Unfolding Symbolic Attributed Graph Grammars

Maryam Ghaffari Saadat^{1(✉)}, Reiko Heckel¹, and Fernando Orejas²

¹ Department of Informatics, Leicester University, Leicester, UK
{mgs17,rh122}@leicester.ac.uk

² Department of Computer Science, Polytechnic University of Catalonia,
Barcelona, Spain
orejas@cs.upc.edu

Abstract. Attributed graph grammars can specify the transformation of complex data and object structures within a natural rule-based model of concurrency. This is crucial to their use in modelling interfaces of services and components as well as the evolution of complex systems and networks. However, the established concurrent semantics of graph grammars by unfolding does not cover attributed grammars. We develop a theory of unfolding for attributed graph grammars where attribution is represented symbolically, via logical constraints. We establish a functorial representation (a coreflection) of unfolding which guarantees it to be correct, complete and fully abstract.

As a case study and running example we demonstrate the use of visual contracts to specify an escrow smart contract.

Keywords: Symbolic attribute graph transformation · Unfolding semantics · Visual smart contracts

1 Introduction

The majority of software developed today is distributed and concurrent, including mobile, service-oriented, cloud- and component-based applications, smart contracts and decentralised applications (Dapps) on blockchain platforms such as Ethereum or Neo, P2P applications, etc. Technical and organisational challenges, such as the lack of a central authority both at runtime and during development are being addressed by model-based software engineering methods. In order to model such applications using graph transformation systems, we rely on a natural rule-based model of concurrency. In its most comprehensive form, the concurrent behaviour of graph grammars is formalised by their *unfolding* [1, 3, 4]. This represents in one structure the branching computations of the grammar in (what could be called) a partial-order variant of its derivation tree.

Many practical applications of graph transformation, including to model-based software engineering, require graphs attributed by numerical or textual data and transformations combining structural with attribute updates, i.e.,

attributed graph grammars [5]. However, the theory of unfolding has so far only been developed for the un-attributed case. Indeed, in the presence of attributes, the concurrent and non-deterministic behaviour of a graph grammar becomes significantly more complex. While, traditionally, dependencies and conflicts between transformations are based on how their left- and right-hand sides overlap in given or derived graphs, and how these overlaps include elements created or deleted by one rule and required or preserved by the other, the presence of data requires a deeper analysis of conflicts and dependencies. For example, two rules updating the same attribute, one after the other, may still be exchangeable if the attribute operations performed are commutative. Also, the choice of different assignments for variables may lead to infinitely branching systems even if the branching structure of their underlying structural grammar is finite.

In this paper we lift the theory of unfolding to attributed graph grammars, providing the semantic foundations for concurrent and distributed system models with data. By treating attribution purely logically, based on the notion of symbolic attributed graphs, we are able to separate attribute computations from structural unfolding. In particular, the use of *lazy symbolic graph transformations* [9] allows to abstract from specific attribute values while keeping track of all constraints such values have to satisfy. The resulting theory follows the un-attributed case in establishing unfolding as a coreflection, i.e., a right adjoint to the inclusion of the subcategory of occurrence grammars into the category of attributed graph grammars. This shows the construction to be correct and complete in representing only and all steps of the given grammar, and to be fully abstract (i.e. represented in a minimal way).

We introduce the main concepts of symbolic attributed graph grammars, including the notion of lazy symbolic transformations, and present the escrow contract model as running example. Section 3 recalls the unfolding of classical DPO graph grammars, which is extended to attributed grammars in Sects. 4 and 5. In Sect. 6 we draw conclusions and discuss related and future work.

2 Symbolic Attributed Graph Transformation

Given a data algebra D with non-empty carrier sets, an *attributed graph* [5] over D is a graph whose *graph objects* (nodes and edges) are labelled over D . Elements of D are represented as *data nodes* and connected to graph objects by *attribute links*. Formally this structure is known as an *E-graph* [5]. A *symbolic attributed graph* [8] specifies a class of attributed graphs by means of constraints. Formally $\langle G, \Phi \rangle$ is an E-graph whose graph objects are labelled by a set of *variables* $x \in X_G$, together with a set of *constraints* Φ over X_G .

In our case, constraints are equations over X_G using values in D as constants. Each substitution $\sigma : X_G \rightarrow D$ extends canonically to such equations. If they hold in D , denoted $D \models \sigma(\Phi)$, this defines an *attributed graph* $\sigma(G)$ replacing variables x with data values $\sigma(x)$. Hence, the semantics of $\langle G, \Phi \rangle$ is $Sem(\langle G, \Phi \rangle) = \{\sigma(G) \mid D \models \sigma(\Phi)\}$. To simplify notation we will identify G with the symbolic attributed graph, denoting its constraints Φ_G .

Fig. 1. Symbolic attributed type and instance graphs

An *attributed graph morphism* $h : \langle G_1, \Phi_1 \rangle \rightarrow \langle G_2, \Phi_2 \rangle$ is a graph homomorphism $h : G_1 \rightarrow G_2$ such that $D \models (\Phi_2 \Rightarrow h(\Phi_1))$, where $h(\Phi_1)$ are the formulas obtained by replacing in Φ_1 every variable x_1 by $h(x_1)$. Symbolic graphs and morphisms over D form the category $\mathbf{SGraph}_D$, or just $\mathbf{SGraph}$ if D is understood.^{1,2}

In the rest of the paper, we will work with typed graphs. Given a dedicated *type graph* TG defining the types of nodes, edges and attributes, a *TG-typed (symbolic attributed) instance graph* is a graph G with a morphism $t_G : G \rightarrow TG$ mapping elements in G to their types. The category $\mathbf{SGraph}_{TG}$ has TG -typed instance graphs $\langle G_i, t_{G_i} \rangle$ as objects and as arrows $h : \langle G_1, t_{G_1} \rangle \rightarrow \langle G_2, t_{G_2} \rangle$ attributed graph morphisms $h : G_1 \rightarrow G_2$ such that $t_{G_2} \circ h = t_{G_1}$.

