In M. Leuschel, S. Gruner and S. Lo Presti, eds, *3rd Workshop on Automated Verification of Critical Systems*, Tech. Report DSSE-TR-2003-2, pp. 150-160. University of Southampton, 2003

Towerds Model Checking Graph Grammars

Arend Rensink
Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE Enschede, The Netherlands
rensink@cs.utwente.nl

December 3, 2003

Abstract

We sketch a setup in which transition systems are generated from graph grammars and subsequently checked for properties expressed in a temporal logic on graphs. We envisage this as part of an approach where graph grammars are used to express the behavioural semantics of object-oriented programs, thus enabling automatic verification of those programs.

This paper describes work in progress.

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 Bandera [2] and Java Pathfinder [11] 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). Current 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.

For this purpose, in previous work we have proposed a temporal logic to reason about allocation and deallocation (with Distefano and Katoen, see [4]), interpreted on History-Dependent Automata (developed by Montanari and Pistore [17]), extended to allow a finite representation of unboundedly growing states, albeit only in a limited scenario. In this paper we propose to use *graph grammars* to generate transition systems consisting of graphs as states and partial graph morphisms as transitions. We define an extension of the logic studied in [4] that includes *regular navigation expressions* over graphs. We give some illustrative properties on a running example to demonstrate the strengths of our approach.

We are confident that the resulting logic can be model checked on finite graph transition systems, using an extension of the algorithm defined in [4]. The extension to unbounded states is left for future work.

We intend to let theory development and (research scale) tool implementation go hand in hand. The current status of the implementation is that we have developed a (prototype) *simulator* for graph grammars, capable of generating small graph transition systems (of up to 10^3 states); the next step is to extend this with a model checking algorithm for the logic presented here. The tool can be tried out; see [18].

This paper describes work in progress.

Related work. There is an extensive literature on graph grammars and graph transformations, but only a fragment of that addresses their application for modelling and analysing dynamic behaviour. Especially the work of Heckel, Corradini and others [10, 9, 8, 3, 1] is closely related. For instance, the interpretation of temporal logic over graph grammars is investigated in [10, 6] and other analysis techniques in [1]. The novelty of the current approach lies in the application of ideas from history-dependent automata [17] and allocational logic [4].

The application of graph transformations to object-oriented languages (which is not an aspect covered in this paper but is part of our planned future work) has been studied in, e.g., [20, 21].

2 Graph transition systems

We model system states as *graphs* of a particular kind. To define them formally, first we need some auxiliary concepts.

Concepts and notation 1

- *We assume a countable universe of nodes, denoted $\mathcal{N}$ and ranged over by p, q;*
- *We assume a countable universe of labels, denoted $\mathcal{L}$ and ranged over by a, b.*

Our graphs are edge-labelled but not node-labelled, and edges have no identity.[1]

Definition 2 (graph) *A graph is a tuple $G = \langle N, E \rangle$ where*

- *$N \subseteq \mathcal{N}$ is a set of* nodes*;*
- *$E \subseteq N \times \mathcal{L} \times N$ is a set of* edges*, such that $N \cap E = \emptyset$.*

Concepts and notation 3

- *The set of all graphs will be denoted $\mathcal{G}$, ranged over by G, H.*
- *The node and edge sets of a graph G will often be denoted N_G and E_G; likewise, for a graph G_i we use N_i, E_i.*
- *Given an edge $e = (p, a, q) \in E$, we call p, q and a the* source*,* target *and* label *of e, denoted $src(e)$, $tgt(e)$ and $\ell(e)$, respectively.*
- *Given graph G, E^* denotes the minimal subset of $N \times \mathcal{L}^* \times N$ such that $(p, \epsilon, q) \in E^*$ (where ϵ denotes the empty sequence) and if $(p, w, q) \in E^*$ and $(q, a, r) \in E$ then $(q, w\,a, r) \in E^*$. then*
- *When depicting graphs, for convenience we sometimes represent self-edges by inscribing their labels in the nodes and omitting the edges. Examples are the* empty*-labelled self-edges in Figure 1 below.*

[1] Note that this is analogous to the traditional definition of *labelled transition systems*; however, we use graphs in quite a different role, so that in fact this analogy is misleading.

