Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 324 (2016) 15–30

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Translating Controlled Graph Grammars to Ordinary Graph Grammars

Alex Bertei[1] Luciana Foss[2] Simone A. da Costa Cavalheiro[3]

Technology Development Center (CDTec)
Federal University of Pelotas
Pelotas, Brazil

Abstract

Graph Grammar (GG) is an appropriate formal language for specifying complex systems. In a GG the system states are represented by graphs and the changes between the states are described by rules. The use of GGs is interesting as there are several techniques for the specification and verification of systems that are described in this language. Besides this, GGs have a graphical representation which is quite intuitive, making the language easy to understand even for non-theorists. Controlled graph grammars (CGGs) are GGs that allow defining a sequence of rule applications, considering an auxiliary control structure, allowing one to control the intrinsic non-determinism of this formalism. However, for this type of grammar there are no tools for verification of properties. Therefore, the aim of this paper is to translate the controlled graph grammars into ordinary graph grammars, transferring the flow control from an auxiliary structure to the state. Hence, the main contribution of this paper is to permit the use of Rodin theorem prover to carry out the verification of properties for CGG specifications. This is possible through a mechanism where a controlled graph grammar is translated into a regular graph grammar using dependencies and conflicts between rules.

Keywords: Graph grammar, controlled graph grammar, dependent and conflicting rules

1 Introduction

Software and hardware systems are found everywhere and each day we face more complex and sophisticated systems. The development of systems has been a hard task to be performed, making it necessary a complete specification with no mistakes. Each day, these systems grow in scale and scope, often having to interact with other complex and independent environments. Along this increase of complexity, the possibility of mistakes is intensified which can lead to catastrophic losses, either in terms of time, money or even lives. Therefore the techniques to assist the development of reliable and correct systems are becoming more and more necessary [1]. One

[1] Email: abertei@inf.ufpel.edu.br
[2] Email: lfoss@inf.ufpel.edu.br
[3] Email: simone.costa@inf.ufpel.edu.br

http://dx.doi.org/10.1016/j.entcs.2016.09.004

of the ways to reach such a goal is through the use of formal methods, techniques based in mathematical formalisms which can offer strict and efficient measures to project, model and analyze computer systems [2]. A specification must be compact, accurate and with no ambiguity, that is, it must be given through a language with well-defined syntax and semantics, which use mathematical concepts. These concepts are important as they enable stating if a computational system presents a certain property or fulfills its specification [3].

In the last decade several case studies and industrial applications confirmed the significant importance of the use of formal methods to improve the quality of both hardware and software systems. However, the high cost to use formal methods causes them, in a general way, to be used only in the development of high integrity systems, where there is a high probability that the mistakes result in the loss of lives or serious damages. Well defined specifications, verified with respect to critical properties, have provided a basis for a correct and efficient source code generation. An outstanding example is the Paris metro system [4]. The system is fully automatic and had its critical safety parts formally developed by Matra Transport International using the B method [5].

Graph grammar (GG) is a formal language which has been studied since the 1970s [6] and applied at Computer Science areas which require dynamic graphs models, that is, graphs which can suffer transformations or manipulations in their structure. GG is a visual language which permits formally specifying and verifying systems with complex characteristics such as parallelism, concurrence and distribution. Besides this, there are different techniques and tools which permit the usage of model checking [7][8][9][10] and theorem proving [11][12][13], for the analysis of properties of systems described in this language. In a graph grammar, the states of the system are represented by graphs and the behavior or the changes of states are defined by graph transformation rules. This language permits the specification of formal languages, pattern recognition, image recognition and generation, construction of compilers, modeling of concurrent and distributed systems, software engineering, database projects, among others [15]. In the usual graph grammar approaches, the sequence of rule applications is not defined by a control structure but by the resources (data) available in the present state.

Controlled graph grammars, on the other hand, permit defining a sequence in the rule applications, not taking into account only the state but also an auxiliary control structure, as, for example, regular expressions or state machines. However, there are no tools which enable the formal verification of properties for this type of grammar. In [11] a relational approach for GGs was proposed allowing the use of theorem prover of the Rodin tool [14]. The tool makes static (syntactic) and dynamic verifications, using the Event-B language. Thus, this paper presents a proposal to integrate controlled graph grammars in the technique of theorem proving for GGs, allowing then the analysis of properties of systems specified using GGs with control structures for the application of rules.

In this paper we consider graph grammars in which control is given through regular expressions. The restrictions imposed by the regular expressions (RE) can

not be directly translated to the Event-B language. Such impossibility is due to the fact that there is no kind of control structure on the Event-B allowing to define the execution order of the events. An event is enabled whenever its guards are fulfilled. Then, the control structures imposed by the REs are transferred to the data. In order to do this, a translation from CGGs to GGs was defined, where the sequence of rule applications is given by conflicts and dependencies between these rules.

2 Graph Grammars

Graph grammars (GGs) have been used to specify various types of software systems [15] , where graphs are used to represent states and graph transformation rules for operations or state changes of the system. Since graph grammars are at the same time, intuitive and formal, and permit to deal with aspects of concurrence and distribution in a simple way, they become a promising method for reliable software development.

GG is a formal language suitable for specifying a wide range of computational systems. It is a visual language, which makes it quite intuitive and allows the easy understanding of the model even for the non theorists. GGs generalize Chomsky grammars using graphs instead of strings.

