Complexity Analysis of Reactive Graph Grammars

Aline Brum Loreto *

Laira Vieira Toscani *

Leila Ribeiro *

Abstract

The aim of this paper is to present a way to calculate a complexity measurement of graph grammar specifications of reactive systems. The basic operation that describe the behavior of a graph grammar is a rule application. Therefore, this operation will be used to characterize the tasks to be performed within a system. The complexity measurement defined here will give us the minimum number of steps that must be present in a computation that performs a desired task.

Keywords: graph grammars, complexity.

*Instituto de Informática/CPGCC
Universidade Federal do Rio Grande do Sul
e-mail: {loreto,laira,leila}@inf.ufrgs.br
This work has been partially supported by projects QaP-For(Fapergs), Platus (CNPq) and Graphit (CNPq/DLR).

1. Introduction

Reactive systems [MP92] are a special kind of system in which everything that occurs is a reaction to some kind of stimulus. The idea is that the system is composed by a number of communicating entities that send stimulus to each other and cooperate in order to perform some application. Here we will call the entities by *objects* and the stimulus by *messages*. Reactive systems are usually concurrent because each of its components acts independently. Many of the most used concurrent applications of computers nowadays can be suitably modeled as reactive systems, for example, control systems and client/server applications. Due to the concurrency, distributed, non-deterministic and dynamic aspects of these applications, they are very difficult to analyze. This is specially true in case one wants to answer questions about the computational efforts involved in the execution of a reactive system.

To be able to investigate a system, we must first describe it using a formal description technique. Here we will use graph grammars [Ehr79, EHK+97] for this purpose. A graph grammar specification of a system consists of an initial graph and a set of rules. The initial graph represents the initial state of the system, that is, the objects and messages (triggers) that are present when the system is initialized. The rules describe the behavior of the system. Graph grammars rely on simple but powerful concepts: graphs represent in a natural way the distribution of the objects in a system; each rule describes a local change, and may be applied in parallel with others if they are not in conflict (do not try to delete the same items); in case there are conflicting rules enabled, the choice of the one to be applied is non-deterministic.

The complexity of a specification/program is always related to the computational work. Usually, this work is measured in terms of time ou memory needed to perform some task. But in special kinds of systems, other units are also interesting to consider. For example, in distributed applications, the number of messages exchanged to perform some task is of great interest because in such systems the time needed to execute is rather consumed by communication than by CPU [Lyn96]. In general many complexity measurements are interesting for a particular application domain, but many of them are very difficult (if not impossible) to be computed. The challenge is always to find out a useful measurement that is possible to be computed. The specification of a system is an abstract description of its intended implementation: it describes the properties the implementation has to satisfy. Complexity is one of these properties. If it is possible to state something about the complexity of the system already in the specification phase, the cost of the development will decrease and the generated system will be more efficient. As usual, once an implementation is proposed, we have to assure that this implementation satisfies all the necessary properties of the specification, including the complexity. For specification methods that follow the operational approach, like graph grammars, investigation about complexity measures can be very interesting for some application domains, like distributed and

concurrent systems.

The investigation of complexity of parallel systems in general is usually related to an execution on a concrete architecture [Jaj97, Akl89]. What is measured is the complexity of performing one or more tasks, typically the amount of time needed to complete the execution of all these tasks. In reactive systems like the ones studied in this paper, there is a great amount of parallelism involved: each entity may act in parallel with others, and the entity itself may perform many actions in parallel. Moreover, many of these systems are not meant to terminate, they rather receive messages, change their internal state, send some messages in reaction and remain ready to receive other messages (note that, maybe the task started by the original message did not end with this reaction, but will continue to be performed by the other entities that received the new messages). Thus, the notion of completion of a task is not straightforward. There are some questions, like *Will a reaction ever be triggered?*, that have no answer because we consider non-deterministic systems and do not assume fairness (and therefore in the general case a message that triggers some reaction may be indefinitely postponed). But there are some questions that are of interest in this kind of context and may be answered:*Is it possible that the system comes to a state in which a particular reaction is triggered?, How many steps must the system perform before such a state is reached?, Is it possible that this reaction is triggered a number of times?*. In this paper, we will provide a way to give answers to these questions. Such answers will be a great help for the construction of distributed systems, because many of them have a reactive nature and programmers can not rely only on tests (the same test executed twice may yield completely different results).

The paper is structured as follows: Sect. 2. we give an introduction to graph grammars and in Sect. 3. we show how they can be used to model reactive systems; in Sect. 4. we present a way to calculate a kind of complexity measurement for reactive systems based on the a graph grammar specification; in Sect. 5. we summarize our results and discuss the possible improvements to our approach.

2. Graph Grammars

We will follow the algebraic Single-Pushout Approach to graph grammars[Löw90, Löw93]. The technical definitions within this approach are described using category theory, and specially the approach we follow is called Single-Pushout because the application of a rule to a match is defined as a pushout in a category of graphs and partial graph morphisms.

Classically, a graph grammar consists of an initial graph, representing the *initial state* of a system, and a *set of rules* that can be used to transform the states of the described system. States are described by graphs. To allow more comprehensive representations of a state using a graph, typing mechanisms describing different kinds of vertices and edges may be used. There are many ways to define typing mechanisms

for graphs, here we will use the concept of a typed graph\cite{typed}. The idea of a typed graph is to use a graph, called *type graph*, to define the possible kinds of vertices and edges of a system, and an actual graph is then a graph consisting of instances of elements of the type graph. A typed graph can thus be described by a graph morphism relating each instance with its type. Figure 4 shows a (typed) graph grammar GG with type graph AG depicted in Figure 3, that is, and each of the other graphs in Figure 4 are graphs typed over AG (the mapping is implicitly defined by the same symbols of vertices and edges). The rules specify the behavior of the system in terms of local state changes. The left-hand side of the rule specifies a pattern that must be present in some state for the rule to be applied; the right-hand side shows the effect of the application of the rule; and the mapping from left- to right-hand side describes deletion (items that are not mapped), creation (items that are not in the range of the mapping) and preservation (items that are mapped). In the grammar GG, the mapping from left- to right-hand sides of the rules is indicated by using the same item on the both sides to specify that the item is preserved, different indices indicate that an item was deleted and another one of the same type was created.