A transition between two states is modelled by an injective (partial) morphism between the graphs. (In this we follow the so-called *single pushout approach* in graph transformations; see below.)

Definition 4 *Let G, H be two graphs. A* morphism *from G to H is a pair of partial functions $f = (f_N, f_E)$ with $f_N: N_G \rightharpoonup N_H$ and $f_E: E_G \rightharpoonup E_H$, such that the following partial confluence properties hold:*

- $tgt_H \circ f_E \subseteq f_N \circ src_G$;
- $tgt_H \circ f_E \subseteq f_N \circ tgt_G$;
- $\ell_H \circ f_E \subseteq \ell_G$.

The morphism is called injective [total] *if both f_N and f_E are injective [total].*

We use injective morphisms to model transitions from the morphism's domain to its codomain. The intuition is that nodes and edges in the source without an image in the target are deleted by taking this transition, whereas nodes and edges in the target without an inverse image in the source are created. This extends the principle of *reallocations*, as used in History-Dependent Automata (see Montanari and Pistore [17]), from multisets to graphs.

Concepts and notation 5

- *We will write f instead of f_N and f_E whenever there is no chance of confusion.*
- *We will write $f: G \to H$ to indicate that f is a morphism from G to H. In that case we write $dom(f) = G$ and $cod(f) = H$.*
- *The set of all graph morphisms is denoted $\mathcal{M}$; the subset of injective [total] morphisms is denoted $\mathcal{M}_{\mathrm{i}}$ [$\mathcal{M}_{\mathrm{t}}$].*

This sets the stage for the definition of our models of behaviour, which are transition systems over graphs.

Definition 6 *A* graph transition system *(*GTS*) is a triple $\langle S, \to, s_0 \rangle$ where*

- *$S \subseteq \mathcal{G}$ is a set of states;*
- *$\to \subseteq \mathcal{M}_{\mathrm{i}}$ is a set of transitions;*
- *$s_0 \in S$ is the initial state.*

Concepts and notation 7 *We sometimes write $G \xrightarrow{f} H$ to indicate that $f: G \to H$ is a GTS transition.*

An example graph transition system is depicted in Figure 1. This shows a circular buffer, consisting of three slots, each of which points to its predecessor through a prev-labelled edge. Two slots are designated first and last, respectively. Slots can be either free (indicated by a corresponding self-edge labelled empty — see Notation 3 for a remark concerning the representation of this self-edge) or occupied (indicated by a val-edge to a node representing the value).

The initial state of Figure 1 is the upper left one. If there are free slots in a state, the one directly preceding first may be filled with a fresh value (which corresponds to the operation of *inserting* that value into the buffer); if there are occupied slots, the one labelled last may be freed by deleting the value in that slot (which corresponds to *retrieving* the value from the buffer). The corresponding morphisms in the transition system are denoted ⟨insert⟩ and ⟨retrieve⟩. For the ⟨insert⟩-morphisms,

