.

Proposition 1 (pushout property). *Given morphisms f_i as in Def. 2, the pushout is a co-limit of the diagram consisting of the span f_1, f_2, meaning that*

- *$g_1 \circ f_1 = g_2 \circ f_2$;*
- *Given any $h_i \colon H_i \to L$ for $i = 1, 2$ such that $h_1 \circ f_1 = h_2 \circ f_2$, there is a unique morphism $k \colon K \to L$ such that $h_1 = k \circ g_1$ and $h_2 = k \circ g_2$; in other words, such that the following diagram commutes*

$$\begin{tikzcd}
G \arrow[r, "f_1"] \arrow[d, "f_2"'] & H_1 \arrow[d, "g_1"'] \arrow[ddr, bend left, "h_1"] & \\
H_2 \arrow[r, "g_2"] \arrow[drr, bend right, "h_2"'] & K \arrow[dr, "k"] & \\
& & L
\end{tikzcd}$$

Proof (sketch). We construct k. For a given $W \in V_K$, let i be such that $W \cap V_i \neq \emptyset$; let $k_V(W) = h_i(w)$ for $w \in W \cap V_i$. This is well-defined due to the fact that W is a $\simeq$-equivalence class, in combination with the confluence of the f_1g_1/f_2g_2- and f_1k_1/f_2k_2-squares of the diagram. k_E is defined likewise. k satisfies the necessary commutation properties by construction. Its uniqueness in this regard can be established by observing that no other image for any of the nodes or edges of K will make the pushout diagram commute.

The mechanism we use for generating state spaces is based on graph grammars, consisting of a set of graph production rules and a start graph. The necessary ingredients are given by the following definition.

Definition 3 (graph grammar)

- *A graph production rule is a tuple $r = \langle p\colon L \to R, ac\rangle$, where $p \in$* Morph*, and ac is an* application condition *on total graph morphisms $m\colon L \to G$ (for arbitrary G) that is well-defined up to isomorphism of G. We write $m \models ac$ to denote that m satisfies ac. (Well-definedness up to isomorphism of G means that $m \models ac$ if and only if $\phi \circ m \models ac$ for all graph isomorphisms $\phi\colon G \to H$.)*

  A graph grammar is a tuple $\mathcal{G} = \langle \mathcal{R}, I\rangle$, where $\mathcal{R}$ is a set of production rules and I is an initial graph.
- *Given a graph production rule r, an r-*derivation *is a four-tuple (G, r, m, H), typically denoted $G \overset{r,m}{\Longrightarrow} H$, such that $m\colon L_r \to G \models ac_r$ and H is isomorphic to the pushout graph; i.e., the following square is a pushout:*

$$\begin{tikzcd}
L_r \arrow[r, "p_r"] \arrow[d, "m"'] & R_r \arrow[d, "m'"] \\
G \arrow[r, "f"] & K \cong H
\end{tikzcd}$$

  A $\mathcal{G}$-derivation ($\mathcal{G}$ a graph grammar) is an r-derivation for some $r \in \mathcal{R}_{\mathcal{G}}$.

The definition is slightly sloppy in that our pushout construction is only defined if the right hand side R_r and the host graph G have disjoint node sets. This is in practice not a problem because we are free to take isomorphic representatives where required; in particular, we can make sure that the derived graphs have nodes that are distinct from all right hand side graphs.

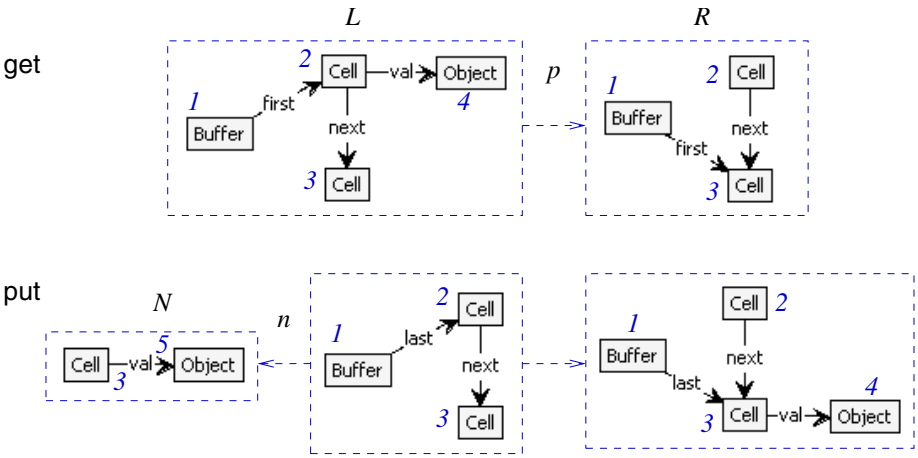

Fig. 1. Graph production rules for a circular buffer

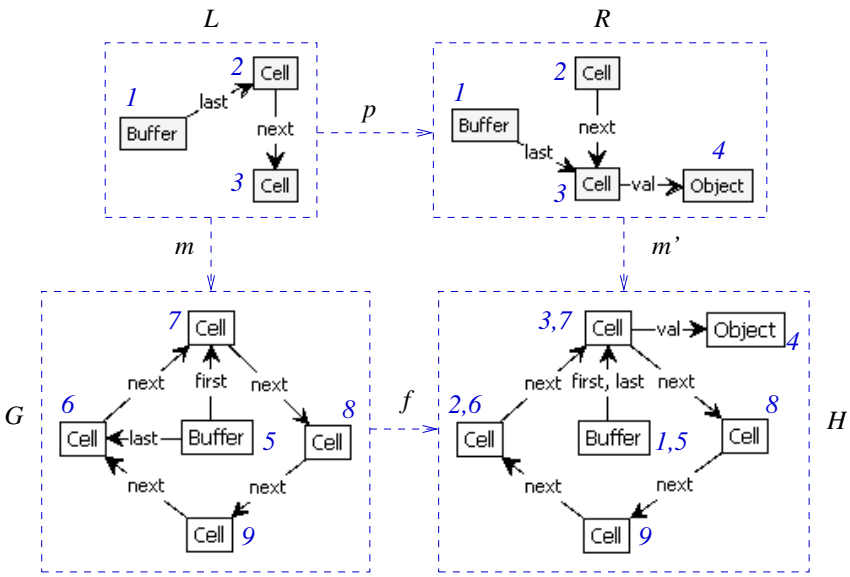

Fig. 2. Example derivation using the rule get from Fig. 1

As a running example, we use the graph grammar consisting of the two rules in Fig. 1, which retrieve, resp. insert, objects in a circular buffer. The rule put has a so-called *negative application condition* (see [13]), in the form of a morphism $n\colon L \to N$ from the left hand side of the rule; the satisfaction of such a condition is defined by