Graph grammar: *is a tuple* $GG = (T, I^T, N)$ *where* T *is a graph, called the type graph,* I^T *is a graph typed over* T*, called the initial graph,* N *is a set of rules typed over* T*.*

3. Reactive Graph Grammars

A reactive system as a system consisting of autonomous entities that we will call *objects* that communicate and cooperate with each other through *messages*. Objects may have an internal state and relate to other objects within the system. The behavior of an object is described through its *reactions* to the receipt of messages (triggers). An object may perform many (re)actions in parallel.

Here we will describe a reactive system using a graph grammar. Therefore, we have to identify within a graph grammar what are the objects, messages and attributes, and then show how to specify reactions within this formalism. The structural part will be modeled by distinguishing different kinds of vertices and edges within the graphs that model states of the system (see Figure 1). Objects and messages will be modeled as vertices. A message must have as destiny an object and may have as arguments other objects and/or attributes of data types. An object may know other objects and may have attributes of data types modeling its internal state. This graph (Figure 1) can be considered as a type graph for a reactive system, and therefore we will call it *reactive model graph*. Note that a type graph models kinds of objects and links that may be present in an actual state of the system, but say nothing about the number of elements of each kind that must be present at a particular state. Although this is the desired (model) type-graph for reactive applications, here we will take into consideration only the items in boldface in this graph. The impacts of considering

attributes will be discussed in Sect. 5..

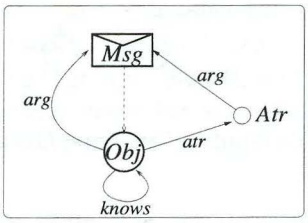


Figure 1: Reactive Model Graph RG

For each specific reactive system we may have various types of objects and messages that are relevant for that application. Thus, to build a specification for a reactive system using graph grammars one must first define what we call the *application type-graph*. This graph must be typed over the reactive model type-graph. The resulting structure of a reactive graph grammar is illustrated in Figure 2. Formally, this structure can be defined as a doubly-typed graph grammar (see [Rib96b], [DR00] for the formal definitions). One of the advantages of defining explicitly the model type-graph within the specification is to ease the comparison among specifications with respect to different model graphs (once we relate the model graphs, the relationships among the specifications can be obtained automatically).

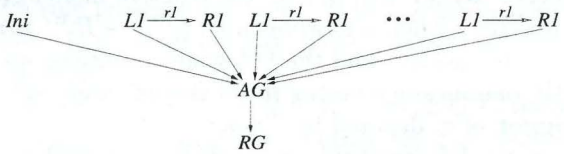


Figure 2: Structure of a Reactive Graph Grammar

Example 3.1

To model a producer/consumer application, we may define a type producer (P) and a type consumer (C). Producers may receive messages of type *produce* and *tmp*, and consumers may only receive messages of type *consume*. Furthermore, this graph specifies that producers may know consumers but not vice versa and that messages have no arguments. The behavior of producer/consumer application having these kinds of entities will be described using a graph grammar in Example 3.2.

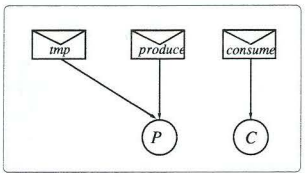


Figure 3: Application Type Graph AG

For a reactive graph grammar we will only allow rules that consume an element of type message, i.e., each rule represents a reaction to the kind of message that was consumed. Moreover, only one message may be consumed at a time be each rule. Note that the system may have many rules that specify reactions to the same kind of message (non-determinism), and that many rules may be applied in parallel if their triggers (messages) are present at an actual state (graph). Many messages may be generated in reaction to one message. Here we will restrict the number of generated messages of the same kind to one (to allow a simpler analysis of causality among rules). To make sure that a rule may be applied whenever its trigger is found in the actual state graph we will require that whenever a message appears in a graph, it has exactly all specified arguments and one destination.

The following definition is given in a semi-formal way because the corresponding formal definitions, although straightforward, require a number of concepts that are not needed elsewhere in this paper and were therefore not introduced.

Definition 1 Reactive Rule. Let RG be the reactive model graph and AG be a finite graph typed over RG. Then a morphism $r : L^{AG} \rightarrow R^{AG}$ is a ***(reactive) rule*** iff L and R are finite r is injective and the following conditions are satisfied:

i) There is exactly one message vertex m on the left-hand side of a rule. In this case, m is called ***trigger*** of r, denoted by $Trig(r)$.

ii) The message on the left-hand side of a rule is consumed by the application of the rule ($Trig(r) \notin dom(r)$).

iii) Messages have exactly one destination. Moreover all items in L and R must be connected. This latter condition is to avoid that a rule has non-local side effects.

iv) Objects may not be deleted.

v) A rule may not create two items (messages or objects) with the same type.

Now we can define a reactive graph grammar.

Definition 2 Reactive Graph Grammar. A ***reactive graph grammar*** is a tuple $GG = (AG, I, Rules)$ where AG, called the ***type*** of the grammar, is a finite graph typed over the reactive model graph RG, I is a finite graph typed over AG, called the ***initial graph*** of the grammar, and $Rules$ is a finite set of reactive rules typed over AG.

Example 3.2

Figure 4 shows a graph grammar specifying the behavior of a producer/consumer system. In the initial state, rules $p1$ and $p2$ are enabled (message *produce* triggers these rules). The indices 1 and 2 are used to distinguish items that are not the same, although having the same type. Everything that is on the left- and right-hand side of the rule will be preserved when the rule is applied. Items that are on the left- but not on the right-hand side are deleted (consumed), and items that are on the right- but not on the left-hand side are created.