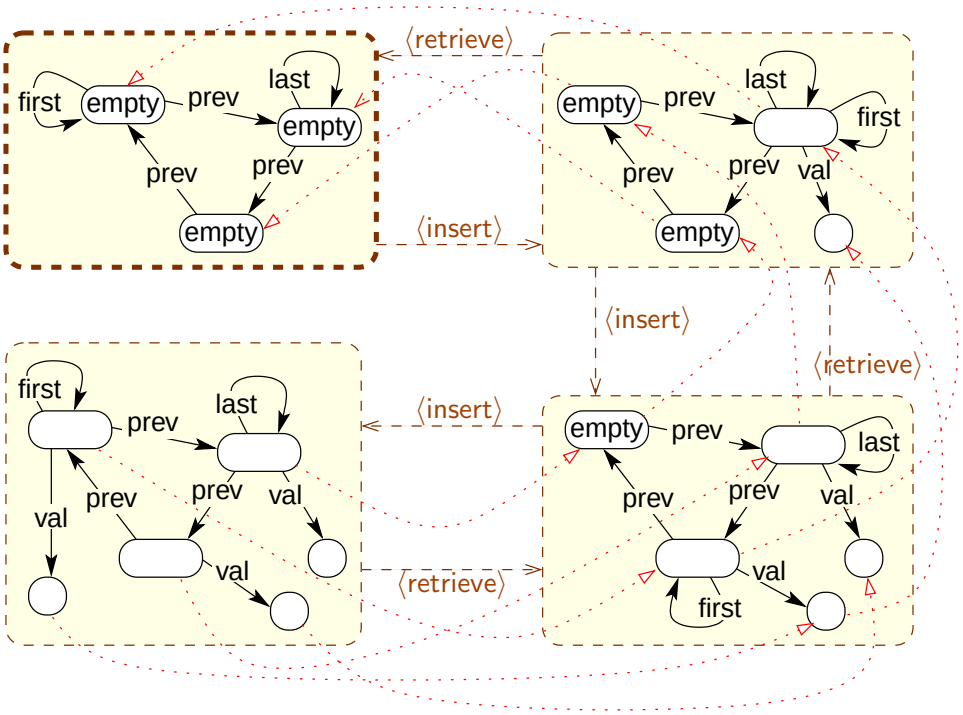

Figure 1: Graph transition system modelling a three-place circular buffer

the node mappings are implied by the position within the graph — i.e., each node in the source graph is mapped to the node in the same relative position of the target graph. For the ⟨retrieve⟩-morphisms, the node mappings are indicated by dotted lines. All edge mappings are left implicit. Besides identifying elements to be deleted and created, the graph morphisms also serve to reduce the state space. A straightforward "traditional" encoding of a three-place circular buffer (where slots are modelled as either full or empty) would give rise to a model with twelve reachable states; in general, the number of states is quadratic in the length of the buffer. Using graphs, we need only four states for the three-element buffer; in general the number of states is linear in the buffer length. This is due to the exploitation of symmetry by the graph morphisms. For instance, ⟨insert⟩ followed by ⟨retrieve⟩ leads back to the initial state; however, the concatenation of the corresponding morphisms is not the identity, but rather an auto-isomorphism; in other words, a symmetry of the state. This is in fact precisely the same principle as that used in minimisation of history-dependent automata (see [16]).

Besides exploiting symmetry, the morphisms in our GTSs also keep track of the identity of entities. For instance, Figure 1 contains all the information necessary to check that entities are retrieved in the order they are inserted, that no entity is inserted without eventually being retrieved, and that no entity can be retrieved without having been inserted. All this is established through the node identities of the value nodes; no actual values need be introduced. We will come back to this in the presentation of a temporal logic (Section 4).

3 Graph grammars

Those familiar with Petri nets will have noted a connection between graph transition systems and Petri net state spaces (which are essentially multisets of entities connected by multiset morphisms). In fact, it is quite possible to give a Petri net model of a buffer that shares many of the characteristics

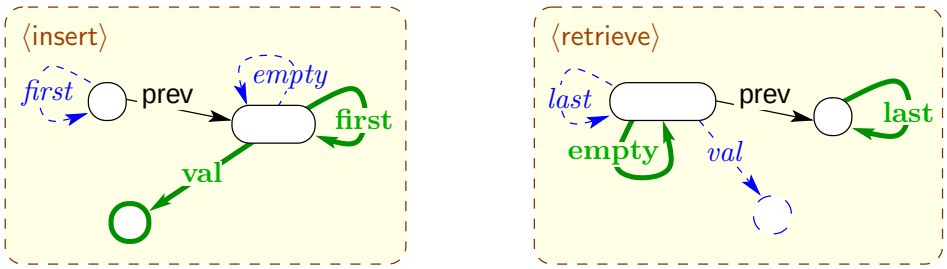

Figure 2: Graph production rules for Figure 1

of Figure 1 — but not all; in particular, the ordering of the values, which is maintained by the prev-links in our model, cannot be modelled in quite the same way in a Petri net.