Example 1 (type and instance graphs of escrow model). Figure 1 shows a type graph in the bottom right corner and one of its instances in the top left. Note that, according to our definitions, this is an E-graph whose data nodes are variables related to their values by equations. For example, node $w1$ is a node of type `Wallet` and has an attribute link of type `balance` pointing to an (invisible) variable node (say b) whose value is given by an equation $b = 100$ in the constraints Φ_G .

Modern blockchain technologies allow developers to offer smart contracts as services based on the blockchain's distributed data model and other basic features, such as wallets and payments. A popular example is an escrow service,

¹ We use D to denote both the many-sorted algebra and the disjoint union of its carrier sets.

² We write $h(\Phi_1)$ in a slight abuse of notation. To be more rigorous (and less readable) we should write $h_X^\#(\Phi_1)$, where h_X is the restriction of h to the set of variables of G_1 , mapping them to variables in G_2 , and $h_X^\#$ is the (unique) extension of h_X to terms over D 's signature.

to act as intermediary between sellers and buyers in commercial transactions. The type graph defines the data model for such a service. Type escrow acts as container and defines a stake factor, by which the price of a widget is multiplied to define the stake to be raised by both seller and buyer. Items to be sold and bought are of type Widget, with attributes for cost (reflecting the cost to the seller at which a widget was obtained or produced), value (the presumed value to a prospective buyer), and delivery (shipping cost). Type Contract represents the smart contract, created when buyer and seller enter into a transaction, to maintain the relevant data.

In general, an attribute can have a set of values, i.e., link to several variable nodes. For a TG -typed graph G and a graph object (node or edge) o in G we write $o.at$ for the set of variables x such that there is an attribute link al in G from o to x with $t_G(al) = at$. If $o.at$ is singleton, we write $o.at = x \in G$. Generally, in rules, input and reachable graphs, attribution is unique, i.e., $o.at$ is singleton for all graph objects o and attributes at .

As illustrated in the example above, an attributed graph can be presented as a symbolic one by replacing all its attribute values in the data algebra D by variables, and for each value v , where x_v is the variable replacing it, including an equation $x_v = v$ in the set of constraints. We call such symbolic graphs *grounded*.

A symbolic attributed graph G is *linear* if each variable $x \in X_G$ occurs as an attribute value of at most one node or edge in G . A *TG-typed symbolic attributed graph transformation rule* over D is a triple $r = \langle \Phi_L, L \hookleftarrow K \hookrightarrow R, \Phi_R \rangle$, where $L \hookleftarrow K \hookrightarrow R$ is a span of TG -typed graph inclusions such that L is linear, $X_L = X_K \subseteq X_R$, and where Φ_L, Φ_R are sets of constraints over $X_L \cup D$ and $X_R \cup D$, respectively, satisfying $D \models (\Phi_R \Rightarrow \Phi_L)$.

Typed symbolic attributed graph transformation is defined following the double-pushout (DPO) approach [5] by a transformation of the underlying E-graphs satisfying the constraints: Given a rule r as above and a morphism $m : L \rightarrow G$, a transformation $\langle G, \Phi_G \rangle \xrightarrow{r, m} \langle H, \Phi_G \cup m'(\Phi_R) \rangle$ exists iff there is a DPO step $G \xrightarrow{r, m} H$ of the E-graphs such that $D \models (\Phi_G \Rightarrow m(\Phi_L))$ and $\Phi_G \cup m'(\Phi_R)$ is satisfiable in D . That means, if the left-hand side constraints are implied by the constraints in the given graph, the right-hand side constraints are added to the derived graph assuming that the result is consistent.

The use of the DPO approach to define the structural transformation means that rule r is only applicable at match m if the *dangling* and *dangling identification conditions* are satisfied. Match m satisfies the dangling condition if no node in G about to be deleted by r is the source or target of an edge outside $m(L)$ (because such an edge would be left “dangling” when deleting the node). The identification condition requires that a node or edge x in $L \setminus R$ (i.e., $m(x)$ is deleted by the rule) cannot be identified with any other object y in L , i.e., $m(x) = m(y)$ implies $x = y$. This ensures resource consciousness for deleted graph objects while allowing non-injective matching in general.

Example 2 (sell rule and transformation). The top half of Fig. 2 shows an attributed graph transformation rule from our escrow example specifying an operation `sell(u1:User, w:Widget, p:Int): Contract`. We only show graphs L and R ,

with the interface left K implicit as their intersection. The rule assumes objects and data on the left of the arrow, introducing variables such as $e : Escrow$ and sf to capture them for use in the right-hand side to create links to new objects such as $c : Contract$ and in expressions of attribute assignments and updates. The match m is controlled by input parameters $u1, w, p$ subject to the condition in the yellow box restricting the choice of price p .

Fig. 2. Attributed rule and transformation (Color figure online)

The variables in this rule are $X_L = \{p, sf, b1, cst, val, d\}$ and $X_R = X_L \cup \{b1', b, s\}$, the latter referring to the new values of the $w.balance$, $c.balance$, and $c.stake$, respectively. The input parameter p is part of the left-hand side and therefore chosen by the match. This leads to branching over all values of p , a degree of non-determinism that makes the direct computation of all derivations impossible. $\Phi_L = \{cst + d < p < val\}$ and $\Phi_R = \Phi_L \cup \{balance = p \times sf, price = p, stake = p \times sf, balance = b1 - p \times sf\}$. In the bottom half we show an application of the rule, transforming an instance graph which, in addition to the objects in the rule, has a second user. Note that attributes here have actual values, and how new values are computed in the derived graph on the right.