Definition 2.1 [Graph and graph morphism] A graph $G = (V_G, E_G, o^G, d^G)$, consists of a set of vertices V_G, a set of edges E_G, source and target functions o^G, $d^G : E_G \to V_G$. A (partial) graph morphism $g : G \to H$ from a graph G to a Graph H is a tuple $g = (g_V, g_E)$ consisting of two partial functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ which are weakly homomorphic, i. e. $g_V \circ o^G \geq o^H \circ g_E$ and $g_V \circ d^G \geq d^H \circ g_E$. These restrictions can be defined by the weak commutativity of the diagrams on the right. A morphism g is called total or injective if both components are total or injective, respectively.

$$\begin{array}{ccc} E_G & \xrightarrow{g_E} & E_H \\ {\scriptstyle o^G}\downarrow & \geq & \downarrow{\scriptstyle o^H} \\ V_G & \xrightarrow[g_V]{} & V_H \end{array} \qquad \begin{array}{ccc} E_G & \xrightarrow{g_E} & E_H \\ {\scriptstyle d^G}\downarrow & \geq & \downarrow{\scriptstyle d^H} \\ V_G & \xrightarrow[g_V]{} & V_H \end{array}$$

Instead of using simple graphs which only have vertices and edges, some typing mechanism in the graphs is generally used. The graph typing can be done through labels, attributes or type graph. In the case of this paper, the typing is provided by a type graph. A type graph of a grammar characterizes all the types of vertices and edges allowed in the system, and all graphs of the grammar are restricted to these types. This restriction is defined by a graph morphism mapping each graph in the type graph.

Definition 2.2 [Typed graph and typed graph morphism] A typed graph G^T is a tuple $G^T = (G, t^G, T)$, where G and T are graphs and t^G: $G \to T$ is a total graph morphism called typing morphism. A typed graph morphism between graphs G^T and H^T with type graph T is a morphism $g : G \to H$ such that $t^G \geq t^H \circ g$ (that is, g may only map elements of the same type).

In a graph grammar, the states of the system are represented by graphs and the possible changes of states are described by rules. A rule defines elements that

should be present in the graph so that it can be applied and the changes are made by its application, where some elements are eliminated and others are created. The behavior of a graph transformation system is determined by the application of rewriting rules, also called graph productions [15]. Following the Double Pushout approach (DPO) [15], a rule is composed by three graphs: the left-hand side L, the right-hand side R, and interface K which represents elements that L and R have in common. It specifies that, once an occurrence of the graph L is found in the current state, it can be replaced by the graph R, preserving K.

Definition 2.3 [Rule] A T-typed (graph) rule is a tuple $q : L_q \xleftarrow{l_q} K_q \xrightarrow{r_q} R_q$ where q is the name of the rule, L_q, K_q and R_q are T-typed graph, l_q is an inclusion and r_q is an injective morphism. The class of all T-typed graph rules is denoted by $Rule(T)$.

In general, a graph grammar describes a system which is composed by a type graph, that characterizes the types of vertices and edges allowed in the system; an initial graph, that represents the initial state of the system; and a set of rules, which describe the possible state changes that may occur in the system. Besides this, the rules can have associated names. To do this, a set of rule names is defined and each name is associated to a rule by a function.

Definition 2.4 [Graph grammar] A (typed) graph grammar is a tuple $GG = (T, I, P, \pi)$, such that T is a type graph (the type of the grammar), I is a graph typed over T (the initial graph of the grammar), P is a set of rule names and $\pi : P \to Rule(T)$ is a total function that associates each name in P to a rule typed over T.

Example 2.5 An example is presented to illustrate a graph grammar specification. This example describes a system in which a supermarket customer makes their purchase and make the payment. Besides, the customer may have to get in line in order to accomplish the payment. Figures 1 and 2 present type and initial graphs, respectively. Figure 3 shows the set of rules of the $SuperMarket$ graph grammar. For reasons of space K graphs in Figure 3 have been omitted, but they can be reconstructed by the intersection of L and R. All graphs of the specified system can only have vertices and edges with types that appear in the type graph.

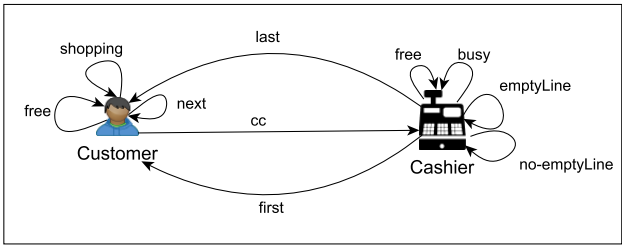


Fig. 1. Type graph of the $SuperMarket$ graph grammar

The type graph has two types of vertices, one represents the customers and the other represents the cashiers. The $free$, $shopping$ and $next$ edges have as source