In fact, GTSs can be generated from *graph grammars*, which are indeed a generalisation of Petri nets (see, e.g., [13, 7, 12]). Of the many flavours of graph grammars (see [19] for an authorative overview), for the purpose of this paper we follow the *single-pushout approach* introduced by Löwe [15], mainly because this offers a technically simpler presentation. Since in this paper we use graph grammars merely as factories for graph transition systems and we do not use any underlying graph rewriting theory, the basic ideas of this paper are independent of this choice.

Definition 8 (production rule) *A* graph production rule *is a graph whose nodes and edges are partitioned into (possibly empty) sets* N^{del}, N^{use} *and* N^{new} *(resp.* E^{del}, E^{use} *and* E^{new}*) such that*

- $src(e) \in N^{\mathsf{del}}$ *or* $tgt(e) \in N^{\mathsf{del}}$ *implies* $e \in E^{\mathsf{del}}$*;*
- $src(e) \in N^{\mathsf{new}}$ *or* $tgt(e) \in N^{\mathsf{new}}$ *implies* $e \in E^{\mathsf{new}}$*.*

For those familiar with the single-pushout approach: the above definition is equivalent to usual presentation, but lends itself better to visualisation. We point out the connection briefly in Section A.

The intuition is that a production rule *deletes* del nodes and edges, *uses* (i.e., requires the presence of) use nodes and edges, and *creates* new nodes and edges. We will represent production rules graphically by using dashed dark blue lines and italic font for del, continuous thin black lines and default font for use and fat green lines and bold font for new. For instance, Figure 2 shows the two production rules that give rise to the graph transformations in Figure 1. An *application* of a production rule P to a given graph G involves finding a total and surjective morphism $\mu\colon P \to Q$ such that

- $\mu(p) = \mu(q)$ implies $p = q$ or $p, q \notin N^{\mathsf{new}}$ — i.e., μ is injective on the new nodes);
- $N_G \cap N_Q = \mu(N^{\mathsf{del}} \cup N^{\mathsf{use}})$ and $E_G \cap E_Q = \mu(E^{\mathsf{del}} \cup E^{\mathsf{use}})$ — i.e., the del and use nodes are projected by μ into G whereas the new nodes are kept disjoint from G.

Given such a μ, P transforms G into a new graph H defined by

$$\begin{aligned} N_H &= (N_G \setminus \mu(N^{\mathsf{del}})) \cup \mu(N^{\mathsf{new}}) \\ E_H &= ((E_G \setminus \mu(E^{\mathsf{del}})) \cup \mu(E^{\mathsf{new}})) \restriction N_H \end{aligned}$$

(The restriction to N_H in the definition of E_H comes down to deleting edges whose sources or targets are not in N_H — which may be necessary if $\mu(p) = \mu(src(e))$ for some $p \in N^{\mathsf{del}}$ and $e \notin E^{\mathsf{del}}$.) The connection between G and H is fixed by the injective morphism $f\colon G \to H$ defined by $f = (id_{N_G \cap N_H}, id_{E_G \cap E_H})$.

For instance, example applications of the rules in Figure 2 can be found in Figure 1. It should be noted that every μ gives rise to a different transformed graph H. However, variations in $\mu \restriction N^{\mathsf{new}}$ lead to isomorphic copies of H; therefore, only variations in $\mu \restriction (N^{\mathsf{del}} \cup N^{\mathsf{use}})$ need be considered. For instance, in the case of Figure 1 every rule can be applied at most once in a given state.