$$m\colon L \to G \models n \quad :\Leftrightarrow \quad \nexists f\colon N \to G : f \circ n = m \restriction dom(n)$$

(where $dom(n)$ is the sub-graph of L on which n is defined, and $m \restriction H$ stands for the restriction of the morphism m to the graph H). The morphisms are indicated by the numbers in the figure: nodes are mapped to equally labelled nodes in the target graph (if such a target node exists, elsewhere the morphism is undefined), and edges are mapped accordingly. An example derivation is shown in Fig. 2, given a match $m = \{(1,5),(2,6),(3,7)\}$. In terms of Def. 2, this gives rise to

$$\bar{V} = \{1,2,3,4,5,6,7,8,9\}$$
$$W = \{\{1,5\},\{2,6\},\{3,7\},\{4\},\{8\},\{9\}\} \ .$$

3 Graph Transition Systems

As related in the introduction above, the core of our approach is explicit state space generation, where the states are essentially graphs. Rather than completely identifying states with graphs (like we did in the original [25]), in this paper we follow [2] by merely requiring that every state *has* an associated graph. This leaves room for cases where there is more information in a state than just the underlying graph.

Definition 4 (graph transition system). *A* graph transition system *(GTS) is a tuple* $S = \langle Q, T, q_0 \rangle$ *where*

- *Q is a set of states with, for every $q \in Q$, an associated graph $G_q \in \mathsf{Graph}$;*
- *$T \subseteq Q \times \mathsf{Morph} \times Q$ is a set of transitions, such that $src(\alpha) = G_q$ and $tgt(\alpha) \cong G_{q'}$ for all $(q, \alpha, q') \in T$. As usual, we write $q \xrightarrow{\alpha} q'$ as equivalent to $(q, \alpha, q') \in T$.*
- *$q_0 \in Q$ is the initial state.*

S is called symmetric *if $(q, \alpha, q') \in T$ implies $(q, \alpha \circ \phi, q') \in T$ for all $\phi \colon G_q \cong G_q$.*

We write q_t, α_t, q'_t for the source state, morphism and target state of a transition t, and q_i etc. for the components of t_i. Note that the target graphs of the morphisms associated with the transitions are only required to be isomorphic, rather than identical, to the graphs associated with their target states. Obviously, this only makes a difference for graphs having non-trivial symmetries, since otherwise the isomorphisms are unique and might as well be appended to the transition morphisms. Using the definition given here, we avoid to distinguish between symmetric cases, and hence it is possible to minimise with respect to bisimilarity — see below.

An example symmetric GTS is shown in Fig. 3. The morphisms associated with the transitions are indicated by node mappings at the arrows; all the morphisms have empty edge mappings. The two left-to-right transitions are essentially the same, since their associated morphisms are "isomorphic;" that is, there is an isomorphism between their target graphs that equalises them —namely, based on node mapping $(3,4),(4,3)$. On the other hand, this is not true for the right-to-left transitions: the node mapping $(1,2),(2,1)$ is not an isomorphism of the left hand side graph. Indeed, by symmetry, the presence of each of the right-to-left transition implies the presence of the other.

We can now understand a GTS as *being generated by* a graph grammar $\mathcal{G}$, if the start state's associated graph is isomorphic to the start graph of $\mathcal{G}$, and there are transitions corresponding to all the derivations of $\mathcal{G}$ (modulo isomorphism). That is, we call a GTS S generated by $\mathcal{G}$ if the following conditions hold:

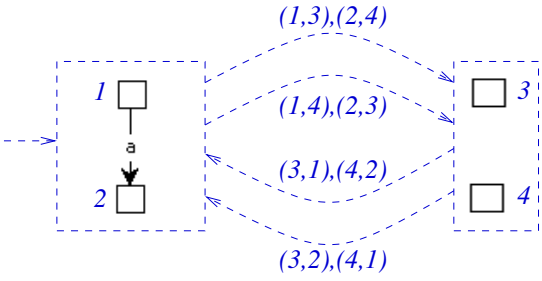

Fig. 3. Example symmetric GTS, removing and re-adding an edge

- G_{q_0} is isomorphic to I;
- For any $q \in Q$ and any $\mathcal{G}$-derivation $G_q \xrightarrow{r,m} H$, S has a transition $q \xrightarrow{\alpha} q'$ such that $\alpha \cong p_r \uparrow m$.
- Likewise, for any transition $q \xrightarrow{\alpha} q'$, there is a $\mathcal{G}$-derivation $G_q \xrightarrow{r,m} H$ such that $\alpha \cong p_r \uparrow m$.

For instance, Fig. 4 shows a GTS generated using the rules in Fig. 1, taking G from Fig. 2 as a start graph.

Grammar-generated GTSs are close to the history-dependent automaton (HDA) of Montanari and Pistore (see [24]). There, states have associated sets of names, which are "published" through labelled transitions, the labels also having names and the transitions carrying triple co-spans of total injective name functions, from the source state, target state and label to a common set of names associated with the transition.