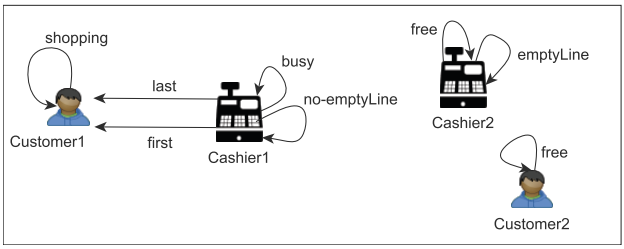


Fig. 2. Initial graph of the *SuperMarket* graph grammar

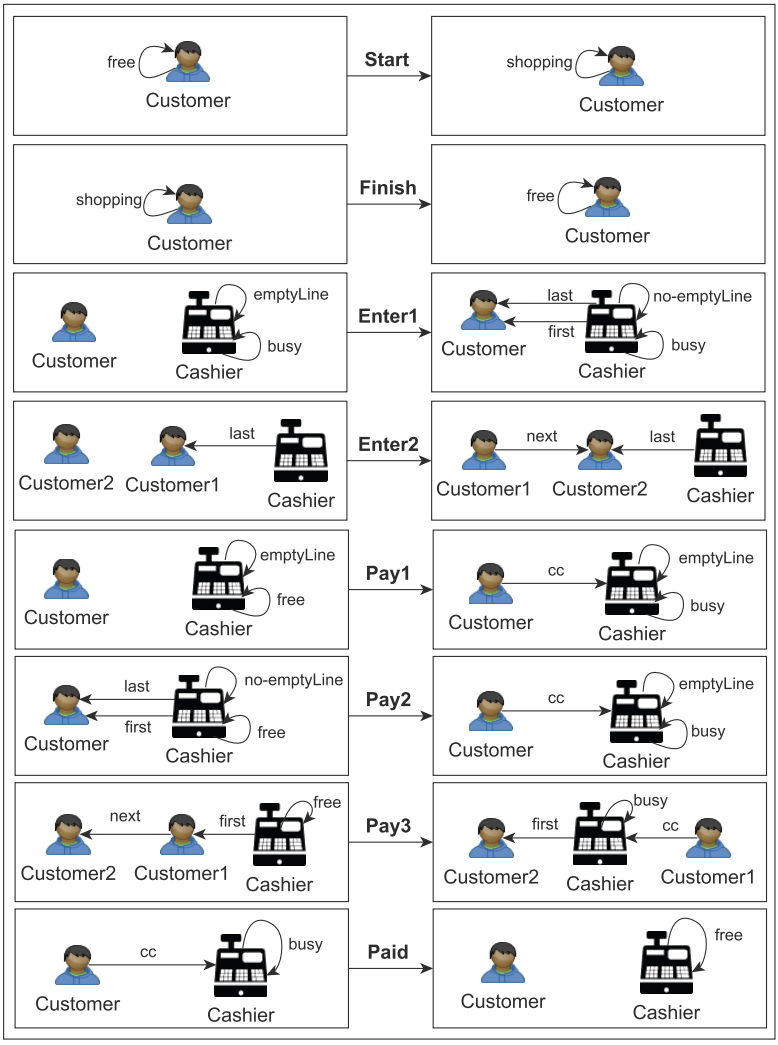


Fig. 3. Rules of the *SuperMarket* graph grammar

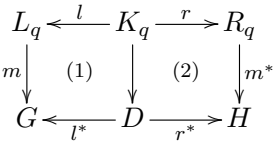
and target *Customer* vertices, where the first two describe the customer status and the last connects the customer to a cashier line. The *Cashier* vertex has loop edges *free*, *busy*, *emptyLine* and *no-emptyLine* which describe the cashier and the line status. The *first* and *last* edges, which have source in the *Cashier* vertex and target in the *Customer* vertex, indicate the first and last positions in

the cashier line, respectively. The initial graph describes the initial configuration of the $SuperMarket$ system, which is composed by two cashiers and two customers. One cashier ($Cashier2$) is free and has no line; the other one is busy, with only one customer ($Customer1$) in the respective line (then this customer is the first and the last one in the line). The other customer is free and can start shopping at anytime. The rules describe the customers behavior. Arriving at the supermarket, the customer starts shopping ($Start$ rule). Eventually, he can go to a cashier. If there is a line in the cashier, the customer must enter in the end ($Enter1$ and $Enter2$ rules), otherwise he is immediately served and the cashier change its status to busy ($Pay1$ rule). When the cashier is free and there is a customer at the beginning of the line, this customer leaves the line and is served, the connections and status of the line are rearranged and the status cashier is changed to busy ($Pay2$ and $Pay3$ rules). The customer leaves the cashier after paying his purchases, and the cashier becomes free ($Paid$ rule). When the customer ends shopping he changes his status to $free$ ($Finish$ rule).

The behavior of a system specified using a graph grammar can be described by applications of the graph grammar rules in graphs which represent the system states. Thus, the semantic is given by a set of graphs which derive from the initial graph. The application of a rule to a graph (derivation step) is enabled as long as there is an occurrence of the left-hand side of a rule in the present state graph, that is, if there is a total graph morphism mapping the left-hand side of the rule into the state graph.

An application of a production is given by a direct derivation in the double pushout approach. In this approach, a direct derivation is defined by two pushouts: the first deletes all items that shall be consumed and the second includes all items that shall be created. Intuitively, the pushout of two graphs with respect to another one, called interface graph, is given by the gluing of these two graphs together, identifying the items in the interface. A derivation is a finite or infinite sequence of direct derivations where the final graph of one is the start graph of another.

Definition 2.6 [Direct derivation] Given a T-typed G, a T-typed graph rule $q : L_q \longleftarrow K_q \longrightarrow R_q$ and a match (i.e. an injective T-typed graph morphism) $m : L_q \to G$, a direct derivation from G to H using q (based on m) exists iff the diagram on the right can be constructed, where both squares are pushouts in $T-Graph$ (the category of graphs and graph morphisms). In this case the direct derivation is denoted by $\delta : G \overset{q,m}{\Longrightarrow} H$ or $\delta : G \overset{q}{\Longrightarrow} H$ if we do not make explicit m.



The construction of the diagram above depends on the existence of D, called pushout complement. To ensure this existence, the match m must satisfy the gluing condition with respect to l. This condition is divided into two parts: the dangling condition, i.e., if a vertex is deleted, there can be no edges arriving or leaving from this vertex; and the identification condition, i.e., two vertices can be identified by m only if they are preserved.