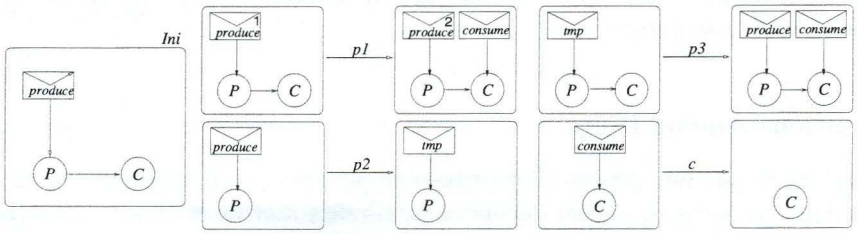


Figure 4: Graph Grammar GG

The behavior of a graph grammar is given by the applications of rules to graphs representing the actual states of the system, starting from the initial graph. The applications of rules may occur in parallel if the rules do not try to delete the same items. Note that, if a rule preserve an item that is deleted by another rule, these two may occur in parallel. This situation corresponds to one write and one read access occurring at the same time.

To be able to apply a rule $r : L^{AG} \rightarrow R^{AG}$ to a graph G^{AG} representing the actual state of a system one must first find out whether the rule can be applied. This is done by finding a *match* of the left-hand side of the rule in this actual graph. An application of a rule in a graph, called *derivation step*, deletes from the actual graph everything that is to be deleted by the rule and adds the items that shall be created by the rule. Formally, a match is described by a total (typed) graph morphism and a derivation step is a pushout in the category of (doubly-typed) graphs.

$$\begin{array}{ccc} L^{AG} & \xrightarrow{r} & R^{AG} \\ {\scriptstyle m}\downarrow & PO & \downarrow{\scriptstyle m^*} \\ G^{AG} & \xrightarrow[r^*]{} & H^{AG} \end{array}$$

A true concurrency semantics for graph grammars can be described by an unfolding construction that gives us a partial order of derivation steps [Rib96a]. The unfolding construction encompasses informations about all possible computations that

are described by the given graph grammar. Therefore, we could use it to answer questions about the minimum length of a computation to reach some state, or the minimum length of computations involving a number of applications of the same rule, etc. However, the unfolding is usually infinite because many of the applications of reactive systems are not meant to terminate. Moreover, the construction is quite complex. In this paper we will provide means to answer some questions about a grammar without having to build all possible computations. This will be done based on (potential) causality and conflict relationships among rules. These relationships have been defined in [Kor95, Rib96a, Rib98] to define axiomatizations of computations of a graph grammar. Here we will give a different interpretation to them to describe *possible computations.*

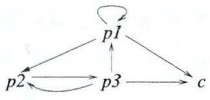
3.1 Relations Among Rules

To characterize the class of all possible computations of a graph grammar three relationships are needed: causal dependency, conflict and weak conflict (see [Kor95, Rib96a, Rib98]). The third relationship (weak conflict) is needed due to the possibility of preservation of items when a rule is applied (this relationship expresses conflicts between preservation and deletion of items). However, the restrictions made on graph grammars to define reactive graph grammars ruled out the occurrence of weak conflicts, and therefore they will not be introduced here. But, as soon as we considered attributes of objects, this kind of conflict will naturally arise (see discussion in the conclusion). Now we will (informally) introduce the causal and conflict relationships.

(Potential Causal) Dependency Relation ($\prec$): *The intuitive idea of this relation is that a rule a is a (direct) potential cause of a rule b if a creates some item of a type that is needed (preserved/deleted) by b.* Note that this does not imply that b can only be applied after a has been applied. This is because the trigger of b may be present already on the initial state, or the trigger of b may be generated by another rule c. Therefore, we can say that if a is a potential cause of b then there *may* be a sequence of rule applications in which a creates the trigger of b. The transitive closure of $\prec$ will be denoted by $\prec^+$.

Example 3.3

In the grammar of Figure 4, we have the following direct relationships: $p1 \prec p1, p1 \prec p2, p1 \prec c, p2 \prec p3, p3 \prec p1, p3 \prec p2, p3 \prec c$. They can be better visualized by the graph below, where $p1 \rightarrow p2$ indicates that $p1 \prec p2$. If there is a way from p_i to p_{ii} then $p_i \prec^+ p_{ii}$.



(Potential) Conflict Relation: *The potential conflict relationship relates two rules* $r1$ *and* $r2$ *that need the same type of trigger.* This means that in any computation in which one of these two rules appears, there must have been a choice between them. But not every application of $r1$ is in conflict with an application of $r2$ because they may try to use different copies of the trigger existing in an actual graph.

Example 3.4

In the grammar of Figure 4 we have the potential conflict relation: $p1\#p2$ because both $p1$ and $p2$ have the same trigger (*produce*).

3.2 Semantics of Graph Grammars

The semantics of a graph grammar can be defined as the class of all computations that can be performed using the rules of the grammar starting with the initial state. These computations may be sequential or concurrent, giving raise to sequential and concurrent semantic models. Figure 5(a) Illustrates a *sequential derivation* for a grammar with starting graph I. In this derivation we have a total order ($<$) of derivation steps ($s1 < s2 < s3$) that denotes the sequence in which they have occurred in this computation. If we make a suitable gluing[†] of all intermediate graphs of this derivation, we obtain a structure called *concurrent derivation* (Figure 5(b)). Now, the total order that existed in the sequential derivation is lost, but we may define a partial order ($\prec$) between the steps that describes the causality relation: if $s1 \prec s2$ then $s1$ must occur to allow the occurrence of $s2$. A concurrent derivation can be seen as an equivalence class of sequential derivations (all possible sequential derivations corresponding to the totalizations of $\prec$ are in this class). Note that a concurrent derivation seems very much like a graph grammar if we consider the graph C, called *core graph*, as being the type of the grammar. This means that we can describe all computations of a graph grammar using graph grammars. Of course, the graph grammar used to describe the semantics are a special kind of grammars (for example, the causality relationship must be a partial order – see [Kor95, Rib98] for the axioms defining this class of grammars), in which each rule corresponds to a derivation step of the original grammar.

[†]This gluing is actually a colimit of a diagram in the category of doubly-typed graphs [Kor96, Rib96a].