If we limit our rules to injective morphisms, then the derivation morphisms will be injective, too. Injective partial morphisms $\alpha: G \to H$ are in fact equivalent to

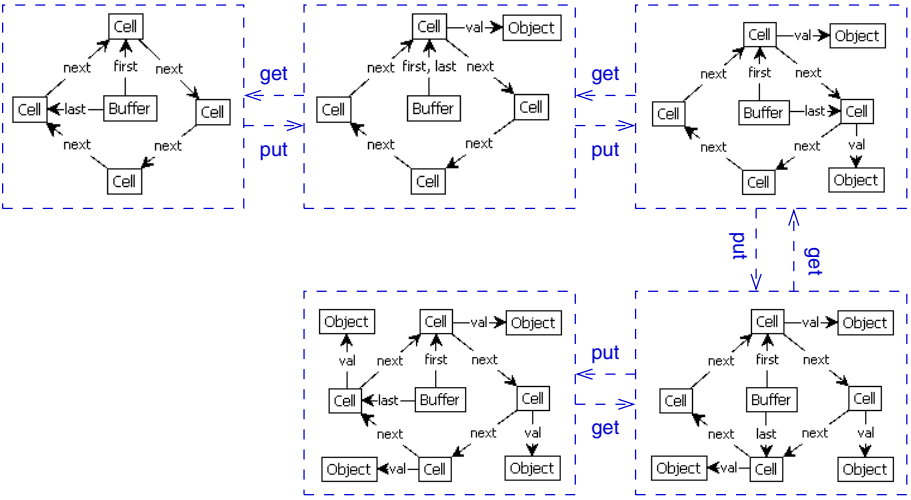

Fig. 4. GTS generated from the circular buffer rules

co-spans of monos $\alpha_L: G \to U, \alpha_R: H \to U$, where U is the "union" of G and H while gluing together the elements mapped onto each other by α. If, furthermore, we require all matchings to be injective as well (through the application condition ac), any $\mathcal{G}$-generated GTS gives rise to a HDA, where the names are given by node identities. The only catch is that, as mentioned before, the actual isomorphism from $tgt(\alpha)$ to $G_{q'}$ in a transition $q \xrightarrow{\alpha} q'$ is not part of the GTS, whereas in HDA the name mappings are precise. On the other hand, this information is abstracted away in *HDA with symmetries* (see [24]). We conclude:

Proposition 2. *If $\mathcal{G}$ is such that all rules in $\mathcal{R}$ are injective, and $m \models ac$ only if m is injective, then every $\mathcal{G}$-generated GTS uniquely gives rise to a HDA with symmetries, in which the transition labels are tuples of rules and matchings.*

In contrast to HDAs, however, GTS transitions are reductions and not reactions. In other words, they do not reflect communications with the "outside world". In fact, the behaviour modelled by a GTS is not primarily captured by the transition labels but by the structure of the states; as we will see, the logic we use to express GTS properties can look inside the states. For that reason, although we can indeed define a notion of bisimilarity —inspired by HDA bisimilarity— which abstracts away to some degree from the branching structure, the relation needs to be very discriminating on states.[3]

Definition 5 (bisimilarity). *Given two GTSs S_1, S_2, a bisimulation between S_1 and S_2 is an isomorphism-indexed relation $(\rho_\phi)_\phi \subseteq (Q_1 \times Q_2) \cup (T_1 \times T_2)$ such that*

- *For all $q_1 \ \rho_\phi \ q_2$, the following hold:*
  - *$\phi: G_{q_1} \cong G_{q_2}$;*
  - *For all $q_1 \xrightarrow{\alpha_1} q_1'$, there is a $q_2 \xrightarrow{\alpha_2} q_2'$ such that $(q_1, \alpha_1, q_1') \ \rho_\phi \ (q_2, \alpha_2, q_2')$;*
  - *For all $q_2 \xrightarrow{\alpha_2} q_2'$, there is a $q_1 \xrightarrow{\alpha_1} q_1'$ such that $(q_1, \alpha_1, q_1') \ \rho_\phi \ (q_2, \alpha_2, q_2')$;*
- *For all $t_1 \ \rho_\phi \ t_2$: $q_1 \ \rho_\phi \ q_2$, $\alpha_1 \cong \alpha_2 \circ \phi$, and there is a ψ such that $q_1' \ \rho_\psi \ q_2'$;*
- *$q_{0,1} \ \rho_\phi \ q_{0,2}$ for some ϕ.*

S_1 and S_2 are said to be bisimilar*, denoted $S_1 \sim S_2$, if there exists a bisimulation between them.*

For instance, although the GTS generated by a graph grammar is not unique, it is unique modulo bisimilarity.

Theorem 1. *If S_1 and S_2 are both generated by a graph grammar $\mathcal{G}$, then $S_1 \sim S_2$.*

Thus, bisimulation establishes binary relations between the states and transitions of two GTSs. As usual, this can be used to *reduce* GTSs, as follows:

- Between any pair of GTSs there exists a *largest* bisimulation, which can be defined as the union of all bisimulations (pointwise along the ϕ). (The proof that this is indeed again a bisimulation is straightforward.)
- If S_1 and S_2 are the same GTS (call it S), then the largest bisimulation gives rise to an equivalence relation ρ over the states and transitions of S.

[3] In this paper, we use bisimilarity only to *minimise* GTSs, not so much to establish a theory of equivalence.