Definition 2.7 [Dangling condition] Let $l : K \to L$ and $m : L \to G$ be two graph morphisms, where m is injective. Then there exists a pushout complement $\langle D,k : K \to D,l^* : D \to G\rangle$ of $\langle l, m\rangle$ iff the dangling condition is satisfied, i.e.: no edge $e \in E_G - m_E(E_L)$ is incident to any vertex in $g_v(V_L - l_v e(V_K))$.

The rules can be applied sequentially or in parallel. The sequential semantic of a graph grammar is provided by all the sequences of derivation steps using the GG rules, where each final graph of a step is the initial graph of the next.

If there is a rule that create some item that is needed by another one it is possible to consider that it causes this second rule. This stresses the idea that a cause provides the necessary condition for some action. In addition, because some rules can be applied simultaneously (in parallel), it is necessary that these rules are not in conflict, that is, one rule cannot delete some item used by the other rules. The following definition review causal dependency and conflict relations.

Definition 2.8 [(Causal) dependency and conflict relations] A rule p is a (direct) cause of a rule q if p creates some item that is needed (preserved/deleted) by q. This implies that q can only happen after p has happened.
A rule p is in conflict with a q rule if p deletes some item that is needed (preserved/deleted) by q. This implies that q can not happen if p has happened (p excludes the occurrence of q).

3 Flow Control in Graph Grammars

Formalisms which permit specifying graph transformations based on rules are usually not deterministic, that is, when there are several rules that can be applied at the same time (that is, there are several occurrences of the grammar rules in a state graph) the rule to be applied is chosen in a non-deterministic way.

This non-determinism can arise for two reasons: there may be more than one rule that can be applied to a given graph; or, a rule can be applied to different parts of the graph. Thus, in order to set or specify the transformation of a graph, it is often desirable to regulate this process. For example, choosing the rules according to priorities, or defining a particular sequence of steps [10].

Control conditions can be used to restrict the non-determinism of the rule applications of a graph grammar without taking into account the state but an auxiliary control structure. A typical example is to permit only derivations where the allowed rule application sequences belongs to a special control language. Regular expressions (REs) can be used to specify these conditions, that is, they are useful to prescribe the sequence in which the rules must be applied.

Here, we consider the class of regular expressions EXP over vocabulary Σ which are defined by $\varepsilon \in EXP$, $\Sigma \subseteq EXP$ and $(E_1; E_2)$, $(E_1 + E_2)$, $(E_1)^*$, $\in EXP$, if E_1,$E_2 \in EXP$. In order to omit parenthesis, assume the following precedence: first $*$, then ; and finally $+$. Intuitively, the expression ε describes the application of no rule, $p \in \Sigma$ specifies application of rule p, exactly once, $(E_1; E_2)$ describes the application of E_1 followed by the application of E_2, $(E_1 + E_2)$ specifies the non-deterministic choice

between E_1 and E_2; and $(E_1)^*$ is the arbitrary, but finite, consecutive iteration of E_1. The language denoted by a regular expression E is denoted by $\mathcal{L}(e)$ and contains all the words that satisfy the "scheme" defined by RE E. The path labels of a derivation ρ:$H_0 \Rightarrow^{p_1,m_1} H_1 \Rightarrow^{p_2,m_2} ... \Rightarrow^{p_n,m_n} H_n$, denoted by $path(\rho)$, is given by $p_1; p_2; ...; p_n$.

Controlled graph grammars are defined by a pair $CGG = (G, exp)$, where G is a graph grammar and exp is a set of regular expressions over the set of rules of the G grammar.

Definition 3.1 [Controlled graph grammar] A controlled graph grammar is given by a pair CGG= (GG,exp), where $GG = (T, I, P, \pi)$ is a graph grammar and Exp = $\{e|e \in EXP\}$ is a set of regular expressions over P. The set of all derivations of CGG, denoted by $DerC(CGG)$, is defined by the set of all derivations $\rho \in Der(GG)$, where $path(\rho) \in \mathcal{L}(e)$, for all e $\in Exp$.

Example 3.2 In the $SuperMarket$ graph grammar described in Example 2.5, an undesirable behavior of a customer, but allowed by the GG is: the customer can pick up goods and leave without paying (applying first the $Start$ rule and then the $Finish$ rule). In order to avoid this situation, a control structure can be added to permit only desirable behaviors. Using the regular expression $(Start; (Pay1 + (Enter1 + Enter2); (Pay2 + Pay3)); Paid; Finish)^*$ the customer must start to shopping ($Start$), go to a cashier $(Pay1+(Enter1+Enter2); (Pay2+Pay3))$, make the payment ($Paid$) and then he can leave the supermarket ($Finish$). This sequence of steps (rule applications) can be indefinitely applied. In this controlled graph grammar, it is clear that the customer will not be able to leave the supermarket without passing into some cashier and pay for its purchases.

3.1 Translation

Since the restrictions imposed by the REs cannot be translated to a control structure of the Event-B, the control structures imposed by the REs are transferred to the data. In order to do this, a translation of the CGGs to the GGs was defined by adding dependencies and conflicts in the rules of the grammar. These dependencies and conflicts are necessary to keep the order of the application of the rules that are defined by the REs. In order to introduce dependencies and conflicts, the right- and the left-hand side of the rules and the initial graph are changed, adding new vertices. Each added vertex must be new (it must be from a type which does not exist yet), since it cannot cause side effects, that is, it cannot create other dependencies or conflicts besides the expected ones. All new vertices must also be added in the type graph of the new grammar. Since a rule can appear several times in a RE and have different order restrictions, there can be several instances of the same rule, where each one will have different added dependencies and conflicts.