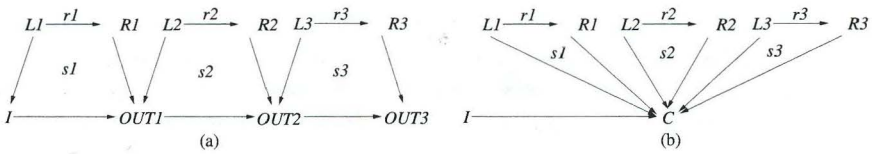


Figure 5: (a) Sequential derivation (b) Concurrent Derivation

By a ***computation*** we mean a concurrent derivation. The ***length*** of a computation κ, denoted by $|\kappa|$, is the number of steps involved in this computation. As a concurrent derivation is a graph grammar, the causality relationship within a computation have already been defined. Note that the interpretation of the causality relationship in a computation is different than in a graph grammar: in a graph grammar $r1 \prec r2$ means that there *may* be a computation in which rule $r2$ depends on elements delivered by the application of rule $r1$, while in a computation $s1 \prec s2$ means that the derivation step $s2$ created the necessary items for the occurrence of derivation step $s2$. Given a computation κ, κ_s is a ***subcomputation*** of κ if all steps in κ_s are also in κ and the causality relationship is preserved, i.e., , $s1 \prec_{\kappa_s} s2 \Rightarrow s1 \prec_{\kappa} s2$[‡].

4. Complexity of Reactive Graph Grammars

When considering the parallel execution of tasks, we may have two different measurements: one of them is the time cost (taking into account that some tasks may run in parallel), and the other is the computational effort needed to complete the task (that is independent of the fact that some operations may execute in parallel).

To execute a graph grammars means to apply rules. Therefore it is reasonable to say that the complexity can be measured in terms of the number of rule applications needed to perform some task, that is, a rule application is considered as a fundamental operation. We assume that all rule applications need similar computational efforts to be performed. An implementation that preserves rule applications will preserve the complexity of the system, that is, a corresponding notion of satisfaction of a specification must take rule applications into acount. This is the approach followed here. Here we have restricted the kind of graph grammar by using the notion of messages and objects: in all graphs involved in a grammar each time a message appears, all (and exactly) its arguments must also be present, and each rule deletes exactly one message. Thus, to find out is a rule can be applied we must just find the trigger message of this rule in the set of vertices of type message in the state graph. For practical applications, these restrictions are not a problem (see, for example, the

[‡]This subcomputation relation corresponds to a concurrent prefix relationship between computations. To model sequential prefixes we would have to require that the causality relationship is also reflected (see [Kor96]).

specification of a telephone system presented in [Rib96b, KR97]). Note that, in our approach, each application of a rule correponds exactly to a message exchange in the system. Therefore, the complexity measure that we give is given in terms of number of exchanged messages to perform some task, that is a reasonable unit of measure for distributed/concurrent systems.

As a graph grammar is inherently non-deterministic and we do not assume fairness, questions like *Will a rule r ever be applied?, What is the length of the biggest computation in which a rule r appears?* make no sense. What we could ask is *Is there a computation in which the rule r is applied?, What is the length of the smallest computation in which r appears?.* Moreover, as the systems we are considering are reactive systems, questions like *How many steps are necessary for the system to complete?* have usually no answer because these systems typically are not meant to terminate. Therefore, we may associate completion of some task with the application of some specific rule (in the producer/consumer example, we may say that each time rule c is applied one cycle produce/consume is finished). In this context, it is rather reasonable to ask questions like *How many steps are necessary for a rule to execute a number n of times?.* The necessary number of steps characterizes the minimum length of computation that performs the required number of rule applications. A way to calculate this will be provided in this section.

4.1 Definitions

Definition 3 *Let* $GG = (I, T, Rules)$ *be a graph grammar and* $r, r_i \in Rules$, $i \in N$.
(Direct) Causes: $Cause_r = \{r_i | r_i \prec r\}$
Cause independent rules:$r_1 \prec r$ *and* $r_2 \prec r$ *are cause-independent rules if* $(\nexists r_0.$ $r_0 \prec^+ r_1$ *and* $r_0 \prec^+ r_2)$.
Path:$p = \langle r_1, r_2, \ldots, r\rangle$ *is a path to* r *if* $r_1 \prec r_2 \prec \cdots \prec r$
Rule in path:r *is in* $p = \langle r_1, \cdots, r_n\rangle$ *if* $\exists i \in [1..n].r_i = r$
Length:$|p| = n$, *if* $p = \langle r_1, \cdots, r_n\rangle$
Paths:$paths(r) = \{p | p$ *is a path to* $r\}$
Cycles:$cycles(r) = \{p = \langle r, \cdots, r_n\rangle \in paths(r) | r_n = r$ *and* $\forall i \in [2, n-1].r_i \neq r\}$
Feasible path:$feasiblePath(r) = \{p = \langle r_1, \cdots, r\rangle \in paths(r) | \forall c \in cycles(r).|p| < |c|\}$
Feasible rule:$feasibleRule(r) = \{r1 | r1$ *is in* p *and* $p \in feasiblePath(r)\}$

Let $c(n, r)$ denote the cost of n occurrences of rule r. In the following we will build stepwise an expression that gives a minimum cost for such a computation. Consider $Cause_r$ the set of the direct causes of r (i.e., $r_i \prec r$). Each occurrence of r must depend either on one of these causes or on messages that trigger r that are present already on the initial state of the graph grammar (I). We are interested in defining the smallest computation that contains n occurrences of rule r. If the trigger for this rule is already on the initial graph, to apply this rule one time, there can be no smallest path than using this trigger. Therefore, we will always use first all triggers

that are present in the initial graph, and then check in which other ways we may get the necessary trigger for r. Let n_i ($n_i > 0$) denote the number of occurrences of r that are dependent on a rule $i \in Cause_r$, and $trig(r)$ be the number of messages that trigger r present on the initial graph.Thus, to be able to apply r n times we need to have n_i occurrences of r_i satisfying:

$$n - trig(r) = \sum_{i \in Cause_r} n_{r_i},\ n_{r_i} > 0 \tag{4.1}$$

Example 4.1

Consider the graph grammar in Figure 4. Suppose we want to calculate the minimum number of rule applications necessary to apply rule c twice, that is, we want to calculate $c(2, c)$. If there would be enough occurrences of the trigger of c (message $consume$) in the initial graph (Ini), then the minimum number of steps would be exactly 2, corresponding to the two applications of rule c. But in this example we have no occurrences of message $consume$ in the initial graph, that is, $trig(c) = 0$. Therefore, the only way to apply rule c is to apply first a rule that generate a message $consume$. The rules that do this are in $Cause_c = \{p1, p3\}$. Thus, to generate the two necessary triggers for c we have the following possibilities: apply $p1$ twice ($n_{p1} = 2, n_{p3} = 0$), apply $p3$ twice ($n_{p1} = 0, n_{p3} = 2$), or apply $p1$ and $p3$ once ($n_{p1} = 1, n_{p3} = 1$). Note that, according to equation 4.1, in each case the sum of n_{p1} and n_{p3}must be 2 (because $n = 2$ and $trig(c) = 0$).

In equation 4.1 we have defined the restrictions for n_i. In the simpler case, when we have all necessary triggers in the initial graph, we have:

$$n \leq trig(r)\ and\ c(n, r) = n \tag{4.2}$$

But, if there are not enough triggers for r in the initial state ($n < trig(r)$), considering the cost of n_{r_i} occurrences of r_i (to generate the missing triggers), plus the n occurrences of r (the cost of applying this rule the required number of times), we arrive at the following expression to describe the cost of applying n times rule r:

$$n + \sum_{r_i \in Cause_r} c(n_{r_i}, r_i) \tag{4.3}$$

If the rule r is not a direct or indirect cause of any of the r_is ($r \not\prec^+ r_i$), expression 4.3 gives us already the cost of this combination of n_{r_i} occurrences of r_i.

Now suppose that $r \prec^+ r_i$, that is, there is a chain of causally related rules from r to r_i. If we look at expression 4.3, we may notice that the occurrences of r that preceded r_i were counted twice: one in the term n and one in $c(n_{r_i}, r_i)$. Therefore we

must diminish these occurrences from the total amount of necessary rule applications. Actually, we need to subtract at most one occurrence of r prior to r_i because if there are more than one, the others must have already been subtracted when the corresponding $c(n_{r_i}, r_i)$ was calculated.

We need to determine, for each i, if r_i have occurred using triggers that have been created by some occurrence of r or not. In the first case, we must subtract the occurrence of r needed to apply r_i, and in the latter case not. This will be done via the function $rep(r, n_{r_i}, r_i)$. For each i, n_{r_i} minus the number of executions of r_i that may occur not dependent on r gives us the number of times we shall consider.

As we are only interested in the smallest path, we will only consider paths that lead to occurrences of r_i that are smaller than the smallest cycle containing r. These paths are exactly the ones belonging to $feasiblePath(r_i)$. The definition $feasibleRule$ will be used when we have a situation in which $r_i \prec r \prec^+ r_i$. In this case there is a cycle to r_i (if we require that all causes of each rule are cause-independent, then this cycle must be unique). Moreover, any path to r_i that contains r_i must be longer than the smallest cycle (or equal, if the path is the smallest cycle itself). The fact that $r_i \in feasibleRule(r)$ means that there is a path from r to r_i that is shorter than the smallest cycle on r_i (and therefore can not contain r_i). Furthermore, we need to find out whether such a path could ever be initiated in the considered grammar. This is described by the existence of the necessary triggers in the initial graph:

$$n_{r_i} - \sum_{fr \in feasibleRule(r_i)-\{r\}} trig(fr)$$

However, we should not subtract a negative number of times. In the case we have $n_{r_i} < \sum_{fr \in feasibleRule(r_i)-\{r\}} trig(fr)$ then the smallest way to reach all n_{r_i} occurrences of r_i does not include occurrences of , and therefore nothing should be subtracted from the result of 4.3. Thus, we obtain

$$rep(r, n_{r_i}, r_i) = \begin{cases} 0, & \text{iif } r \not\prec^+ r_i \\ max\left\{0, n_{r_i} - \sum_{fr \in feasibleRule(r_i)-\{r\}} trig(fr)\right\}, & \text{if } r \prec^+ r_i \end{cases} \tag{4.4}$$

Remark 1 *If $r \not\prec^+ r_i$, all n_{r_i} occurrences of r_i did not include r, that is, the number of occurrences (between the n_i's) of r_i including r is zero. Conversely, if $r \prec^+ r_i$ for each i, the number of occurrences of r_i (between the n_{r_i}'s) that include r is n_{r_i} minus the ones that came from paths that are smaller than any containing r. Thus, $rep(r, n_{r_i}, r_i)$ represents the number of occurrences of r_i dependent on r.*

Example 4.2

In our example, to calculate $c(2,c)$ we found out that we need to use triggers generated by $p1$ and/or $p3$ (see example 4.1). Thus we will have to find out whether during the generation of these triggers the rule c was applied. This can be done using equation 4.4: $rep(c,n,p1) = 0$ and $rep(c,n,p3) = 0$, for any number n (because $c \not\prec^+ p1$ and $c \not\prec^+ p3$). To calculate the cost of applying 2 times rule $p3$, we would have to calculate $rep(p3,2,p2)$ because $p2 \prec p3$ and $trig(p3) = 0$. In this case, we have $p3 \prec^+ p2$ and therefore we have to calculate the feasible rules of $p2$ (see figure of Example 3.3): $cycles(p2) = \{\langle p2,p3,p2\rangle, \langle p2,p3,p1,p2\rangle, \langle p2,p3,p1,p1,p2\rangle, \ldots\}$, $feasiblePath(p2) = \{\langle p2\rangle, \langle p3,p2\rangle, \langle p1,p2\rangle\}$ (only paths shorter than $|\langle p2,p3,p2\rangle| = 3$), $feasibleRules(p2) = \{p2,p3,p1\}$. Therefore, $rep(p3,2,p2) = max\{0, 2-(trig(p1)+trig(p3))\} = max\{0, 2-(1+0)\} = 1$. This means that, if we apply rule $p3$ two times depending on rule $p2$, one of the occurrences of rule $p2$ must have been dependent on an application of $p3$.