- Given the equivalence ρ, pick a representative from every equivalence class of states Q/ρ. For any $q \in Q$, let $\hat{q}$ denote the representative from $[q]_\rho$.
- Similarly, for every equivalence class of transitions $U \in T/\rho$, pick a representative with as source the (uniquely defined) state $\hat{q}_t$ for $t \in U$. For any t, let $\hat{t}$ denote the selected representative from $[t]_\rho$. (Note that this means $q_{\hat{t}}$ is the representative state selected from $[q_t]_\rho$, but $q'_{\hat{t}}$ is not necessarily the representative for $[q'_t]_\rho$.)
- Define $\hat{S} = \langle \hat{Q}, \hat{T}, \hat{q}_0 \rangle$ where

$$\hat{Q} = \{\hat{q} \mid q \in Q\}$$
$$\hat{T} = \{(q_{\hat{t}}, \alpha_{\hat{t}}, \hat{q}'_t) \mid t \in T\} \ .$$

Thus, this construction collapses states with isomorphic graphs and transitions with isomorphic graph changes. For instance, in Fig. 3, only one of the two left-to-right transitions remains in the reduced transition system. The transition system in Fig. 4 cannot be reduced further (there are no non-trivial isomorphisms); in fact, it has already been reduced up to symmetry, since (for instance) the precise graph reached after applying $\mathsf{put} \cdot \mathsf{get}$ from the initial state is not identical to the start graph; rather, it can be thought of as isomorphically rotated clockwise by 90°. Indeed, reduction with respect to bisimilarity exactly corresponds to symmetry reduction for model checking (see, e.g., [11]).

It may actually not be clear that $\hat{S}$ is well-defined, since it relies on the choice of representatives from the equivalence classes Q/ρ and T/ρ. To show well-definedness we must first define *isomorphism* of GTSs.

Definition 6 (GTS isomorphism). *Two GTSs S_1, S_2 are* isomorphic *if there exists a pair of mappings $\phi_Q \colon Q_1 \to (\mathsf{Morph} \times Q_2)$ and $\phi_T \colon T_1 \to T_2$ such that*

- *For all $q_1 \in Q_1$, $\phi_Q(q_1) = (\psi, q_2)$ such that $\psi \colon G_{q_1} \cong G_{q_2}$;*
- *For all $t_1 \in T_1$, if $\phi_Q(q_1) = (\psi, q_2)$ and $\phi_Q(q'_1) = (\psi', q'_2)$ then $\phi_T(t_1) = (q_2, \alpha_2, q'_2)$ such that $\alpha_2 \circ \psi \cong \alpha_1$;*
- *$\phi_Q(q_{1,0}) = (\psi, q_{2,0})$ for some ψ.*

The following theorem states the crucial properties of the GTS reduction.

Theorem 2. *Given any GTS S, the reduced GTS $\hat{S}$ is well-defined up to isomorphism and satisfies $\hat{S} \sim S$. Furthermore, for any bisimulation $(\hat{\rho}_\phi)_\phi$ between $\hat{S}$ and itself, $\hat{q}_1 \ \hat{\rho}_\phi \ \hat{q}_2$ implies $\hat{q}_1 = \hat{q}_2$ for all $\hat{q}_1, \hat{q}_2 \in \hat{Q}$.*

Proof (sketch).

- It is straightforward to check that any choice of representatives from the ρ-induced equivalence classes of states and transitions gives rise to an isomorphic GTS. For instance, in Fig. 3 it is not important which of the two left-to-right transition is chosen. (This indifference crucially depends on the condition $\alpha_2 \circ \psi \cong \alpha_1$ in the isomorphism condition for transitions; if we would require $\alpha_2 \circ \psi = \psi' \circ \alpha_1$ then the uniqueness up to isomorphism would break down.)
- Let $(\rho_\phi)_\phi$ be the largest bisimulation over S, used in the construction of $\hat{S}$. $\hat{S} \sim S$ is then immediate, using as bisimilarity the restriction of $(\rho_\phi)_\phi$ on the left hand side to states of $\hat{S}$.

– It can be proved that $\hat{q}_1 \ \hat{\rho}_\phi \ \hat{q}_2$ implies that also $\hat{q}_1 \ \rho_\phi \ \hat{q}_2$ in the original GTS S. But then $\hat{q}_1$ and $\hat{q}_2$ are both representatives of the same ρ-equivalence class of states in S, implying they are the same.

(Note that it is *not* the case that $\hat{S}$ has only trivial bisimulations, i.e., such that $q \ \rho_\phi \ q$ implies that ϕ is the identity, since in contrast to [24] we are not abstracting graphs up to symmetry.) The following is immediate.

Corollary 1 (canonical GTS). *Given a graph grammar $\mathcal{G}$, there is a smallest GTS (unique up to isomorphism) generated by $\mathcal{G}$. We call this the* canonical *GTS of $\mathcal{G}$, and denote it $S_\mathcal{G}$.*

The following property of the canonical GTS is a consequence of the definition of derivations and the assumption that ac is well-defined up to isomorphism.

Proposition 3 (symmetry of canonical GTSs). *For any graph grammar $\mathcal{G}$, the canonical GTS $S_\mathcal{G}$ is symmetric.*

4 First-Order Temporal Logic

Besides providing a notion of symmetry, the transition morphisms of GTSs also keep track of the identity of entities. For instance, Fig. 4 contains all the information necessary to check that entities are retrieved in the order they are inserted and that no entity is inserted without eventually being retrieved. All this is established through the node identities of the val-labelled nodes; no data values need be introduced. Such properties can be expressed formally as formulae generated in a special temporal logic.

4.1 First-Order Linear Temporal Logic

The usual temporal logics are *propositional*, meaning that their smallest building blocks are "atomic" propositions, whose truth is a priori known for every state.[4] For expressing properties that trace the evolution of entities over transitions, however, we need variables that exist, and remain bound to the same value, *outside* the temporal modalities. An example of a logic that has this feature is *first-order linear temporal logic* (FOLTL), generated by the following grammar:

$$\Phi ::= x \mid a(x,y) \mid \mathsf{tt} \mid \neg\Phi \mid \Phi \vee \Psi \mid \exists x.\Phi \mid \mathsf{X}\Phi \mid \Phi \mathsf{U} \Psi \ .$$

The meaning of the predicates is:

– x expresses that the first-order variable x has a definite value (which is taken from Node). As we will see, this is not always the case: x may be undefined.
– $a(x,y)$ expresses that there is an a-labelled edge from node x to node y.
– tt (true), $\neg\Phi$ and $\Phi \vee \Psi$ have their standard meaning.