Constraints added by r can be satisfied by several non-isomorphic graphs. Rule r is *ground-preserving* if for any grounded graph G and transformation $G \xrightarrow{r, m} H$, graph H is grounded. All rules in this paper are ground-preserving.

To separate the handling of constraints, we may want to apply a rule $r = \langle \Phi_L, L \hookleftarrow K \hookrightarrow R, \Phi_R \rangle$ without checking entailment of $m(\Phi_L)$. This is equivalent to applying the corresponding *lazy rule* $r^\emptyset = \langle \emptyset, L \hookleftarrow K \hookrightarrow R, \Phi_R \rangle$, in a *lazy transformation*.

Proposition 1 (lazy transformation of grounded graphs). *A symbolic graph transformation rule $r = \langle \Phi_L, L \hookleftarrow K \hookrightarrow R, \Phi_R \rangle$ and its lazy counterpart $r^\emptyset = \langle \emptyset, L \hookleftarrow K \hookrightarrow R, \Phi_R \rangle$ are equivalent on grounded graphs. That means, given a symbolic transformation $G \xrightarrow{r, m} H$ with G grounded, this also represents a lazy transformation using $r^\emptyset$, and vice versa.*

Sometimes we want to translate graphs, rules and transformations from one type graph to another. This is possible because a morphism $f : TG_1 \rightarrow TG_2$ between type graphs induces a translation of instances. Formally, we denote by $f_{TG}^> : \mathbf{SGraphs}_{TG_1} \rightarrow \mathbf{SGraphs}_{TG_2}$ the covariant retyping functor from TG_1 -typed to TG_2 -typed instance graphs, defined by composition of f_{TG} with the typing morphisms as $f_{TG}^>(\langle G_1, t_{G_1} \rangle) = \langle G_1, f_{TG} \circ t_{G_1} \rangle$. Note that, the functor only affects the typing, i.e., the attributed graphs with their variables and constraints remain unchanged. Hence $f_{TG}^>$ also acts as identity on morphisms.

Definition 1 (attributed graph grammar). A (typed symbolic) attributed graph grammar (AGG) over a data algebra D is a 4-tuple $\mathcal{AG} = \langle TG, G_{in}, P, \pi \rangle$, where TG is a type graph over D , G_{in} is the TG -typed initial graph, P is a set of rule names, and π is a mapping associating to any $p \in P$ a TG -typed rule $\pi(p) = \langle \Phi_{L_p}, L_p \leftarrow K_p \hookrightarrow R_p, \Phi_{R_p} \rangle$ over D .

An AGG morphism $f : \langle TG, G_{in}, P, \pi \rangle \rightarrow \langle TG', G'_{in}, P', \pi' \rangle$ between grammars over the same data algebra D is a pair $\langle f_{TG}, f_P \rangle$ where $f_{TG} : TG \rightarrow TG'$ is a retyping morphism and $f_P : P \rightarrow P'$ is a mapping of rule names such that

- $f_{TG}^>(G_{in}) = G'_{in}$
- For all $p \in P$, $\pi'(f_P(p)) = \langle \Phi_{L_p}, f_{TG}^>(L_p) \leftarrow f_{TG}^>(K_p) \hookrightarrow f_{TG}^>(R_p), \Phi_{R_p} \rangle$.

The category of attributed graph grammars and their morphisms is **AGG**.

A derivation in $\mathcal{AG}$ is a finite sequence of transformations $s = (G_{in} = G_0 \xrightarrow{p_1, m_1} \dots \xrightarrow{p_n, m_n} G_n)$ with $p_i \in P$. Since $f_{TG}^>$ preserves pushouts and rules spans and constraints remain the same but for typing, morphisms of attributed graph grammars preserve derivations. That is, a derivation s in $\mathcal{AG}$ is translated by f to a derivation $f(s) = (f_{TG}^>(G_{in}) = f_{TG}^>(G_0) \xrightarrow{f_P(p_1), f_{TG}^>(m_1)} \dots \xrightarrow{f_P(p_n), f_{TG}^>(m_n)} f_{TG}^>(G_n))$.

Given an attributed graph grammar $\mathcal{AG} = \langle TG, G_{in}, P, \pi \rangle$, its *underlying (unattributed) grammar* is $\mathcal{AG}_G = \langle TG_G, G_{in,G}, P, \pi_G \rangle$ where TG_G and $G_{in,G}$ are the restrictions of TG and G_{in} to their graph components (excluding variables and attribution edges) and for all $p \in P$ with $\pi(p) = \langle \Phi_{L_p}, L_p \leftarrow K_p \hookrightarrow R_p, \Phi_{R_p} \rangle$ their rules reduce to $\pi_G(p) = \langle \emptyset, (L_p)_G \leftarrow (K_p)_G \hookrightarrow (R_p)_G, \emptyset \rangle$. It is easy to see that a derivation s in $\mathcal{AG}$ reduces to a derivation s_G in the underlying grammar $\mathcal{G}_G$. Similarly, by $\mathcal{AG}_{EG}$ we denote the underlying E-graph grammar, retaining attribute links and variables, but dropping all constraints.

Example 3 (attributed graph grammar for escrow smart contract). The rules specifying the service are shown in Fig. 3 using an integrated notation merging left- and right hand side graphs into a single rule graph $L \cup R$. We indicate by colours and labels which elements are required but preserved (grey), required and deleted (red), or newly created (green). That means, the left-hand side L is given by all grey and red elements, the right-hand side by all grey and green ones, and the interface K by the grey elements only.

The service works as follows. The price is set by the seller using input parameter $p > 0$, which is recorded in the contract created as part of the sell operation.

This step also incurs the stake of $stakeFactor \times p$, which is transferred from the seller's wallet to the balance of the contract. If the seller changes their mind and withdraws the widget, the stake is refunded.