Then, expression 4.3 modifies to:

$$\sum_{r_i \in Cause_r} c(n_{r_i}, r_i) + \left(n - \sum_{r_i \in Cause_r} rep(r, n_{r_i}, r_i) \right) \tag{4.5}$$

If, for some i, $c(n_{r_i}, r_i) = \infty$, then this combination of n_i's and r_i's can never occur, what implies that, if r can occur n times, it is not using this combination of causes. In this case, expression 4.5 assumes value ∞. It is also possible that we have, for all i, $c(n_{r_i}, r_i) \neq \infty$, meaning that, isolated, we may have n_{r_i} occurrences of r_i,but we may have the case that it is impossible to perform them together (because, for example, they use the same trigger). This situation will not be considered now and will be discussed later.

In case we have a cycle involving r and r_i, if r_i occurs at least once, r will be enabled any number of times. The next function is used to find out whether it is possible that r_i occur at least once.

$$f(r_i) = \begin{cases} \infty, & if \sum_{cr \prec^+ r_i} trig(cr) = 0 \\ 0, & otherwise \end{cases}$$

Thus, we can replace expression 4.5 by

$$\begin{cases} \infty & , \text{if } \sum_{r_i \in Cause_r} f(r_i) = \infty \\ \sum_{r_i \in Cause_r} c(n_{r_i}, r_i) + \left(n - \sum_{r_i \in Cause_r} rep(r, n_{r_i}, r_i)\right) & , \text{otherwise} \end{cases} \tag{4.6}$$

This way we have characterized each combination of n_{r_i} occurrence of rule r_i. As we want to calculate the minimum cost for this computation, we now have to minimize expression 4.6, subject to the restrictions defined in 4.1. Then we finally reach the expression defining the minimum number of rule applications that are necessary in a computation to have n occurrences of a rule r:

$$c(n.r) = \begin{cases} n & , \text{if } trig(r) \geq n \\ min_{Restr} \left\{ \begin{cases} \infty & , \text{if } \sum_{r_i \in Cause_r} f(r_i) = \infty \\ CostComp & , \text{otherwise} \end{cases} \right\} & , \text{if } trig(r) < n \end{cases} \tag{4.7}$$

$$\text{with } Restr = \sum_{r_i \in Cause_r} n_{r_i} = n - trig(n); n_{r_i} \geq 0$$

$$\text{and } CostComp = \sum_{r_i \in Cause_r} c(n_{r_i}, r_i) + \left(n - \sum_{r_i \in Cause_r} rep(r, n_{r_i}, r_i)\right)$$

Example 4.3

Now we can calculate the minimum cost of applying rule c two times. According to the discussion in Example 4.1, we have 3 cases to calculate the costs cA, cB and cC. The minimum of these is the lenght of the smallest computation we are looking for:

$c(2,c) = min\{cA, cB, cC\}$ where

$cA = c(2,p1) + c(0,p3) + (2-0) = c(2,p1) + 2$

$cB = c(0,p1) + c(2,p3) + (2-0) = c(2,p3) + 2$

$cC = c(1,p1) + c(1,p3) + (2-0) = c(1,p1) + c(1,p3) + 2$

Now we have to calculate the component expressions:

$c(2,p1) = min\{c(1,p1) + (2 - rep(p1,1,p1)), c(1,p3) + (2 - rep(p1,1,p3))\}$
$= min\{1 + (2-1), 2 + (2-0)\}$
$rep(p1,1,p1) = max\{0, 1-0\} = 1,$
$feasibleRule(p1) - \{p1\} = \emptyset$
$rep(p1,1,p3) = max\{0, 1 - (trig(p2) + trig(p3))\} = max\{0, 1-(1+0)\} = 0$
$feasibleRule(p3) - \{p1\} = \{p2, p3\}$

$c(2,p3) = min\{c(2,p2) + (2 - rep(p3,p2,p2))\} = min\{3 + (2-1)\} = 4$

$c(1,p1) = c(1,p2) = 1$ because $trig(p1) = 1 = trig(p2)$

$c(1,p3) = min\{c(1,p2) + (1 - rep(p3,1,p2))\} = min\{1 + (1-0)\} = 2$
$rep(p3,1,p2) = max\{0, 1 - trig(p2)\} = max\{0,0\} = 0,$
$feasibleRule(p2) - \{p3\} = \{p2\}$

Thus, we have that $c(2,c) = min\{2+2, 4+2, 1+2+2\} = 4$

Example 4.4

Now consider the same rules, but with an empty initial graph. Then we have:

$c(1,p1) = min\{\infty, \infty\} = \infty$ because we have now $trig(r_i) = 0$ for all rules $r_i \in \{p1, p3, p2\}$ and therefore $f(p1) = f(p3) = \infty$.

In the cost expression given above, we have not considered the fact that the same two rules may need the same trigger (and are, therefore, not only in potential but in real in conflict). As the computation of the cost of each cause of a rule is done separately, we may reach a conclusion that the minimum number of steps needed is actually smaller that the real one because of the sharing of triggers (in some cases, there may be no computation performing the required number of steps – that is, the cost in infinite – and our result could be a natural number). However, there may be cases in which there are no potential conflicts and this situation occurs. An example would be a rule that generates the triggers for two others that both generate the trigger for another rule. This situation is called cause-dependency (two causes of the same rule depend on a common rule). The cases for which the expression given here gives an exact result (that is, for which there is actually a computation of that minimum length) are as follows: there are no potential conflicts between the rules of the grammar and all causes of each rule are cause-independent.

The following theorem states that if the cost expression we have defined gives a value x for applying n times a rule r then there can be no computation of the corresponding grammar that has n applications of rule r that has length smaller than x. This theorem proves that, if we consider a graph grammar with cause-independent rules, we have really defined a lower bound for the computation length necessary to have the required number of rule applications. Note that the theorem is valid also for grammar that have conflicts, just that in this case it can be that there is no computation having n occurrences of r that has exactly the length computed by $c(n,r)$.