[4] In practice, such propositions may themselves well be (closed) first-order formulae, evaluated over each state.

– $\mathsf{X}\Phi$ and $\Phi \mathsf{U} \Psi$ are the usual linear temporal logic operators: $\mathsf{X}\Phi$ expresses that Φ is true in the next state, whereas $\Phi \mathsf{U} \Psi$ expresses that Ψ is true at some state in the future, and until that time, Φ is true.

In addition, we use the common auxiliary propositional operators $\wedge$, $\Rightarrow$ etc., as well as the temporal operators G (for "Globally") and F (for "in the Future"). Furthermore, we use $\exists x : a.\Phi$ with $a \in \mathsf{Label}$ as abbreviation for $\exists x.a(x,x) \Rightarrow \Phi$. Some example formulae which can be interpreted over the circular buffer system of Fig. 4 are:

1. $\forall c : \mathsf{Cell}.\nexists v.\mathsf{val}(c,v)$ is a non-temporal formula expressing that in the current state there is no val-edge.
2. $\forall c : \mathsf{Cell}.\ \mathsf{F}\ \exists v.\mathsf{val}(c,v)$ is a temporal formula expressing that all currently existing cells will eventually be filled (though maybe not all at the same time).
3. $\mathsf{F}\ \forall c : \mathsf{Cell}.\exists v.\mathsf{val}(c,v)$ expresses that eventually all cells will be filled (at the same time).
4. $\exists o : \mathsf{Object}.\mathsf{X}\ \neg o$ expresses that there exists an Object-node that will be gone in the next state.
5. $\forall b : \mathsf{Buffer}.(\exists c, o.\mathsf{first}(b,c) \wedge \mathsf{val}(c,o))\ \mathsf{U}\ (\nexists c, o.\mathsf{val}(c,o))$ expresses that eventually the buffer is empty, and until that time, the first cell has a value.
6. $(\mathsf{X}\ \exists o.\mathsf{val}(c,o)) \wedge (\nexists o.\mathsf{X}\ \mathsf{val}(c,o))$ expresses that, although in the next state the cell c will have a value, that value does not already exist in the current state. This implies that that value is freshly created in the next state.

Formulae are interpreted over infinite sequences of graph morphisms, in combination with a valuation of the free variables. The definition requires some auxiliary concepts and notation.

– A *path* is an infinite sequence of consecutive morphisms $m_1\, m_2 \cdots$, i.e., such that $tgt(m_i) = src(m_{i+1})$ for all $i \geq 1$. We let σ range over paths. For all $1 \leq i \leq |\sigma|$, σ_i denotes the i'th element of σ (i.e., m_i), and, σ^i the tail of σ starting at the i'th element (i.e., $m_i\, m_{i+1}\, \cdots$).
– A *run* of a GTS S is an infinite sequence of pairs $\rho = (t_1, \phi_1)(t_2, \phi_2) \cdots$ such that $q_1 = q_0$, and for all $i \geq 1$
  - $q'_i = q_{i+1}$;
  - Either $t_i \in T$ or $t_i = (q, id_{G_q}, q)$ where $\nexists t \in T : q_t = q$ (so we stutter upon reaching a final state);
  - ϕ_i: $tgt(\alpha_i) \cong src(\alpha_{i+1})$.

  Given a run ρ, the path σ_ρ generated by ρ is the sequence of morphisms $(\phi_1 \circ \alpha_1)(\phi_2 \circ \alpha_2) \cdots$.
– θ is a partial valuation of variables to elements of Node. If f is a graph morphism, then $f \circ \theta$ is a new valuation defined by concatenating f_V with θ. We use $\theta\{v/x\}$ (with $v \in \mathsf{Node}$) to denote a derived valuation that maps x to v and all other variables to their θ-images.

Satisfaction of a formula is expressed through a predicate of the form $G, \sigma, \theta \models \phi$, where $G = src(\sigma_1)$. The following set of rules defines this predicate inductively. The main point to notice is the modification of the valuation θ in the rule for $\mathsf{X}\phi$. Here the effect of the transformation morphism is brought to bear. For one thing, it is possible

that a variable becomes undefined, if the node that it was referring to is deleted by the transformation.

$G, \sigma, \theta \models x$	iff $\theta(x)$ is defined
$G, \sigma, \theta \models a(x, y)$	iff $(\theta(x), a, \theta(y)) \in E_G$
$G, \sigma, \theta \models \mathsf{tt}$	always
$G, \sigma, \theta \models \neg\Phi$	iff not $G, \sigma, \theta \models \Phi$
$G, \sigma, \theta \models \Phi_1 \vee \Phi_2$	iff $G, \sigma, \theta \models \Phi_1$ or $G, \sigma, \theta \models \Phi_2$
$G, \sigma, \theta \models \exists x : \Phi$	iff $G, \sigma, \theta\{v/x\} \models \Phi$ for some $v \in N_G$
$G, \sigma, \theta \models \mathsf{X}\Phi$	iff $src(\sigma_2), \sigma^2, \sigma_1 \circ \theta \vdash \Phi$
$G, \sigma, \theta \models \Phi_1 \mathsf{U} \Phi_2$	iff $\exists i \geq 0 : G, \sigma, \theta \models \mathsf{X}^i\Phi_2$ and $\forall 0 \leq j < i : G, \sigma, \theta \models \mathsf{X}^j\Phi_1$

We define the *validity* of a formula on a GTS S as follows:

$$S \models \Phi \quad \text{if for all runs } \rho \text{ of } S \text{ and all valuations } \theta\text{:} \quad G_{q_0}, \sigma_\rho, \theta \models \Phi$$

For instance, of the example formulae presented above, nrs. 1, 2, 5 and 6 (provided c is mapped to the cell pointed to by first) are valid on the GTS of Fig. 4, whereas the others are not. Property 2, for instance, holds because the morphisms associated with the transitions are such that after a finite number of transitions, each cell is mapped onto a cell with an outgoing val-edge. Property 5, on the other hand, is trivially valid since the start state is already empty; however, when another state is picked as start state it becomes invalid, since although it would hold for some paths of that modified GTS, there are runs that never enter the state where the buffer is empty.