Definition 3.3 Given a controlled graph grammar $CGG = ((T_C, I_C, P_C, \pi_C), exp)$ and a regular expression $e = \sum_{e_i \in exp} e_i$, an equivalente graph grammar $GG = (T, I, P, \pi)$ is obtained as follows:

(1) $(e_1, P_1, \pi_1, T_1) = depRules(e, P_C, \pi_C, T_C)$;

(2) $(e_2, P_2, \pi_2, T, I) = depState(e_1, P_1, \pi_1, T_1, I_C)$;

(3) $P = rules(e_2)$;
$\pi = rules(e_2) \lhd \pi_2$ [4];

The construction of grammar GG is defined in three steps. In the first step (1), the dependencies and conflicts between rules are introduced; in the second step (2) conflicts between the initial rules are defined, by adding elements in the initial graph, and finally, (3) the set of rules is obtained with the new created rules. In the next subsections these steps are detailed.

For a RE e, over a set of rule names, the set of initial rules of e is defined by $initial(e) = \{v | w \in \mathcal{L}(e) \ \wedge v \in prefix(w) \ \wedge |v| = 1\}$. In a similar way, the set of final rules of e is defined by $final(e) = \{v | w \in \mathcal{L}(e) \ \wedge v \in suffix(w) \ \wedge |v| = 1\}$ [5].

3.2 Adding dependencies and conflicts between rules

In the first step, the rules and type graph are updated including new vertices to define dependencies and conflicts to reflect the application sequence defined by e. The resulting set of rules contains all original rules and one new version for each changed rule. The changed rules will receive new names, then the regular expression e is also updated to change the names of the rules. The functions $depRules(e, P_C, \pi_C, T_C)$ define the construction of these new elements.

Definition 3.4 The function $depRules : EXP \times P^2 \times (P \to Rule(T)) \times Graphs \to EXP \times P^2 \times (P \to Rule(T)) \times Graphs$ that adds dependencies/conflicts between rules by is defined by:
$depRules(e, P, \pi, T) =$

[4] Restriction of Domain: $S \lhd r = \{x \mapsto y | x \mapsto y \in r \wedge x \in S\}$, where S is a set and r is a relation.

[5] Given a word w, the $prefix(w)$ and $sufix(w)$, denote the sets of all prefixes and sufixes of w, respectively; and $|w|$ denote the length of w

$$
\begin{cases}
(e, P, \pi, T), & \text{if } e = r, r \in P \\
(e, P', \pi', T), & \text{if } e = \varepsilon, \\
 & P' = P \cup \varepsilon, \quad \pi' = \pi \cup \{\varepsilon \mapsto \emptyset \leftarrow \emptyset \rightarrow \emptyset\} \\
(s' + t', P_t, \pi_t, T_t), & \text{if } e = s + t, \text{ where} \\
 & (s', P_s, \pi_s, T_s) = depRules(s, P, \pi, T) \\
 & (t', P_t, \pi_t, T_t) = depRules(t, P_s, \pi_s, T_s) \\
(s''; t'', P', \pi', T') & \text{if } e = s; t, \text{ where} \\
 & (s', P_s, \pi_s, T_s) = depRules(s, P, \pi, T) \\
 & (t', P_t, \pi_t, T_t) = depRules(t, P_s, \pi_s, T_s) \\
 & x \notin V_{T_t}, \quad T' = (V_{T_t} \cup \{x\}, E_{T_t}, o^{T_t}, d^{T_t}) \\
 & (s'', P'_s, \pi'_s) = right(s', P_t, \pi_t, x) \\
 & (t'', P'_t, \pi'_t) = left(t', P_t, \pi_t, x) \\
 & P' = P_t \cup P'_t \cup P'_s, \quad \pi' = \pi_t \cup \pi'_t \cup \pi'_s \\
(s'; P', \pi', T') & \text{if } e = s^*, \text{ where} \\
 & (e_s, P_s, \pi_s, T_s) = depRules(s, P, \pi, T) \\
 & x \notin V_{T_s}, \quad T' = \{V_{T_s} \cup \{x\}, E_{T_s}, o^{T_s}, d^{T_s}\} \\
 & (e'_s, P'_s, \pi'_s) = \overline{right}(e_s, P_s, \pi_s, x) \\
 & (s', P''_s, \pi''_s) = \overline{left}(e'_s, P'_s, \pi'_s, x) \\
 & P' = P_s \cup P'_s \cup P''_s, \quad \pi' = \pi_s \cup \pi'_s \cup \pi''_s
\end{cases}
$$

In order to add the dependencies, the regular expressions are decomposed in sub-expressions and in each one of the changes are made whenever necessary, as not all operations of the REs demand the addition of dependencies and conflicts. A RE e can be of one the following types: ε (empty), rule (atomic), concatenation, choice or closure. The types that require the addition of dependencies/conflicts are concatenation and closure. So for an atomic RE r nothing should be changed. For an empty RE ε, the empty rule (a rule that does not delete, preserve or create anything) is added to the set of rules. For a choice, only the changes of its sub expressions should be reflected. For the concatenations besides reflecting the changes of each sub expression, it is also necessary to add the dependency created by this operation, where for each final rule of the first sub expression a new vertex is added in the right-hand side. Moreover, for each initial rule of the second sub expression the same vertex is added to the left-hand side. Finally, for a closure s^* the changes of s must be reflected and the dependencies must be added. This operation introduce a circular dependency that is defined by adding a new vertex in the left- and right-hand sides of each initial and final rules of s, respectively. Moreover, the initial and final rules must remain in order to allow to interrupt the cycle.