Theorem 4..1 *Let $GG = (I, T, Rules)$ be a graph grammar in which all causes of each rule are cause-independent and κ be a computation of GG with $|\kappa| = m, m \geq 1$. If κ contains n occurrences of derivation steps using rule r, then, for all subcomputations κ' of κ that also has n occurrences of r, we have $c(n,r) \leq |\kappa'|$.*

Proof. In case $n = 0$, then for any $r \in Rules$, $n \leq trig(r)$ and therefore $c(n,r) = 0$, what implies that $c(n,r) \leq |\kappa'|$, for all subcomputation κ of κ'. The proof for $n \geq 1$ will be by induction on $m = |\kappa|$. Let $Cause_r = \{cr | cr \prec r\}$.

IB: Let κ be a computation of GG with $|\kappa| = 1$. If κ contains n occurrences of derivation steps using rule r, then $n \leq 1$ because $|\kappa| = 1$. In this case $n = 1$ because by assumption $n \geq 1$, and therefore the computation κ must consist of one derivation step using rule r because it has only one derivation step. Moreover, if this derivation step was possible, then $trig(r) \geq 1$. Then we have $n \leq trig(r)$, what implies that $c(n,r) = n = 1$ and therefore $c(n,r) = |\kappa|$.

IH: For any computation κ of GG with $|\kappa| \leq m$. If κ contains n occurrences of derivation steps using rule r, then, for all subcomputations κ' of κ that also have n occurrences of r, we have $c(n,r) \leq |\kappa'|$.

IS: Let κ be a computation with $|\kappa| = m + 1$. Let κ' be a subcomputation of κ that has n occurrences of rule r.

Case 1 *Suppose* $trig(r) \geq n$. *In this case* $c(n, r) = n$. *Any computation* κ' *that has* n *occurrences of rule* r *must consist of at least* n *derivation steps, that is* $|\kappa'| \geq n = c(n, r)$.

Case 2 *Suppose* $trig(r) > n$ *and* $r \not\prec^+ r$. *Let* $r_1, r_2, \ldots r_j \in Cause_r$. *The computation* κ *contains* n *occurrences of rule* r, *each one preceded by* r_i, $i = 1..j$. *Let* n_{r_i} $(n_{r_i} > 0)$ *be the number of times that* r *occurs in* κ *depending on a rule* r_i. *Then* $\sum_{r_i \in Cause_r} n_{r_i} + trig(r) = n$. *As* $trig(r) < n$ *and* $n > 0$, κ *has at least one occurrence of* r *whose trigger was not in the initial graph. Thus, there must be an* $r \not\prec^+ r$ *such that* $trig(r) \neq 0$, *and we must have* $\sum_{r_i \in Cause_r} f(r_i) \neq \infty$. *Besides, as* $r \not\prec^+ r$ *by assumption,* $c(n, r) = min\left\{\left(n - \sum_{r_i \in Cause_r} rep(r, n_{r_i}, r_i)\right)\right\}$, *subject to* $n_{r_i} > 0$ *and* $\sum_{r_i \in Cause_r} n_{r_i} = n - trig(r)$. *For each* r_i, *with* $n_{r_i} \neq 0$, *consider the smaller subcomputation* κ_i *of* κ *that contains the* n_{r_i} *occurrences of* r_i. *Using the induction hypothesis for* $\kappa_i (|\kappa_i| < |\kappa'| \leq |\kappa|$ *because* κ_i *does not contains the occurrences of rule* r*), we obtain* $c(n_{r_i}, r_i) \leq |\kappa_i|, i = 1..j$. *Therefore we can conclude that* $\sum_{r_i \in Cause_r} c(n_{r_i}, r_i) \leq \sum_{r_i \in Cause_r} |\kappa_i|$. *As, by assumption, the causes of each rule of the grammar are cause-independent, all subcomputations* κ_i *do not share common resources, and thus* $\sum_{r_i \in Cause_r} |\kappa_i| \leq |\kappa'| - n$, *and this implies that* $n + \sum_{r_i \in Cause_r} c(n_{r_i}, r_i) \leq |\kappa'|$. *As we are considering the minimum of all combinations in the definition of* $c(n, r)$, *we must have that* $c(n, r) \leq |\kappa'|$.

Case 3 *Suppose* $trig(r) > n$ *and* $r \not\prec^+ r$. *Following the same reasoning as the latter case, we may identify the* r_i *and* n_{r_i} *of a computation* κ *and conclude that* $\sum_{r_i \in Cause_r} f(r_i) \neq \infty$. *In this case, this means that* $c(n, r) = min_{Restr}\left\{\sum_{r_i \in Cause_r} c(n_{r_i}, r_i) + \left(n - \sum_{r_i \in Cause_r} rep(r, n_{r_i}, r_i)\right)\right\}$, *where* $Restr = \sum_{r_i \in Cause_r} n_{r_i} = n - trig(n); n_{r_i} \geq 0$. *The induction hypothesis gives us* $c(n_{r_i}, r_i) \leq |\kappa_i|, i = 1, ..j$ *and thus* $\sum_{r_i \in Cause_r} c(n_{r_i}, r_i) \leq \sum_{r_i \in Cause_r} |\kappa_i|$. *The subcomputations* κ_i *are independent by assumption, but there may be occurrences of* r *in one of these subcomputations (in at most one because if more than one have occurrences of* r *than the grammar is not cause-independent). These occurrences have already been considered in the corresponding* $c(n_{r_i}, r_i)$ *and must therefore be subtracted. By remark 1,* $rep(r, n_{r_i}, r_i)$ *represents the number of occurrences (between the* n_{r_i}*'s) of* r_i *dependent on* r. *So we get* $\sum_{r_i \in Cause_r} c(n_{r_i}, r_i) \leq \sum_{r_i \in Cause_r} |\kappa_i| \leq |\kappa'| - \left(n - \sum_{r_i \in Cause_r} rep(r, n_{r_i}, r_i)\right)$. *This implies* $\sum_{r_i \in Cause_r} c(n_{r_i}, r_i) + \left(n - \sum_{r_i \in Cause_r} rep(r, n_{r_i}, r_i)\right) \leq |\kappa'|$. *As we are considering the minimum of all combinations in the definition of* $c(n, r)$, *we must have that* $c(n, r) \leq |\kappa'|$.