Concepts and notation 9 *We write $G \xrightarrow{P,\mu,f} H$ to denote that the application of P to G under μ gives rise to the transformation $f\colon G \to H$.*

Definition 10 (graph grammar) *A* graph grammar *is a tuple $(G, \mathcal{P})$ consisting of an* initial graph *G and a set of production rules $\mathcal{P}$.*

We say that a graph grammar *generates* a graph transition system if the transitions in the GTS correspond to all possible rule applications to all reachable states, with the proviso that there may be only one isomorphic representative for every state. Formally, a GTS $T = (S, \to, G)$ is generated by a graph grammar $(G, \mathcal{P})$ if S is a minimal set of graphs such that

- $G \in S$;
- If $H \xrightarrow{P,\mu,f} K$ for some $H \in S$ and $P \in \mathcal{P}$, then there is a state $F \in S$ with an isomorphism $g\colon K \to F$ such that $H \xrightarrow{g \circ f} F$ is a transition.

Note that this does not uniquely define the transition system generated by a given grammar; however, any two transition systems satisfying these criteria are isomorphic, in the following sense.

Proposition 11 *If $(S_i, \to, s_{0i})$ for $i = 1, 2$ are transition systems generated by a given graph grammar then there exists a bijection $f\colon S_1 \to S_2$ such that (i) $f(s_{01}) = s_{02}$ and (ii) for all $G \in S_1$, there is a graph isomorphism $g_G\colon G \to f(G)$ such that*

$$G \xrightarrow{f} H \quad \textit{iff} \quad f(G) \xrightarrow{g_G \circ f \circ g_H^{-1}} f(H) \ .$$

For this reason we are justified in speaking of *the* graph transformation system generated by a given graph grammar. For instance, Figure 1 is the GTS generated by the graph grammar consisting of the production rules in Figure 2, with as initial graph the top left hand state of Figure 1. Note that a change to the initial graph results in a different GTS; e.g., starting with the left hand side in Figure 3, the rules in Figure 2 generate the behaviour of a five-slot circular buffer, whereas for the right hand side (which is an inconsistent state since one slot is neither full nor marked empty) the generated GTS shows a deadlock after the sequence $\langle \mathsf{retrieve} \rangle \cdot \langle \mathsf{insert} \rangle$.

4 Temporal logic

Having established the models, it is now time to present the logic to express properties to be verified. Formulas are generated by the following grammar:

$$\begin{array}{lcl} \pi & ::= & \ell \mid \pi + \pi \mid \pi.\pi \mid \pi^* \ . \\ \zeta & ::= & x \mid Z \mid \zeta.\pi \ . \\ \varphi & ::= & x \in \zeta \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x : \varphi \mid \exists Z : \varphi \mid \mathsf{X}\varphi \mid \varphi \ \mathsf{U} \ \varphi \ . \end{array}$$

This is a second-order linear temporal logic: x is a first-order variable ranging over nodes, whereas Z is a second-order variable ranging over node *sets*.