The buyer uses the buy operation to indicate their intend to buy at the price $c.price$ as listed in contract c while transferring the same stake as the seller. When entering into a contract, seller and buyer use up a token, modelled by a loop, used to control the number of contracts each user can engage in. This can either be returned when they leave the contract, allowing them to enter into a new one (in the current version) or it can be a single-use token (in a later variant). If the seller withdraws after the buyer entered into the contract, the buyer can leave with the stake returned, and then the contract is deleted. Note that, due to the dangling condition, leave is only applicable if the contract does

Fig. 3. Rules of the escrow model, in Henshin notation (Color figure online)

not have any further connections, in particular no selling link. That means it is only applicable after withdraw.

Once buyer and seller have entered the contract, the widget can be delivered. This leads to the transfer of the price from the buyer to the contract. The buyer can also change their mind and cancel either before delivery, in which case the buyer's and seller's stakes are refunded, or after delivery, e.g., if they are unsatisfied with the widget received. In the latter case they need to return the widget to get the price refunded. If the widget is to the buyer's satisfaction, they can acknowledge receipt leading to the resolution of the contract and the ultimate payment to the seller's wallet of the price and the refund of both stakes.

3 DPO Occurrence Graph Grammars and Unfolding

As a prerequisite to considering attributed grammars, in this section we review the theory of unfolding for typed DPO graph grammars [4]. In the rest of the paper we assume that all rules are *consuming*, i.e., their left-hand side morphism is not an isomorphism.

A grammar $\mathcal{G} = \langle TG, G_{in}, P, \pi \rangle$ is (*strongly*) *safe* if, for all H such that $G_{in} \Rightarrow^* H$, H has an injective typing morphism. Instance graphs with injective typing can be safely identified with the corresponding subgraphs of the type graph (thinking of injective morphisms as inclusions). Therefore, in particular, each graph $\langle G, t_G \rangle$ reachable in a safe grammar can be identified with the subgraph $t_G(G)$ of the type graph TG . For a safe grammar $\mathcal{G}$, the set of its *elements* is defined as $Elem(\mathcal{G}) = TG_E \cup TG_N \cup P$, assuming without loss of generality that the three sets are mutually disjoint.

Using a net-like language, we speak of *pre-set* $\bullet q$, *context* $\underline{q}$ and *post-set* $q^\bullet$ of a rule q , defined as the sets of element deleted, preserved, or created by the rule. Similarly for a node or edge x in TG we write $\bullet x$, $\underline{x}$ and $x^\bullet$ to denote the sets of rules which produce, preserve and consume x . The *causal relation* of a grammar $\mathcal{G}$ is the binary relation $<$ over $Elem(\mathcal{G})$ defined as the least transitive relation satisfying, for any node or edge x in TG and $q_1, q_2 \in P$: (1) if $x \in \bullet q_1$ then $x < q_1$; (2) if $x \in q_1^\bullet$ then $q_1 < x$; and (3) if $q_1^\bullet \cap q_2^\bullet \neq \emptyset$ then $q_1 < q_2$. As usual $\leq$ is the reflexive closure of $<$. Moreover, for $x \in Elem(\mathcal{G})$ we denote by $[x]$ the set of causes of x in P , namely $\{q \in P : q \leq x\}$.

The *asymmetric conflict relation* of $\mathcal{G}$ is the binary relation $\nearrow$ over P , given by (1) if $\underline{q_1} \cap \bullet q_2 \neq \emptyset$ then $q_1 \nearrow q_2$; (2) if $\bullet q_1 \cap \bullet q_2 \neq \emptyset$ and $q_1 \neq q_2$ then $q_1 \nearrow q_2$; and if $q_1 < q_2$ then $q_1 \nearrow q_2$.

A (nondeterministic) occurrence grammar is an acyclic grammar which represents, in a branching structure, several possible computations starting from its initial graph and using each rule at most once.

Definition 2 (occurrence grammar). An occurrence grammar is a safe graph grammar $\mathcal{O} = \langle TG, G_{in}, P, \pi \rangle$ where

1. for each rule $q : \langle L, t_L \rangle \xleftarrow{l} \langle K, t_K \rangle \xrightarrow{r} \langle R, t_R \rangle$, typing morphisms t_L, t_K, t_R are injective;

2. its causal relation $\leq$ is a partial order, and for any $q \in P$, the set $\lfloor q \rfloor$ is finite and asymmetric conflict $\nearrow$ is acyclic on $\lfloor q \rfloor$;
3. the initial graph G_{in} coincides with the set $Min(\mathcal{O})$ of minimal elements of $\langle Elem(\mathcal{O}), \leq \rangle$ (with the graphical structure inherited from TG and typed by the inclusion);
4. each arc or node x in TG is created by at most one rule in P : $|\bullet x| \leq 1$.

Fig. 4. Type graph, causality (solid) and asymmetric conflicts (dashed) of attributed occurrence grammar (Color figure online)

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

One can show that each occurrence grammar is safe [4].

Example 4 (occurrence graph grammar). An occurrence grammar based on the underlying grammar $\mathcal{AG}_G$ of the escrow model is shown in Fig. 4 with its type graph on the left and relations on the right. In the graph, all preserved (grey) elements are minimal, making up the input graph. The rules are the same as in Fig. 3. Matches and co-matches are defined by name, e.g., $c : Contract$ in any of the rules maps to $c : Contract$ in the type graph. This is possible here because we apply each rule exactly once. The causality and asymmetric conflict relations are justified below, with relations covered by transitive closure omitted.