□

5. Conclusion

In this paper we have defined a special kind of graph grammar that is suitable for the specification of reactive systems and have presented a way to calculate the complexity of these grammars. This complexity is measured in terms of number of rule applications.

Graph grammars have been used to specify a variety of systems [EHK+97], but nevertheless almost no research have been made to investigate the complexity of the specified systems (usually complexity issues were only regarded to build tools for graph grammars that had to cope with the graph isomorphism problem to find matches for rules). Although we have considered only a restricted class of graph grammars, this work can be a starting point in a theory of complexity of graph grammar specifications. Graph grammars can be considered as a generalization of Petri nets [KR96]. Therefore, it would be interesting to investigate to what extent the theory of Petri nets concerning complexity and related issues could be compared to our approach, and also which results for Petri nets (as a special case of graph grammars) can be achieved.

A relevant improvement of our approach (important for practical applications) would be to consider graph grammars with attributes (data types) and also read only/write access to attributes as a condition for the application of rules. This would have the impact that the causality and conflict relationships are no longer enough to characterize the computations of a graph grammar, we need a further relationship called weak conflict (see [Kor96, Rib98, KR98]). Moreover, in this case, a rule may depend not only on its trigger but also on a particular value of an attribute, thus, it is potentially dependent on all other rules that create that trigger and on all that may change the value of that attribute. The reasoning about the minimum length of possible computations in this case becomes much more involved.

Another issue for further research is to investigate the minimum time cost in such systems. One can use the causality and conflict (and possibly also weak conflict) relationships to reason about which rules may be applied in parallel and thus get a minimum time necessary to perform a rule a number of times.

The work developed here is to be implemented within a tool called PLATUS, that is an environment for specification and simulation of reactive systems based on graph grammars currently under development [CR98, CR99, CMR00].

References

[Akl89] S. Akl, *The design and analysis of parallel algorithms*, Prentice Hall, 1989.

[CMR96] A. Corradini, U. Montanari, and F. Rossi, *Graph processes*, Fundamenta Informaticae, vol. 26, no. 3/4, 1996, 241 – 265.

[CMR00] B. Copstein, M. Móra and L. Ribeiro, *An environment for formal modeling and simulation for graph grammars*, 33rd Annual Simulation Symposium, 2000, pp. 74–82.

[CR98] B. Copstein and L. Ribeiro, *Specifying simulation models using graph grammars*, 10th ESS European Simulation Symposium And Exhibition, 1998, pp. 60–64.

[CR99] L. Ribeiro and B. Copstein, *Compositional Construction of Simulation Models using Graph Grammars*, International Workshop and Symposium AGTIVE - Applications of Graph Transform ation with Industrial Relevance, Lecture Notes in Computer Science, vol. 1779, 2000, pp. 87–94.

[DR00] F. Dotti and L. Ribeiro, *Specification of Mobile Code Systems using Graph Grammars*, Formal Methods for Open Object-based Systems IV (S. Smith and C. Talcott, eds.), Kluwer Academic, 2000, pp. 45–64.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini, *Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach*, The Handbook of Graph Grammars, Volume 1: Foundations, World Scientific, 1997, pp. 247–312.

[Ehr79] H. Ehrig, *Introduction to the algebraic theory of graph grammars*, 1st Graph Grammar Workshop, Lecture Notes in Computer Science, vol. 73 (V. Claus, H. Ehrig, and G. Rozenberg, eds.), 1979, pp. 1–69.

[Jaj97] J. Jaja, *An introduction to parallel algorithms*, Addison-Wesley, 1997.

[KR96] M. Korff and L. Ribeiro, *Formal relationships between gra ph grammars and Petri nets*, Lecture Notes in Computer Science, vol. 1073, Springer, 1996, pp. 288–303.

[KR97] M. Korff and L. Ribeiro, *Graph grammars for the specification of concurrent systems*, IX SBES Brazilian Symposium on Software Engineering, 1997, pp. 199–214.

[KR98] M. Korff and L. Ribeiro-Korff, *True concurrency=interleaving + weak conflict*, Eletronic Notes in Theoretical Computer Science, vol. 14, 1998.

[Kor93] M. Korff, *Single pushout transformations of generalized graph structures*, Tech. Report RP 220, Federal University of Rio Grande do Sul, Porto Alegre, Brazil, 1993.

[Kor95] M. Korff, *True concurrency semantics for single pushout graph transformations with applications to actor systems*, Information Systems - Correctness and Reusability (R. J. Wieringa and R. B. Feenstra, eds.), World Scientific, 1995, pp. 33–50.

[Kor96] M. Korff, *Generalized graph structure grammars with applications to concurrent object-oriented systems*, Ph.D. thesis, Technical University of Berlin, 1996.

[Löw90] M. Löwe, *Extended algebraic graph transformations*, Ph.D. thesis, Technical University of Berlin, 1990.

[Löw93] M. Löwe, *Algebraic approach to single-pushout graph transformation*, Theoretical Computer Science, vol. 109, 1993, 181–224.

[Lyn96] N. Lynch, *Distributed algorithms*, Morgan Kaufmann, 1996.

[MP92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems, Specification*, Springer, 1992.

[Rib96a] L. Ribeiro, *Parallel composition and unfolding semantics of graph grammars*, Ph.D. thesis, Technical University of Berlin, 1996.

[Rib96b] L. Ribeiro, *A telephone system's specification using graph grammars*, Tech. Report 96-23, Technical University of Berlin, 1996.

[Rib98] L. Ribeiro-Korff, *Occurrence graph grammars*, 5th WoLLIC International Workshop on Logic, Language, Information and Computation, 1998, pp. 92–100.