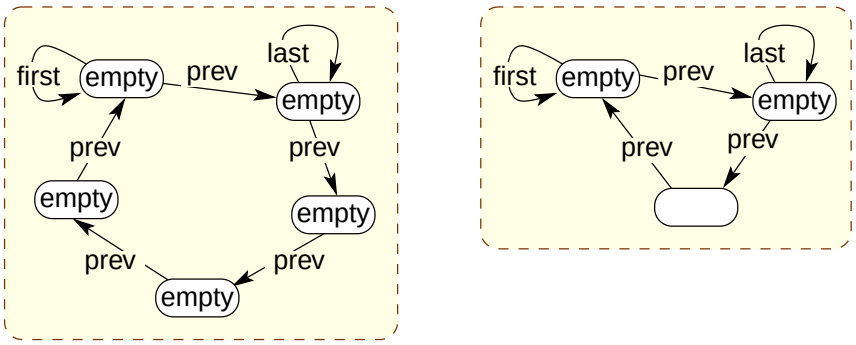

Figure 3: Alternative start states for the production rules in Figure 2

- π stands for (regular) *path expressions* over edge labels, used for navigating through graphs.
- ζ stands for *set expressions*: x is an empty or singleton set, depending on whether x is currently defined (see below), Z is a set of nodes, and $\sigma.\pi$ is the set of nodes reachable from σ by a path in π.
- φ stands for *formulae*. The most unusual are $\exists x : \varphi$, which selects an arbitrary node of a graph satisfying φ, and $\exists Z : \varphi$, which selects an arbitrary set of nodes. The formulae $\mathsf{X}\varphi$ and $\varphi \mathsf{U} \varphi$ stand for "next" and "until", as usual in LTL.

Formulae are interpreted over finite and infinite sequences of graph morphisms, in combination with a valuation of the first- and second-order variables. The definition requires some auxiliary notation.

Concepts and notation 12

- *Given a GTS $\langle S, \rightarrow, s_0 \rangle$, we let $\rightarrow^\omega$ denote the set of maximal paths through the GTS, defined as*
$$\{f_1\, f_2 \cdots \mid s_0 \xrightarrow{f_1} s_1 \xrightarrow{f_2} s_2 \cdots\} \cup \{f_1\, f_2 \cdots f_n \mid s_0 \xrightarrow{f_1} s_1 \cdots \xrightarrow{f_n} s_n \not\rightarrow\}$$
- *We let σ range over $\rightarrow^\omega$. $|\sigma|$ denotes the length of σ ($= \infty$ if σ is an infinite sequence). For all $0 < i \leq |\sigma|$, σ_i denotes the i'th element of σ (i.e., f_i), and, σ^i the tail of σ starting at the i'th element (i.e., $f_i\, f_{i+1} \cdots$).*
- *θ stands for a valuation mapping every first-order variable to either an element of $\mathcal{N}$ or to $\perp$ (representing* undefined*), and every second-order variable to a subset of $\mathcal{N}$.*
- *If θ is a valuation as above and f a graph morphism, then $f \circ \theta$ is defined by concatenating f_N (extended strictly to $\perp$) with θ.*
- *If θ is a valuation, then $\theta[^p/_x]$ (with $p \in \mathcal{N}$) denotes a derived valuation that maps x to p and all other variables to their θ-images; and likewise for $\theta[^P/_Z]$ (with $P \subseteq \mathcal{N}$).*

In order to define the semantics of the logic, first we have to establish the meaning of path and set expressions. $[\![\pi]\!] \subseteq \mathcal{L}^*$ is defined as usual for regular expressions. Set expressions are evaluated relative to a graph and valuation: if $P \subseteq N_G$ then

$$\begin{aligned} [\![x]\!]_{\theta,G} &= \begin{cases} \emptyset & \text{if } \theta(x) = \perp \\ \{\theta(x)\} & \text{otherwise} \end{cases} \\ [\![Z]\!]_{\theta,G} &= \theta(Z) \\ [\![\zeta.\pi]\!]_{\theta,G} &= \{q \mid \exists p \in [\![\zeta]\!]_{\theta,G} : \exists w \in [\![\pi]\!] : (p, w, q) \in E_G^*\} \end{aligned}$$

Thus, for instance, if θ binds Z to the set of all nodes in one of the graphs in Figure 1, then

- $[\![Z.\mathsf{first}]\!]$ is a singleton set consisting of the node with the first-self-edge;
- $[\![Z.\mathsf{val}]\!]$ is the set of all value nodes in the graph;
- $[\![Z.\mathsf{first.val}]\!]$ is either empty (if the entire buffer is empty) or a singleton set with the value referred to by the first buffer element.