- $sell < withdraw, buy$ because $sell$ creates the contract used by $withdraw, buy$;
- $buy < deliver, cancel, leave$ because buy creates the buying link used by $deliver$ and consumed by both $leave, cancel$; $buy \nearrow withdraw$ because buy uses the widget link deleted by $withdraw$;
- $withdraw \nearrow leave$ because $leave$ deletes the contract used by $withdraw$;
- $deliver < ack, return$ because $deliver$ creates the has link used by both $ack, return$; $deliver \nearrow withdraw, leave, cancel$ because $withdraw$ and $cancel$ both consume the selling link and $leave$ the contract used by $deliver$;
- $cancel \nearrow withdraw$ because $withdraw$ deletes the selling link used by $cancel$
- $cancel, leave, ack$ are in mutual conflict because all delete the buying link;

- $ack \nearrow return$ because $return$ deletes the has link used by ack ;
- $return \nearrow ack$ because ack deletes the contract used by $return$;

Note the lack of a causal dependency $withdraw < leave$; however due to the dangling condition $leave$ can only be applied after $withdraw$. For a similar reason, $leave$ cannot occur after $deliver$ because $sell, buy, deliver, leave$ violates the dangling condition due to the lack of $withdraw$ while $sell, buy, withdraw, deliver, leave$ is disallowed by $deliver \nearrow withdraw$. Such subsets of rules, that are not fully executable in any order, do not form *configurations*. Subsets of rules that do, represent reachable graphs.

Occurrence grammars can be created by unfolding consuming graph grammars. The idea is to start from the initial graph of the grammar, then apply in all possible ways its rules while recording each occurrence and each new graph item generated with their causal histories. The basic ingredient here is the gluing operation, that we borrow literally from [4].

By $glue_*(q, m, G)$ we represent the additive application of rule q at match m to graph G , generating new items as specified by the rule and labelling them by $*$, but not removing items that should have been deleted. This is because such items may still be used by another rule in the nondeterministic unfolding.

4 Attributed Occurrence Grammars

In this and the following section we lift the theory of occurrence grammars and unfolding from typed graph grammars to lazy typed symbolic attributed ones. On grounded graphs, lazy rules are equivalent to general symbolic rules. Limiting ourselves to ground-preserving rules and a grounded start graph, all derivations will be grounded, which will allow us to transfer the results to the general case, including non-lazy rules.

Definition 3 (safe attributed graph grammar). *An attributed graph grammar $\mathcal{AG} = \langle TG, P, G_{in}, \pi \rangle$ is safe if TG is linear and its underlying E-graph grammar $\mathcal{AG}_{EG}$ is safe (reachable graphs H have injective typing $t_H : H \rightarrow TG$).*

A reachable graphs H will be seen as subgraphs of TG , and hence $X_H \subseteq X_{TG}$ and $D \models (\Phi_{TG} \implies \Phi_H)$. That means, X_{TG} is a global set of variables for $\mathcal{AG}$. The set of elements of a safe grammar $\mathcal{AG}$ is $Elem(\mathcal{AG}) = P \cup TG_G \cup X_{TG}$, including the rules, graph objects and variables in its type graph, but not its attribute links. With this, the causal and asymmetric conflict relations $<$ and $\nearrow$ carry over to the rules and graph objects of $\mathcal{AG}$ via its underlying E-graph grammar $\mathcal{AG}_{EG}$. Applied to E-graphs, these relations include variable nodes (although variables are never deleted, so $\bullet q \cap X_{TG} = \emptyset$) but no attribution links. In particular, $x \in q^\bullet$ iff $x \in X_{R_q} \setminus X_{L_q}$ and $x \in \underline{q}$ iff $x \in X_{L_q} \cap X_{R_q}$.

Let's examine the impact on the causal and asymmetric conflict relations of omitting attribute links from context, pre- and post-sets. When an attribute is updated, its attribute link is deleted and a new one is created pointing to a new

variable. If two rules q_1, q_2 update the same attribute one after the other, q_1 creates the attribute link resulting in the first update and q_2 deletes the link. That means, in a causal relation including attribute links $q_1 < q_2$ based on clauses 1 and 2 in the definition of $<$. Now, without considering attribute links $q_1 < q_2$ by clause 3 because $v \in q_1^\bullet \cap q_2$. However, disregarding attribute links impacts on asymmetric conflicts because variable nodes are never deleted, so neither clause 1 nor 2 of $\nearrow$ apply. That means, in a structural sense, attribute updates never cause conflicts. This is significant because it allows to separate attribute computations from structural unfolding.

For $e \in \text{Elem}(\mathcal{AG})$ we let $[e] = \{q \in P \mid q \leq e\}$ and $[E]$ the extension to sets $E \subseteq \text{Elem}(\mathcal{AG})$. The set of constraints of $R \subseteq P$ is $\Phi(R) = \bigcup_{p \in R} (t_{L_p}(\Phi_{L_p}) \cup t_{R_p}(\Phi_{R_p}))$. Given $G \subseteq TG$, variable $x \in X_G$ is *current* in G if there is no rule $p \in [G]$ with $o.a = x \in L_p$ and $o.a = y \in R_p \setminus L_p$. Graph G is *current* if all variables in X_G are and every attribute has at most one variable as a label (i.e. cannot have both $o.a = y$ and $o.a = z$ with $y \neq z$ in the same graph G).

Definition 4 (attributed occurrence grammar). An attributed occurrence grammar is a safe attributed graph grammar $\mathcal{AG} = \langle TG, G_{in}, P, \pi \rangle$ where

1. for each rule q the typing morphisms t_{L_p} , t_{K_q} and t_{R_q} are injective
2. for all $q \in P$, $(\nearrow)_{[q]}$ is acyclic, L_q current, $[q]$ finite, and $\Phi([q] \cup \{q\}) \cup \Phi_{G_{in}}$ satisfiable
3. the initial graph G_{in} consists of the minimal elements $\text{Min}(\mathcal{AG})$ of $\langle \text{Elem}(\mathcal{AG}), \leq \rangle$ ³ with attribution and graph structure inherited from TG and typed by the inclusion
4. each element (arc, node, variable or attribution edge) e in TG is created by at most one rule in P : $|\bullet e| \leq 1$

The category of attributed occurrence grammars and AGG morphisms is **AOG**.