The following is an important property, since it shows that bisimilarity minimisation does not change the validity of FOLTL properties.

Theorem 3. *If S_1, S_2 are GTSs, then $S_1 \sim S_2$ implies $S_1 \models \Phi$ iff $S_2 \models \Phi$ for all $\Phi \in$ FOLTL.*

We can now finally formulate the model checking question:

Model checking problem: Given a graph grammar $\mathcal{G}$ and a formula Φ, does $S_{\mathcal{G}} \models \Phi$ hold?

In general, this question is certainly undecidable. In cases where $S_{\mathcal{G}}$ is finite, however, we can use the following (which can be shown for instance by a variation on [29]):

Theorem 4. *Given a finite GTS S and a formula Φ, the property $S \models \Phi$ is decidable, with worst-case time complexity exponential in the number of variables in Φ.*

4.2 Graph-Based Linear Temporal Logic

We now present an alternative logic, which we call GLTL, based only on graphs (rather than predicates and variables) but equivalent (for simple graphs) to FOLTL as presented above. The ideas are based on our own work in [28], originally conceived as an extension to negative application conditions [13]; the same basic ideas were later worked out in a slightly different form in [6]. The extension of this principle to temporal logic is new here.

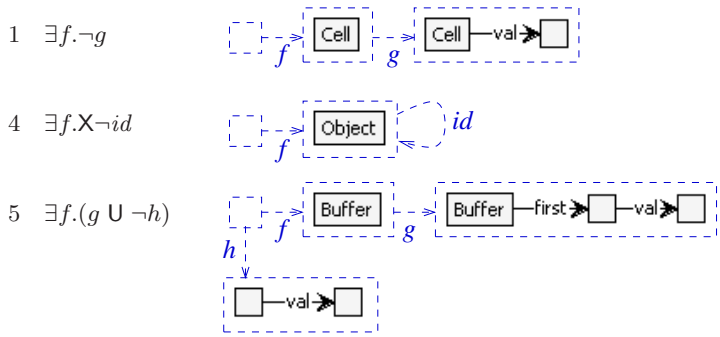

Fig. 5. Formulae in GLTL, corresponding to FOLTL nrs. 1, 4 and 5

The basic idea is to use morphisms as core elements of formulae. Thus, a GLTL formula is generated by the following grammar:

$$\Omega ::= \mathbf{tt} \mid \neg\Omega \mid \Omega \vee \Omega \mid \exists f.\Omega \mid \mathsf{X}\Omega \mid \Omega \;\mathsf{U}\; \Omega \;.$$

As a convenient notation we use the morphism f on its own as equivalent to $\exists f.\mathbf{tt}$.

A formula of the form $\exists f.\Omega$ is evaluated under an existing total matching θ of $src(f)$ to the current graph; the formula is satisfied if θ can be factored through f, i.e., there exists an η from $tgt(f)$ such that $\theta = \eta \circ f$. In fact, $src(f)$ acts as a "type" of $\exists f.\Omega$, and the sub-formula Ω is typed by $tgt(f)$, meaning that its evaluation can assume the existence of η. This notion of "type" replaces the notion of free variables of a (traditional) first-order formula. Types can be computed as follows:

$$\begin{aligned}
type(\mathbf{tt}) &= \langle \emptyset, \emptyset \rangle \\
type(\neg\Omega) &= type(\Omega) \\
type(\Omega_1 \vee \Omega_2) &= type(\Omega_1) \cup type(\Omega_2) \\
type(\exists f.\Omega) &= src(f) \quad \text{if } type(\Omega) \subseteq tgt(f) \\
type(\mathsf{X}\Omega) &= type(\Omega) \\
type(\Omega_1 \;\mathsf{U}\; \Omega_2) &= type(\Omega_1) \cup type(\Omega_2) \;\;.
\end{aligned}$$

Here, the union of two graphs is the (ordinary, not disjoint) union of the node and edge sets, and the sub-graph relation is likewise defined pointwise. (In fact, the types Ω_i of the operands of $\vee$ and U are regarded as sub-types of the type of the composed formula.) The side condition in the type definition for $\exists f.\Omega$ implies that the type can be undefined, namely if the type of Ω is not a sub-graph of $tgt(f)$. We only consider typable formulae.

For example, Fig. 5 shows some GLTL formulae that are equivalent to FOLTL formulae given earlier.

The semantics of GLTL is a relatively straightforward modification of FOLTL. The valuation θ is now a total graph morphism from the type of the formula to the graph. Due to the type definition, this means that in the evaluation we sometimes have to restrict

θ to sub-types. The rules that are different from FOLTL are given below; for the other operators, the semantics is precisely as defined above.

$$\begin{array}{ll} G,\sigma,\theta \models \Omega_1 \vee \Omega_2 & \text{iff } G,\sigma,\theta \restriction type(\Omega_1) \models \Omega_1 \text{ or } G,\sigma,\theta \restriction type(\Omega_2) \models \Omega_2 \\ G,\sigma,\theta \models \exists f : \Omega & \text{iff there is a } \eta\text{: } tgt(f) \to G \text{ such that } \theta = \eta \circ f \text{ and } G,\sigma,\eta \models \Omega \\ G,\sigma,\theta \models \Omega_1 \mathsf{U} \Omega_2 & \text{iff } \exists i \geq 0 : G,\sigma,\theta \restriction type(\Omega_2) \models \mathsf{X}^i \Omega_2 \\ & \quad \text{and } \forall 0 \leq j < i : G,\sigma,\theta \restriction type(\Omega_1) \models \mathsf{X}^j \Omega_1 \end{array}$$

There exists a relatively straightforward translation back and forth between FOLTL and GLTL. We explain the principles here on an intuitive level; see Fig. 5 for some concrete examples.