The $right$ ($left$) function adds a given vertex in the right(left)-hand side of the final (initial) rules of a given regular expression, defining new rules. The regular expression must be updated with the names of the new rules in order to reflect the dependencies of sub-expressions. In addition, this expression shows which the rules

should be kept in the final graph grammar. This information is necessary since during the process of creating dependencies, the original rules are kept (because they can appear more than once in the expression) and at the end of process the non used rules should be discarded (third step).

Considering a typed graph G^T and a new vertex x:

- $addV(G^T, x) = ((V_G \cup \{x\}, E_G, o^G, d^G), (t_{V_G} \cup \{x \mapsto x\}, t_{E_G}), T \cup \{x\})$;
- $extType(G^T, x) = (G, t^G, T \cup \{x\})$.

Definition 3.5 The functions $right, left : EXP \times P^2 \times (P \to Rule(T)) \times Set \to EXP \times P^2 \times (P \to Rule(T'))$ are defined by:

$right(e, P, \pi, x) = (e', P', \pi')$, where:

$P' = \emptyset; \quad \pi' = \emptyset; \quad e' = e;$

$\forall p \in final(e)$, with $i \in \mathbb{N} \wedge isnew(P \cup P', p_i)$

$P' = P' \cup \{p_i\}$

$R_{p_i} = addV(R_p, x);$

$K_{p_i} = extType(K_p, x);$

$L_{p_i} = extType(L_p, x)$

$\pi' = \pi \cup \{p_i \mapsto (L_{p_i} \leftarrow K_{p_i} \to R_{p_i})\}$

$e' = upFinalNames(e', p_i)$

$left(e, P, \pi, x) = (e', P', \pi')$, where:

$P' = \emptyset; \quad \pi' = \emptyset; \quad e' = e;$

$\forall p \in initial(e)$, with $i \in \mathbb{N} \wedge isnew(P \cup P', p_i)$

$P' = P' \cup \{p_i\}$

$L_{p_i} = addV(L_p, x);\ K_{p_i} = extType(K_p, x);\ R_{p_i} = extType(R_p, x)$

$\pi' = \pi \cup \{p_i \mapsto (L_{p_i} \leftarrow K_{p_i} \to R_{p_i})\}$

$e' = upInitialNames(e', p_i)$

The $upFinalNames$ and $upInitialNames$ functions are used to modify the regular expression in order to update the names of the new rules.

Definition 3.6 The functions $upInitialNames, upFinalNames : EXP \times P \to EXP$ are defined by:

$upInitialNames(e, p_i) =$

$$\begin{cases} (p + p_i) & \text{if } e = p \\ upInitialNames(s, p_i); t & \text{if } e = s; t \\ upInitialNames(s, p_i) + upInitialNames(t, p_i) & \text{if } e = s + t \end{cases}$$

$$upFinalNames(e, p_i) = \begin{cases} (p + p_i) & \text{if } e = p \\ s; upFinalNames(t, p_i) & \text{if } e = s; t \\ upFinalNames(s, p_i) + upFinalNames(t, p_i) & \text{if } e = s + t \end{cases}$$

The functions $\overline{right}$ and $\overline{left}$ differ from $right$ and $left$, respectively, by using the functions $\overline{upFinalNames}$ and $\overline{upInitialNames}$ instead of the originals. These new functions, change the name of an old rule by a non-deterministic choice between the old and the new rule names. This modification enables also to keep the original rule (needed to start and finish the iteration cycle).

Definition 3.7 The functions $\overline{upInitialNames}, \overline{upFinalNames} : EXP \times P \to EXP$ are defined by:

$$\overline{upInitialNames}(e, p_i) = \begin{cases} (p + p_i) & \text{if } e = p \\ \overline{upInitialNames}(s, p_i); t & \text{if } e = s; t \\ \overline{upInitialNames}(s, p_i) + \overline{upInitialNames}(t, p_i) & \text{if } e = s + t \\ \overline{upInitialNames}(s, p_i)^* & \text{if } e = s^* \end{cases}$$

$$\overline{upFinalNames}(e, p_i) = \begin{cases} (p + p_i) & \text{if } e = p \\ s; \overline{upFinalNames}(t, p_i) & \text{if } e = s; t \\ \overline{upFinalNames}(s, p_i) + \overline{upFinalNames}(t, p_i) & \text{if } e = s + t \\ \overline{upFinalNames}(s, p_i)^* & \text{if } e = s^* \end{cases}$$

3.3 Adding conflicts between the initial rules

After the creation of dependencies between the rules, a new change is made to add a conflict among the initial rules of the RE. In this intermediary stage (2), the function *depState* is applied on the previous step result to add the conflicts. The addition of this conflict modifies: the set of rules, including the rules with added conflicts; the type graph, including new vertices used to create conflicts; the regular expression, replacing the old rule names with the new rule names; and the initial graph, adding the element to be consumed by the first rules defining thus the conflict between

them.