Given attributed occurrence grammar $\mathcal{AO}$, its underlying grammar $\mathcal{AO}_G$ is an occurrence grammar. In particular, $\mathcal{AO}_G$ is obtained from $\mathcal{AO}$ by dropping its variables, attribute links and constraints. Hence conditions (2–4) in Definition 4 reduce to (2–4) in Definition 2.

Example 5 (attributed occurrence grammar). Figure 5 shows the rules and causal relation of an attributed occurrence grammar extending that of Example 4. Dependencies caused by attribute updates (of the contract’s balance attribute) are shown as red dotted lines. Structurally, each such update reads the variable node representing the old value, creates a new variable node and changes the attribute link from the old to the new variable node. That means, subsequent rule applications access the new version of the variable node, which makes them different from applications of the same rules to the same graph objects accessing earlier versions of the attribute. Hence the duplication of rule occurrences in the

³ Notice that $\text{Min}(\mathcal{AG}) \subseteq N_{TG} \cup E_{TG}$, i.e., it does not contain rules, since the grammar is consuming.

dependency structure. All asymmetric conflicts carry over from the unattributed case, i.e., if $p \nearrow q$ in Example 4, then $p? \nearrow q?$ for all copies $p?, q?$ of p, q . As discussed earlier, there are no new conflicts due to attribute updates.

Furthermore, in any current graph every attribute is labelled by at most one variable. To clarify the importance of this condition, consider the branching from *cancel2* to *withdraw6* and *return2* both of which update attribute *c.balance*. The former substitutes it by $c.balance - c.stake$ and the latter by $c.balance - c.price$. Applying both rules in parallel yields $\{b_2 = b_1 - c.stake, b_3 = b_1 - c.price\} \subset \Phi([G])$ which is always satisfiable. However this result does not correspond to any sequence of rule applications since both updates of *c.balance* use b_1 as the previous value. By preventing an attribute from having more than one label in a current graph, we disallow multiple versions of the same attribute.

Fig. 5. Causality relation of attributed occurrence grammar (Color figure online)

5 Unfolding Attributed Graph Grammars

This section introduces the unfolding construction which, applied to an attributed graph grammar $\mathcal{AG}$, produces an attributed occurrence grammar $\mathcal{U}_{\mathcal{AG}}$ describing the behaviour of $\mathcal{AG}$. The unfolding is equipped with a morphism $u_{\mathcal{AG}}$ to the original grammar $\mathcal{AG}$ which allows to see rules in $\mathcal{U}_{\mathcal{AG}}$ as rule applications in $\mathcal{AG}$, and items of the type graph of $\mathcal{U}_{\mathcal{AG}}$ as instances of items of the type graph of $\mathcal{AG}$. Starting from the initial graph of the grammar, we apply in all possible ways its rules, and record in the unfolding each redex and new graph item generated with their causal histories. In order for these rule applications to reflect reachable transformations, we introduce attribute concurrent graphs representing reachable matches.

Definition 5 (attribute concurrent graph). Let $\mathcal{AO} = \langle TG, G_{in}, P, \pi \rangle$ be an attributed occurrence grammar. A subgraph G of TG is called attribute concurrent if

1. $\neg(x < y)$ for all $x, y \in G$;

2. G is current and $\lfloor G \rfloor$ finite;
3. $\nearrow$ is acyclic on $\lfloor G \rfloor$;
4. for all $e \in TG$ and $n \in \{s(e), t(e)\}$, if $n^\bullet \cap \lfloor G \rfloor \neq \emptyset$ and $\bullet e \subseteq \lfloor G \rfloor$ then $e^\bullet \cap \lfloor G \rfloor \neq \emptyset$;
5. $\Phi_{G_{in}} \cup \Phi(\lfloor G \rfloor)$ is satisfiable.

Proposition 2 (attribute concurrent graphs are coverable). *A graph $G \subseteq TG$ is attribute concurrent if and only if it is coverable in $\mathcal{AO}$, i.e. there is a derivation $\text{Min}(\mathcal{AO}) \Rightarrow^* H$ with $G \subseteq H$.*

For every concurrent graph G one can find a derivation $\text{Min}(\mathcal{O}) \Rightarrow^* H$ which applies exactly once every rule in $\lfloor G \rfloor$, in any order consistent with $(\nearrow_{\lfloor G \rfloor})^*$. Vice versa for each derivation $\text{Min}(\mathcal{O}) \Rightarrow^* G$ in $\mathcal{O}$, the set of rules it applies contains $\lfloor G \rfloor$ and their order is compatible with $\nearrow^*$. Therefore reachable graphs are concurrent. Furthermore, each subgraph of a concurrent graph is concurrent as well, thus so are all coverable graphs.

The proof is based on an analogous result for the un-attributed case, analysing the additional dependencies arising from attribute updates, and deriving the satisfiability of constraints added throughout derivations from that of $\Phi(\lfloor q \rfloor \cup \{q\}) \cup \Phi_{G_{in}}$ in Definition 4.2.

The unfolding of an attributed grammar $\mathcal{AG} = \langle TG, G_{in}, P, \pi \rangle$ is defined as follows. For each n , we construct a partial unfolding $\mathcal{U}(\mathcal{AG})^{(n)} = \langle \mathcal{U}^{(n)}, u^{(n)} \rangle$, where $\mathcal{U}^{(n)} = \langle TG^{(n)}, G_{in}, P^{(n)}, \pi^{(n)} \rangle$ is an attributed occurrence grammar and $u^{(n)} = \langle u_{TG}^{(n)}, u_P^{(n)} \rangle : \mathcal{U}^{(n)} \rightarrow \mathcal{AG}$ an AGG morphism. Intuitively, the occurrence grammar generated at level n contains all possible computations of the grammar with “causal depth” at most n .