Satisfaction of a formula is expressed through a predicate of the form $G, \sigma, \theta \vDash \varphi$, where $G = dom(\sigma_1)$. The following set of rules defines the satisfaction relation for formulae. The main point to notice is the modification of the valuation θ in the rule for $\mathsf{X}\varphi$. Here the effect of the transformation morphism is brought to bear. For one thing, it is possible that a (first-order) variable becomes undefined, if the node that it was referring to is deleted by the transformation.

$$
\begin{array}{lll}
G, \sigma, \theta \vDash x \in \zeta & \text{iff} & \theta(x) \in [\![\zeta]\!]_{\theta,G} \quad \text{(which fails to hold if } x \notin dom(\theta)\text{)} \\
G, \sigma, \theta \vDash \neg\varphi & \text{iff} & \text{not } G, \sigma, \theta \vDash \varphi \\
G, \sigma, \theta \vDash \varphi_1 \vee \varphi_2 & \text{iff} & G, \sigma, \theta \vDash \varphi_1 \text{ or } G, \sigma, \theta \vDash \varphi_2 \\
G, \sigma, \theta \vDash \exists x : \varphi & \text{iff} & G, \sigma, \theta[^p/_x] \vDash \varphi \text{ for some } p \in N_G \\
G, \sigma, \theta \vDash \exists Z : \varphi & \text{iff} & G, \sigma, \theta[^P/_Z] \vDash \varphi \text{ for some } P \subseteq N_G \\
G, \sigma, \theta \vDash \mathsf{X}\varphi & \text{iff} & cod(\sigma_1), \sigma^2, \sigma_1 \circ \theta \vdash \varphi \\
G, \sigma, \theta \vDash \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists i \geq 0 : G, \sigma, \theta \vDash \mathsf{X}^i\varphi_2 \text{ and } \forall 0 \leq j < i : G, \sigma, \theta \vDash \mathsf{X}^j\varphi_1
\end{array}
$$

For instance, in this logic we can express the following properties:

- x and y refer to the same node:
$$x \in y$$
- x has an outgoing a-edge:
$$\exists y : y \in x.a$$
- The node that x refers to will be deleted in the next state:
$$\mathsf{X}(x \notin x)$$
- Every node reachable through a val-edge will eventually be deleted:
$$\forall x : (\exists y : x \in y.\mathsf{val}) \Rightarrow (\mathsf{true}\ \mathsf{U}\ (x \notin x)))$$
- Eventually a fresh val-reachable node will be created:
$$\forall Z : \mathsf{true}\ \mathsf{U}\ (\exists x : x \notin Z \wedge (\exists y : x \in y.\mathsf{val}))$$

In order to evaluate formulae over GTSs, we need to define what the valid runs are. Although in the future we will probably need a Büchi-like acceptance condition, for the purpose of this paper it suffices to allow all maximal runs (finite or infinite) of the GTS. Thus, we define the *validity* of a formula on a GTS as follows:

$$\langle S, \rightarrow, s_0 \rangle \vDash \varphi \quad \text{iff} \quad \forall \sigma \in \rightarrow^\omega, \forall \theta : s_0, \sigma, \theta \vDash \sigma$$

Our main theorem is the following:

Theorem 13 *Given a finite GTS T and a formula φ, the property $T \vDash \varphi$ is decidable.*

It should be noted that this actually has the status of a hypothesis that we strongly believe to be true; the proof has not been done. However, the same proof technique that we used in [4] (based on Pnueli and Lichtenstein [14]) is applicable here, and there does not seem to be a reason to believe that the proof does not carry through.

A Production rules in the Single Pushout approach

Definition 8 deviates from the usual presentation in, e.g., [15, 5], in which a production rule consists of two graphs (L, R) connected by an injective morphism φ. In fact the two definitions are equivalent:

- The graph L in the traditional presentation is the union of del- and use-nodes and edges;
- The graph R in the traditional presentation is the union of use- and new-nodes and edges;
- The morphism φ in the traditional presentation corresponds to the identity over the use-nodes and edges.