Definition 3.8 The function $depState : EXP \times P^2 \times (P \to Rule(T)) \times Graphs \times Graphs \to EXP \times P^2 \times (P \to Rule(T)) \times Graphs \times T\text{-}Graphs$ that adds conflicts between initial rules by is defined by:

$$
\begin{aligned}
&depState(e, P, \pi, T, I) = (e', P', \pi', T', I'), \text{ where:}\\
&P' = \emptyset; \quad \pi' = \emptyset; \quad e' = e;\\
&x \notin V_T;\ T' = (V_T \cup \{x\}, E_T, o^T, d^T);\ I' = addV(I, x);\\
&\forall p \in initial(e), \text{ with } i \in \mathbb{N} \wedge isnew(P \cup P', p_i)\\
&\qquad P' = P' \cup \{p_i\}\\
&\qquad L_{p_i} = addV(L_p, x);\ K_{p_i} = extType(K_p, x);\ R_{p_i} = extType(R_p, x)\\
&\qquad \pi' = \pi' \cup \{p_i \mapsto (L_{p_i} \leftarrow K_{p_i} \to R_{p_i})\}\\
&\qquad e' = upInitialNames(e', p_i)
\end{aligned}
$$

3.4 Choosing the used rules

In the last step (3), the function *rules* calculates the set of all rules of the new graph grammar GG, maintaining only instances of the rules that actually appear in RE e_2 (resulting from the second step). This step is necessary because the set of rules is always incremented with the new instances.

Definition 3.9 The function $rules : exp \mapsto \mathcal{P}(P)$ is defined by:

$$
rules(e) = \begin{cases} \{r\}, & \text{if } e = r \wedge r \in P \\ rules(t) \cup rules(s), & \text{if } e = t; s \vee e = t + s; \end{cases}
$$

The sequence of rule applications is provided now by the dependencies between the rules, where the application of one allows the application of the following one. Through these conditions, the conflict and the dependencies in the rules can represent the same behaviors of the REs.

Example 3.10 The Figures 4, 5 and 6, show the graph grammar GG resulting of the translation from the controlled graph grammar presented in Example 3.2.

Observe that new vertices are added in the type and initial graphs, as well as, in the left- and right-hand sides of the rules. In the type graph are added all new vertices used to create the dependencies and conflicts: ⌘, ❖, ☒, ☺, ✱ and ◉. In the initial graph only one vertex was added, since there is only the *Start* rule in the initial rules of the RE. This same vertex was also added in the left-hand side of *Start*.1 rule. Given that the new vertex is only on the left-hand side of *Start*.1 rule, this rule does not conflict with any other. Note that there are two versions of *Start* rule, this occur due to the closure operation. One version (*Start*.1) is applied

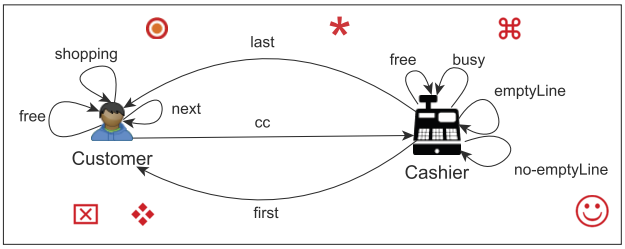


Fig. 4. Type graph of *GG* graph grammar.

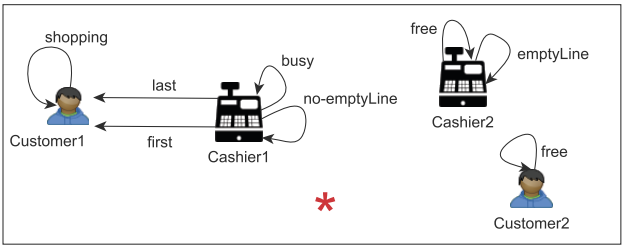


Fig. 5. Initial graph of *GG* graph grammar.

at the first time of the closure iteration, because there is still no vertex in the state graph that enables the other version (*Start*.2). The second version is enabled in the following iterations. Similarly, there are two versions of *Finish* rule, which are in conflict. In order to stop the interactions, the second version of *Finish* rule (*Finish*.1) must be applied. The other conflicts and dependencies were introduced by the inclusion of new vertices in the left- and right-hand sides of the intermediary rules. In this new grammar, *Enter*1.1, *Enter*1.2 and *Pay*1.1 are in conflict and all of them are dependent of the start rules. This means that only one of these rules can be applied, and only after one of the start rules have been applied. Observe that the rule names are all changed during the process of adding new vertices.

4 Conclusion

Graph grammar is a visual and intuitive language that allows specification and formal verification of systems with complex characteristics. This language was chosen because there are different techniques and tools that allow the use of model checking and theorem provers for the verification of systems properties that were described in that language. Controlled Graph Grammars permit to set an order on the rule applications that does not take into account the state but rather an auxiliary control structure, such as regular expressions. Nevertheless, there are no tools that allow formally verify properties for this kind of grammar.

In the present work, a translation of controlled graph grammar was defined, which results in an ordinary graph grammar, where the control is provided by the present data and the state. Therefore, the proposed translation enables the description of controlled graph grammars using Event-B language, allowing to use the theorem proving technique to verify CGG specifications. Thus the software