- ($\mathbf{n} = \mathbf{0}$) $\langle \mathcal{U}^{(0)}, u^{(0)} \rangle$ is defined as $\mathcal{U}^{(0)} = \langle G_{in}, G_{in}, \emptyset, \emptyset \rangle$ with $u_{TG}^{(0)} = t_{G_{in}}$.
- ($\mathbf{n} \rightarrow \mathbf{n} + \mathbf{1}$) Given $\mathcal{U}(\mathcal{AG})^{(n)}$, the partial unfolding $\mathcal{U}(\mathcal{AG})^{(n+1)}$ is obtained by applying all enabled redexes $m : L_q \rightarrow \langle TG^{(n)}, u_{TG}^{(n)} \rangle$ of $q \in P$ in $TG^{(n)}$. Let $P_{(n+1)}$ be the set of all triples $q_i^* = \langle q_i, m_{i,G}, m_{i,X} \rangle$ such that $m(L_q) \subseteq TG^{(n)}$ is attribute concurrent and $m_i(\Phi_p)$ is satisfiable. Then, $\mathcal{U}(\mathcal{AG})^{(n+1)}$ is given by
 - $TG^{(n+1)} = glue_{q_k^*}(q_k, m_k, \dots glue_{q_1^*}(q_1, m_1, TG^{(n)}) \dots)$, the consecutive gluing of $TG^{(n)}$ with $R_{q_1} \dots R_{q_k}$ along $K_{q_1} \dots K_{q_k}$, respectively, with constraints $\Phi_{TG^{(n+1)}} = \Phi_{TG^{(n)}} \cup m_1(\Phi_{R_{q_1}}) \cup \dots \cup m_k(\Phi_{R_{q_k}})$;
 - morphism $u_{TG}^{(n)}$ extended canonically to $u_{TG}^{(n+1)} : TG^{(n+1)} \rightarrow TG$;
 - $P^{(n+1)} = P^{(n)} \cup \{ \langle q, m_G, m_X \rangle \mid \langle q, m \rangle \in P_{(n+1)} \}$;
 - $u_P^{(n+1)} = u_P^{(n)} \cup u_{P_{(n+1)}}$ where $u_{P_{(n+1)}}(\langle q, m_G, m_X \rangle) = q$;
 - $\pi^{(n+1)}(\langle q, m_G, m_X \rangle)$ coinciding with $\pi(q)$ except for retyping.

The result of the gluing operation is independent of the order of redexes applied.

The match of a rule, which is part of its name in the unfolding, records the causal history by way of the graph and data elements it maps to, which in turn are labelled by the name of the rule that created them. This allows to distinguish

rule occurrences with different histories. Note that if a rule is applied twice (also in different steps) with the same graphical match and variable mapping, the generated items are the same and thus they appear only once in the unfolding. By construction it is evident that $\mathcal{U}(\mathcal{AG})^{(n)} \subseteq \mathcal{U}(\mathcal{AG})^{(n+1)}$, component-wise.

Definition 6 (attributed unfolding). *The unfolding $\mathcal{U}(\mathcal{AG}) = \langle \mathcal{U}_{\mathcal{AG}}, u_{\mathcal{AG}} \rangle$ is defined as $\bigcup_n \mathcal{U}(\mathcal{AG})^{(n)}$, where union is applied componentwise.*

The folding morphism u is defined by mapping rules by their names, while on types $u_{TG}(x : X) = X$.

Example 6 (unfolding). In Example 5 we presented a partial unfolding of the grammar in Example 3. The set of rules $\{sell, buy, deliver, cancel, return\}$ admits two possible linearisation $sell, buy, deliver, cancel, return$ and $sell, buy, deliver, return, cancel$ that only differ in the order of applying $cancel$ and $return$. While the structural unfolding does not distinguish between the sequences, in the attributed unfolding they both update attributes $c.balance$ and $w2.balance$. Hence, whichever occurs second will depend on the first. The shared part of both linearisation is shown below, the two distinct parts in the following tables.

i	Rule	Rule application	$X^{(i)} \setminus X^{(i-1)}$	Constraint
1	<i>sell</i>	<i>sell1</i>	$\langle w1.balance, sell1 \rangle$ $\langle c.balance, sell1 \rangle$ $\langle c.price, sell1 \rangle$ $\langle c.stake, sell1 \rangle$	$\langle w1.balance, in \rangle - \langle p, in \rangle \times \langle e.stakeFactor, in \rangle$ $\langle p, in \rangle \times \langle e.stakeFactor, in \rangle$ $\langle p, in \rangle$ $\langle p, in \rangle \times \langle e.stakeFactor, in \rangle$
2	<i>buy</i>	<i>buy1</i>	$\langle w2.balance, buy1 \rangle$ $\langle c.balance, buy1 \rangle$	$\langle w2.balance, in \rangle - \langle c.stake, sell1 \rangle$ $\langle c.balance, sell1 \rangle + \langle c.stake, sell1 \rangle$
3	<i>deliver</i>	<i>deliver1</i>	$\langle c.balance, deliver1 \rangle$ $\langle w1.balance, deliver1 \rangle$ $\langle w2.balance, deliver1 \rangle$	$\langle c.balance, buy1 \rangle + \langle c.price, sell1 \rangle$ $\langle w1.balance, sell1 \rangle - \langle w.delivery, in \rangle$ $\langle w2.balance, buy1 \rangle - \langle c.price, sell1 \rangle$

i	Rule	Rule application	$X^{(i)} \setminus X^{(i-1)}$	Constraint
4	<i>cancel</i>	<i>cancel2</i>	$\langle c.balance, cancel2 \rangle$ $\langle w2.balance, cancel2 \rangle$	$\langle c.balance, deliver1 \rangle - \langle c.stake, sell1 \rangle$ $\langle w2.balance, deliver1 \rangle + \langle c.stake, sell1 \rangle$
5	<i>return</i>	<i>return2</i>	$\langle c.balance, return2 \rangle$ $\langle w2.balance, return2 \rangle$	$\langle c.balance, cancel2 \rangle - \langle c.price, sell1 \rangle$ $\langle w2.balance, cancel2 \rangle + \langle c.price, sell1 \rangle$