- From FOLTL to GLTL, formulae of the form $a(x,y)$ are translated to morphisms f with $src(f) = \langle \{x,y\}, \emptyset \rangle$ and $tgt(f) = \langle \{x,y\}, \{(x,a,y)\} \rangle$; formulae $x = y$ to non-injective morphisms mapping a two-node discrete graph to a one-node discrete graph while merging the nodes; and formulae $\exists x.\Phi$ to $\exists f.\Omega$ where f adds a single, unconnected node to its source.
- From GLTL to FOLTL, $\exists f.\Omega$ is translated to

$$\exists z_1, \ldots, z_n. \textstyle\bigwedge_i a_i(x_i, y_i) \wedge \bigwedge_j (x_j = y_j) \wedge \Phi \ ,$$

  where the z_k are variables representing the nodes that are new in $tgt(f)$ (i.e., not used as images by f), the $a_i(x_i, y_i)$ are edges that are new in $tgt(f)$, the $x_i = y_i$ equate nodes on which f is non-injective (in both cases, the x_i and y_i correspond to some z_j), and Φ is the translation of Ω.

We state the following result without proof.

Theorem 5. *There exist translations gltl*: FOLTL $\to$ GLTL *and foltl*: GLTL $\to$ FOLTL *such that for all GTSs S:*

$$\begin{array}{rcl} S \models \Phi & \iff & S \models gltl(\Phi) \\ S \models \Omega & \iff & S \models foltl(\Omega) \ . \end{array}$$

5 Evaluation and Future Work

In this section we evaluate some issues regarding choices made in the approach, as well as possible extensions and future challenges. In the course of this we will also touch upon related work, insofar not already discussed.

Graph formalism. In Sect. 3, we have discussed our choice of graph formalism, in the light of the existing algebraic theory surrounding DPO rewriting, only little of which has been successfully transferred to SPO. Let us briefly investigate what has to be done to lift our approach to a general DPO setting; that is, to a category of graphs that is an adhesive HLR category, with a set $\mathcal{M}$ of monos.

- In Sect. 3, our graph transitions carry partial morphisms. In a DPO setting, this should be turned into a span of arrows, of which the left arrow (pointing to the source graph of the transition) should be in $\mathcal{M}$. The corresponding notions of bisimilarity and isomorphism will become slightly more complicated, but we expect that the same results will still hold.

– In Sect. 4, it is not clear how to interpret FOLTL in an arbitrary HLR category. On the other hand, GLTL as defined in Sect. 4.2 can easily be generalised. For this purpose, the construction of the type of a formula (which now relies on subgraphs and graph union, neither of which can be generalised directly to categorical constructions) should be revised.
  A straightforward solution is to provide explicit monos with the binary operators to generalise the sub-graph relation; i.e., the formulae become $\Omega_1 \vee_{\iota,\kappa} \Omega_2$ and $\Omega_1 \mathsf{U}_{\iota,\kappa} \Omega_2$, where ι, κ is a co-span of monos (in $\mathcal{M}$) such that $src(\iota) = type(\Omega_1)$, $src(\kappa) = type(\Omega_2)$ and $tgt(\iota) = tgt(\kappa)$ equals the type of the formula as a whole. Furthermore, the morphism f in $\exists f.\Omega$ should be replaced by a span $\xleftarrow{\iota}\xrightarrow{f}$, where $type(\Omega) = tgt(f)$ and $tgt(\iota)$ is the type of the whole formula.

Existing model checkers. In the last decades, model checking has given rise to a large number of successful academic and commercial tools, such as SPIN [16], BLAST [4], JPF [17], Murphi [5] or Bogor [32]. Many of these tools share the aims of the GROOVE project, viz., verifying actual (object-oriented) code. It is, therefore, justified to ask what we can hope to add to this field, given the inherent complexities of the graph transformation approach. In fact, there are two distinct issues involved:

– *Graph transformation as a specification paradigm.* In our approach, we essentially propose to use graph transformations as a language to specify the semantics of programming languages. Existing tools use textual modelling languages for this purpose, such as Promela for SPIN or BIR for Bogor, or rely on the available compilation to byte code, as in the case of JPF.
  We believe graph transformations to be a viable alternative, for the following reasons:
  - Graphs provide a syntax which is very close to, if not coincides with, the intuitive understanding of object-oriented data structures (or even heap structures in general). Thus, graph-based operational semantics is easy to understand (see also [18]).
  - Graphs are also very close to diagram models as used in visual languages, and so provide an integrated view along the software engineering process.
  - As numerous case studies have shown, graph transformation can alternatively be used as a specification formalism in its own right. Verification techniques based on this paradigm can therefore also be used outside the context of software model checking.
– *Graph transition systems as verification models.* Even if one accepts the arguments given above in favour of graph transformation as a specification paradigm, this does not immediately imply creating a new model checker. Instead, it is perfectly thinkable to encode graph derivations in terms of the input languages of one of the existing tools, and so avoid re-inventing (or re-implementing) the wheel. Verification approaches based on this idea are, for instance, [35,12].
  We believe that it is nevertheless worthwhile to implement model checking directly on top of graph transition systems, for the following reasons:

- *Symmetry reduction.* In graphs, symmetry is equivalent to isomorphism, and collapsing the state space modulo isomorphism is an immediate method for (non-trivial) symmetry reduction (see also [30]). This connection is lost when graphs are encoded in some other language, since such an encoding invariably involves breaking symmetries.
- *Unboundedness.* Many of the existing model checkers rely on fixed-size bit vectors to represent states. Graphs, however, are not a priori bounded. In order to perform the encoding, it is therefore necessary to choose an upper bound for the state size, and to increase this if it turns out to have been set too low — which involves repeating the encoding step.
- *Encoding.* The complexity of finding acceptable matchings for rules does not suddenly disappear when the problem is encoded in another language. Instead, the encoding itself typically involves predicting or checking all possible matches; so the complexity of the graph transformation paradigm is (partially) shifted from the actual model checking to the encoding process.