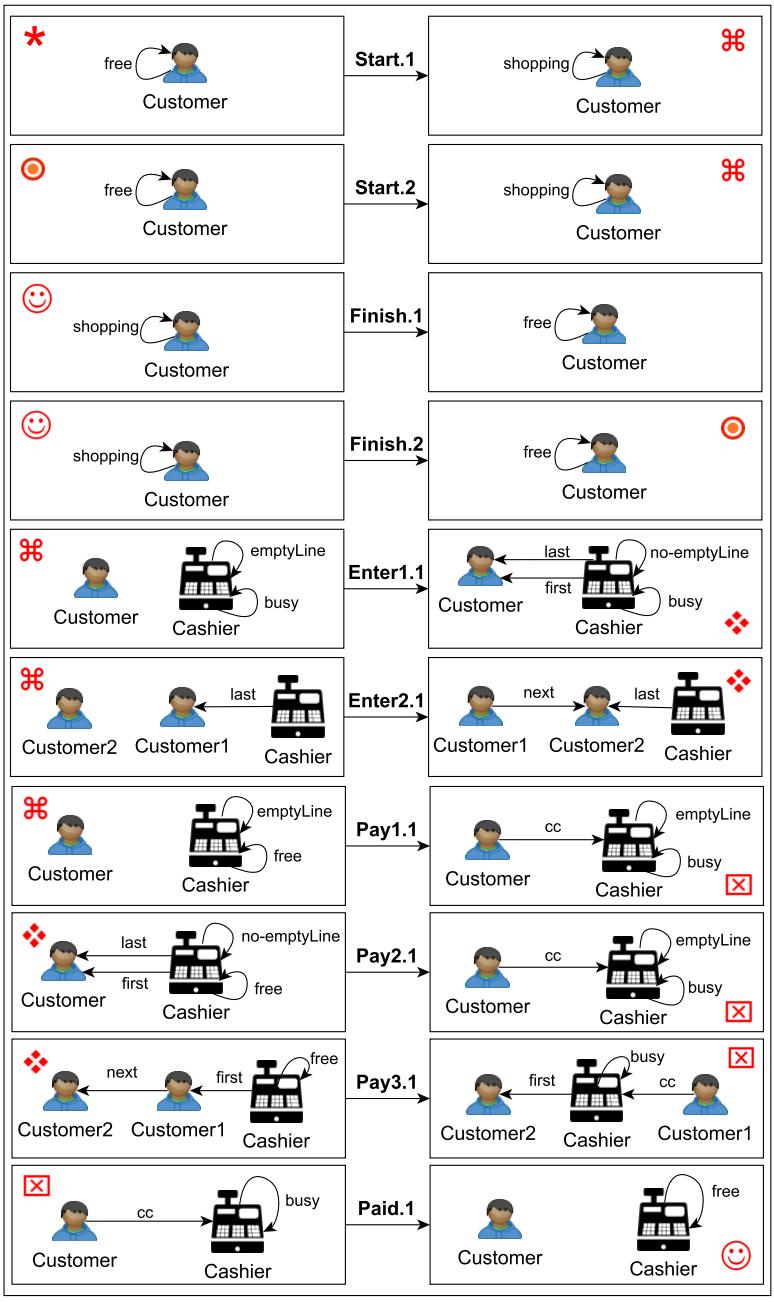


Fig. 6. Rules of *GG* graph grammar.

designers will have the opportunity to use a formal and intuitive language to specify systems and to carry out property verifications of these systems.

As a future work, we intend to investigate the use of variables to simplify the definition of translated graph grammar, which can contain a substantial number of rules, depending on the RE used to control the rule applications. In order to do this it is necessary to introduce CGGs with attributes.

References

[1] Dwyer, M. B., Hatcliff, J., Robby, R.,Puasuareanu, C. S., Visser, W.: Formal Software Analysis Emerging Trends in Software Model Checking. In: Future of Software Engineering. 120–136. Minneapolis (2007)

[2] Craigen, D., Gerhart, S., Ralston, T.: An International Survey of Industrial Applications of Formal Methods. Z User workshop, London (1993)

[3] Deharbe, D., Moreira, A. M., Ribeiro, L., Rodrigues, V. M.,: Introdução a Métodos Formais: Especificação, Semântica e Verificação de Sistemas Concorrentes. Revista de Informtica Terica e Aplicada. 7, 7-48 (2010)

[4] Behm, P., Benoit P.,Faivre, A., Meynadier, J.: METEOR : A successful application of B in a large project. World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, (1999)

[5] Abrial, J.R.: The B-book: Assigning Programs to Meanings . Cambridge University Press, New York, USA (1996)

[6] Ehring,H., Pfender, M., Schneider, H. J.: Graph-grammars: An Algebraic Approach In: annual symposium on switching and automata theory. pp.167-180. IEEE Computer Society. Washington (1973)

[7] Rensink, A.: Explicit State Model Checking for Graph Grammars. In Concurrency, Graphs and Models, Pierpaolo Degano, Rocco Nicola, and Jose; Meseguer (Eds.). Lecture Notes In Computer Science, Vol. 5065. Springer-Verlag, Berlin (2008)

[8] Ferreira, A. P. L., Foss, L., Ribeiro, L: Formal Verification of Object-Oriented Graph Grammars Specifications. Electronic Notes in Theoretical Computer Science. 175, 101-114 (2007)

[9] Baresi, L., Rafe, V., Rahmani, A. T., Spoletini, P.: An Efficient Solution for Model Checking Graph Transformation Systems. Electronic Notes in Theoretical Computer Science. 213, 3-21 (2008)

[10] 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)

[11] Cavalheiro, S. A. d. C.: Relational approach of graph grammars. 2010. phd thesis in Computer science - Federal University Rio Grande do Sul - UFRGS (2010)

[12] Ribeiro, L., Dotti, F. L., Cavalheiro, S. A. d. C., Cristine, F.: Towards Theorem Proving Graph Grammars using Event-B .ECEASST. 30. (2010)

[13] Cavalheiro, S. A. d. C., Ribeiro, L.: Verification of graph grammars using a logical approach. Science of Computer Programming. 77, 480-504 (2012)

[14] DEPLOY. Event-B and the Rodin Platform. http://www.event-b.org/ (last accessed july 2015). Rodin Development is supported by European Union ICT Projects DEPLOY (2008 to 2012) and RODIN (2004 to 2007)

[15] Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific Publishing, River Edge, USA (1997)