Through substitution we find that in both linearisation the end result is the same, i.e., both $c.balance$ and $w2.balance$ are equal to their initial values. However, the causal dependencies are different due to different data flow:

The two paths are confluent (produce the same result) but not equivalent (do not represent the same concurrent computation). For applications where we are

i	Rule	Rule application	$X^{(i)} \setminus X^{(i-1)}$	Formula
4	<i>return</i>	<i>return1</i>	$\langle c.balance, return1 \rangle$ $\langle w2.balance, return1 \rangle$	$\langle c.balance, deliver1 \rangle - \langle c.stake, sell1 \rangle$ $\langle w2.balance, deliver1 \rangle + \langle c.price, sell1 \rangle$
5	<i>cancel</i>	<i>cancel3</i>	$\langle c.balance, cancel3 \rangle$ $\langle w2.balance, cancel3 \rangle$	$\langle c.balance, return1 \rangle - \langle c.stake, sell1 \rangle$ $\langle w2.balance, return1 \rangle + \langle c.stake, sell1 \rangle$

cancel before return		return before cancel	
<i>cancel2</i>	$< return2$	<i>return1</i>	$< cancel3$
$\langle c.balance, cancel2 \rangle$	$< \langle c.balance, return2 \rangle$	$\langle c.balance, return1 \rangle$	$< \langle c.balance, cancel3 \rangle$
$\langle w2.balance, cancel2 \rangle$	$< \langle w2.balance, return2 \rangle$	$\langle w2.balance, return1 \rangle$	$< \langle w2.balance, cancel3 \rangle$

only interested in the end results of non-deterministic concurrent computations, it makes sense to explore whether confluence can be used to restrict the unfolding construction, avoiding exploration of confluent alternatives.

Proposition 3 (attributed unfolding). *$\mathcal{U}_{AG}$ is an attributed occurrence grammar and $u_{AG} : \mathcal{U}_{AG} \rightarrow \mathcal{AG}$ is an attributed graph grammar morphism.*

Since AGG morphisms preserve derivations, this implies the correctness of the unfolding, i.e., u_G maps all derivations in $\mathcal{U}_{AG}$ to derivations in $\mathcal{AG}$. The construction can be lifted to a functor between attributed grammars and occurrence grammars which is a right adjoint to the inclusion functor.

Theorem 1 (coreflection). *Unfolding is a coreflection $\mathcal{U} : \mathbf{AGG} \rightarrow \mathbf{AOG}$, that is, right-adjoint to the inclusion of $\mathbf{AOG}$ in $\mathbf{AGG}$.*

The proof established the unfolding as a co-free construction from attributed grammars to attributed occurrence grammars. That means, the unfolding is maximal among all attributed occurrence grammars equipped with a morphism back to the original grammar, in the sense that there exists a unique commuting morphisms from every other candidate. The existence of this morphism shows that the unfolding is complete in representing all possible behaviours present in the given grammar. Uniqueness shows that they are represented in a minimal way, so there is only one choice as to how the morphism can be defined.

6 Related Work and Conclusion

We presented occurrence grammars and unfoldings for attributed graph grammars. Since unfolding is a coreflection, the semantics is correct (only representing derivations specified by the grammar), complete (representing all such derivations) and fully abstract (represented in a minimal way). Our approach uses symbolic graph transformation [8,9] and the theory of unfolding of graph grammars [1,2], in turn generalising results for Petri Nets [6]. [7] uses approximated unfolding and abstract interpretation to analyse attributed graph grammars.

While their focus is on scalable analysis with numerical data, rather than comprehensive semantics, we share the objective of separating structural from attribute computations.

We will use the relation of occurrence grammars and event structures to develop, based on Rideau and Winskel's concurrent games as event structures [10], a notion of concurrent game over attributed graph grammars, with applications to analysing smart contracts. We also plan to integrate the theories of unfolding for attributed and conditional grammars.

Acknowledgement. We would like to thank the authors of [2] for their support and advice.

References

1. Baldan, P., Corradini, A., Heindel, T., König, B., Sobociński, P.: Unfolding grammars in adhesive categories. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 350–366. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03741-2_24
2. Baldan, P., Corradini, A., König, B.: Unfolding graph transformation systems: theory and applications to verification. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 16–36. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68679-8_3
3. Baldan, P., Corradini, A., Montanari, U., Ribeiro, L.: Coreflective concurrent semantics for single-pushout graph grammars. In: Wirsing, M., Pattinson, D., Henicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 165–184. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_9
4. Baldan, P., Corradini, A., Montanari, U., Ribeiro, L.: Unfolding semantics of graph transformation. *Inf. Comput.* **205**(5), 733–782 (2007). <https://doi.org/10.1016/j.ic.2006.11.004>
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006). <https://doi.org/10.1007/3-540-31188-2>
6. Esparza, J., Heljanko, K.: Unfoldings: A Partial-Order Approach to Model Checking. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2008). <https://doi.org/10.1007/978-3-540-77426-6>
7. König, B., Kozioura, V.: Towards the verification of attributed graph transformation systems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 305–320. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_21
8. Orejas, F., Lambers, L.: Symbolic attributed graphs for attributed graph transformation. *ECEASST* **30** (2010). <http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/405>
9. Orejas, F., Lambers, L.: Lazy graph transformation. *Fundam. Inform.* **118**(1–2), 65–96 (2012). <https://doi.org/10.3233/FI-2012-706>
10. Rideau, S., Winskel, G.: Concurrent strategies. In: 2011 IEEE 26th Annual Symposium on Logic in Computer Science, pp. 409–418. IEEE (2011)