Alternative approaches. The explicit state model checking approach presented here is probably the most straightforward way to define and implement verification for graph grammars. A promising alternative is the Petri graph method proposed by König et al.; see, e.g., [3,1,19]. This approach uses *unfolding* techniques developed originally for Petri nets, and offers good abstractions (identified below as one of the more important future work items in our approach); thus, the aproach can yield answers for arbitrary graph grammars. On the other hand, the logic supported is more limited, and it is not clear if symmetry reduction is possible.

Future work. Finally, we identify the following major challenges to be addressed before explicit-state model checking for graph grammars can really take off.

- *Partial order reduction.* This refers to a technique for only generating part of the state space, on which nevertheless a given fragment of the logic (typically, X-free LTL) can still be checked. Traditional techniques for partial order reduction do not apply directly to the setting of graph grammars, since the number of entities is not *a priori* known. (Note that the unfolding approach of [3,1] is in fact also a partial order reduction.)
- *Abstraction.* Instead of taking concrete graphs, which can grow unboundedly large, one may define graph abstractions, which collapse and combine parts of the graphs that are sufficiently similar. Inspired by *shape analysis* [34], we have investigated possible abstractions in [26,31]. This is still ongoing research; no implementation is yet available.
- *Compositionality.* In Sect. 3 we have pointed out the analogy of GTSs with History-Dependent automata, a long-established model for communicating systems, sharing some of the dynamic nature of graphs. This analogy can be turned around to inspire a notion of communication between graph transition systems where (parts of) graphs are echanged, leading to a theory of compositionality for graph grammars. The only work we know of in this direction is König and Ehrig's *borrowed context* [9], and the *synchronised hyperedge replacement* by Hirsch, Montanari and others [15].

References

1. Baldan, P., Corradini, A., König, B.: Verifying finite-state graph grammars: An unfolding-based approach. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 83–98. Springer, Heidelberg (2004)
2. Baldan, P., Corradini, A., König, B., Lluch-Lafuente, A.: A temporal graph logic for verification of graph transformation systems. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 1–20. Springer, Heidelberg (2007)
3. Baldan, P., König, B.: Approximating the behaviour of graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 14–29. Springer, Heidelberg (2002)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
5. Dill, D.L.: The murϕ verification system. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 390–393. Springer, Heidelberg (1996)
6. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundam. Inform. 74(1), 135–166 (2006)
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. Fundam. Inform. 74(1), 31–61 (2006)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
9. Ehrig, H., König, B.: Deriving bisimulation congruences in the dpo approach to graph rewriting. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 151–166. Springer, Heidelberg (2004)
10. Ehrig, H., Padberg, J., Prange, U., Habel, A.: Adhesive high-level replacement systems: A new categorical framework for graph transformation. Fundam. Inform. 74(1), 1–29 (2006)
11. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design 9(1/2), 105–131 (1996)
12. Ferreira, A.P.L., Foss, L., Ribeiro, L.: Formal verification of object-oriented graph grammars specifications. In: Rensink, A., Heckel, R., König, B. (eds.) Graph Transformation for Concurrency and Verification (GT-VC). Electr. Notes Theor. Comput. Sci, vol. 175, pp. 101–114 (2007)
13. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. 26(3/4), 287–313 (1996)
14. Habel, A., Pennemann, K.H.: Satisfiability of high-level conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 430–444. Springer, Heidelberg (2006)
15. Hirsch, D., Montanari, U.: Synchronized hyperedge replacement with name mobility. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 121–136. Springer, Heidelberg (2001)
16. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
17. Java PathFinder – A Formal Methods Tool for Java, `http://ase.arc.nasa.gov/people/havelund/jpf.html`
18. Kastenberg, H., Kleppe, A.G., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)
19. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 197–211. Springer, Heidelberg (2006)

20. Lack, S., Sobocinski, P.: Adhesive categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)
21. Lambers, L., Ehrig, H., Orejas, F.: Efficient detection of conflicts in graph-based model transformation. Electr. Notes Theor. Comput. Sci. 152, 97–109 (2006)
22. Levendovszky, T., Prange, U., Ehrig, H.: Termination criteria for dpo transformations with injective matches. Electron. Notes Theor. Comput. Sci. 175(4), 87–100 (2007)
23. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science 109(1–2), 181–224 (1993)
24. Montanari, U., Pistore, M.: History-dependent automata: An introduction. In: Bernardo, M., Bogliolo, A. (eds.) SFM-Moby 2005. LNCS, vol. 3465, pp. 1–28. Springer, Heidelberg (2005)
25. Rensink, A.: Towards model checking graph grammars. In: Gruner, S., Presti, S.L., eds.: Workshop on Automated Verification of Critical Systems (AVoCS), Southampton, UK. Volume DSSE-TR-, -02 of Technical Report., University of Southampton (2003) 150–160 (2003)
26. Rensink, A.: Canonical graph shapes. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 401–415. Springer, Heidelberg (2004)
27. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
28. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004)
29. Rensink, A.: Model checking quantified computation tree logic. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 110–125. Springer, Heidelberg (2006)
30. Rensink, A.: Isomorphism checking in GROOVE. In: Zündorf, A., Varró, D. (eds.) Graph-Based Tools (GraBaTs). Electronic Communications of the EASST, European Association of Software Science and Technology, vol. 1, Natal, Brazil (September 2007)
31. Rensink, A., Distefano, D.: Abstract graph transformation. Electr. Notes Theor. Comput. Sci. 157(1), 39–59 (2006)
32. Robby, D.M.B., Hatcliff, J.: Bogor: A flexible framework for creating software model checkers. In: McMinn, P. (ed.) Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART), pp. 3–22. IEEE Computer Society, Los Alamitos (2006)
33. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. I. World Scientific, Singapore (1997)
34. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
35. Varró, D.: Automated formal verification of visual modeling languages by model checking. Software and System Modeling 3(2), 85–113 (2004)