Vice versa, for every production rule (L, R, φ) in the traditional presentation there exists an equivalent rule $P = (N, E)$ according to Definition 8, which can be obtained by taking the disjoint union of L and R while identifying nodes and edges connected through φ.[2] Then N^{del} consists of the nodes of L that are not in the domain of φ and N^{new} of the nodes of R that are not in the codomain of φ; and similar for E^{del} and E^{new}.

[2] More precisely, it is the pushout of $\varphi: G \to R$ and $id_G: G \to L$, where $G = (dom(\varphi_N), dom(\varphi_E))$ and $id_G = (id_{N_G}, id_{E_G})$.

References

[1] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In K. Larsen and M. Nielsen, editors, *Concur 2001: Concurrency Theory*, volume 2154 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, 2001.

[2] Bandera — Java model checker. `http://www.cis.ksu.edu/~santos/bandera`.

[3] A. Corradini, R. Heckel, and U. Montanari. Graphical operational semantics. In A. Corradini and R. Heckel, editors, *ICALP 2000 Workshop on Graph Transformations and Visual Modelling Techniques*. Carleton Scientific, 2000.

[4] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the Era of Network and Mobile Computing*, volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer Academic Publishers, 2002.

[5] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation, part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [19], pages 247–312.

[6] 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, *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation*, volume 1764 of *Lecture Notes in Computer Science*, pages 310–322. Springer-Verlag, 2000.

[7] H. J. Genrich, D. Janssens, G. Rozenberg, and P. S. Thiagarajan. Petri nets and their relation to graph grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Proc. 2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 115–129, 1983.

[8] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 138–153. Springer-Verlag, 1998.

[9] R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems*. PhD thesis, TU Berlin, 1998.

[10] R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Integrating the specification techniques of graph transformation and temporal logic. In I. Prívara and P. Ruzicka, editors, *Mathematical Foundations of Computer Science 1997*, volume 1295 of *Lecture Notes in Computer Science*, pages 219–228. Springer-Verlag, 1997.

[11] Java pathfinder — a formal methods tool for Java. `http://ase.arc.nasa.gov/people/havelund/jpf.html`.

[12] M. Korff and L. Ribeiro. Formal relationship between graph grammars and Petri nets. *Lecture Notes in Computer Science*, 1073:288–??, 1996.

[13] H.-J. Kreowski. A comparison between Petri-nets and graph grammars. In H. Noltemeier, editor, *Graphtheoric Concepts in Computer Science, Proceedings of the International Workshop WG'80*, number 100 in Lecture Notes in Computer Science, pages 306–317. Springer-Verlag, 1981.

[14] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107. ACM, 1985.

[15] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Comput. Sci.*, 109(1–2):181–224, 1993.

[16] U. Montanari and M. Pistore. Minimal transition systems for history-preserving bisimulation. In R. Reischuck and M. Morvan, editors, *STACS '97*, volume 1200 of *Lecture Notes in Computer Science*, pages 413–425. Springer-Verlag, 1997.

[17] U. Montanari and M. Pistore. History-dependent automata. Technical Report TR-11-98, Department of Computer Science, University of Pisa, 1998.

[18] A. Rensink. GROOVE: A graph transformation tool set for the simulation and analysis of graph grammars. Available at `http://www.cs.utwente.nl/~groove`, 2003.

[19] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, Signapore, 1997.

[20] G. Taentzer. How to integrate graph transformation with an object-oriented language. In *Sixth International Workshop on Theory and Application of Graph Transformations*, 1998.

[21] A. Wagner and M. Gogolla. Semantics of object-oriented languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume II: Applications, Languages and Tools, pages 181–121. World Scientific, Signapore, 1999.