Graph Grammars for the Specification of Concurrent Systems[1]

Leila Ribeiro†, Martin Korff‡
Universidade Federal do Rio Grande do Sul†
Instituto de Informática
email: leila@inf.ufrgs.br

Nutec Informática‡
Departamento de Desenvolvimento
email: mkorff@nutec.com.br

Resumo

Gramáticas de grafos tem sua origem em gramáticas de Chosmky, onde os strings são substituídos por grafos. Essas gramáticas possuem algumas vantagens: a representação gráfica ajuda a entender a especificação; elas têm fundamentos teóricos sólidos; e elas são um meio independente de implementação para descrever e analisar sistemas computacionais. Neste artigo é apresentado o resumo de um estudo de caso sobre a especificação de um sistema telefônico usando gramáticas de grafos, bem como são descritos os aspectos semânticos e analíticos resultantes.

Palavras-chave: gramáticas de grafos, especificação de sistema, concurrencia

Abstract

Graph grammars have origined from generalizing Chomsky grammars from Strings to Graphs. They visually support intuition, have a solid theoretical foundation, and provide a formal, implementation independent means for the description of discretely evolving computations and their formal and tractable analysis. In this paper we present the outline of a case study of specifying a telephone system and report on the resulting semantical and analytical issues.

Keywords: graph grammars, system specification, concurrency

1 Introduction

Software engineering techniques must assure that a piece of software is indeed a solution of the original problem. This involves formalizing the problem initially given by some informal ideas and requirements as well as turning this formalization into executable code. Formalizing demands some natural and intuitive means of description. Formal proofs require that the semantics of a specification or program is determined by mathematical models rather than by existing compilers. The progressing inclusion of distribution and communication as well as user interfaces becoming more and more sophisticated must especially be considered. Being formal and suggestive at the same time, while being

[1]**This work is partly supported by the project QaP-For (FAPERGS), by ProTEM, and by CNPq.**

especially suited to treat concurrency aspects, makes graph grammars appear promising in developing reliable software.

Graphs are a very natural means to explain complex situations on an intuitive level. Graph rules may complementary be used to capture the dynamical aspects of systems. The resulting notion of graph grammars generalizes Chomsky grammars from strings to graphs. Although the research area of graph grammars and graph transformations is relatively young — its roots date back to the early seventies — methods, techniques, and results in this area have already been studied and applied in a variety of fields in computer science such as formal language theory, pattern recognition and generations, compiler construction, software engineering, concurrent and distributed system modelling, database design and theory, etc. Here we will stick to the algebraic approach to graph grammars [Ehr79, Löw93]. This has been well-investigated especially in the area of concurrency. Graph grammars can be considered as a syntactical and semantical generalization of Petri-Nets [Cor95, KR96]. A Petri-Net is limited since there is neither a natural way to dynamically change its structure nor to include references into its tokens. Overcoming these limits is especially useful in modelling growing and shrinking communities of objects refering to and communicating with others.

The Private Branching Exchange (PBX) System which has originally been implemented at Nutec nicely fits into this category of systems. It is characterized by a high communication traffic and a high degree of desirable parallelism. The specification outlined below is based on a corresponding case study specification (about 50 pages) developed in one of the student's project at the TU Berlin and the successor version presented in [Rib96b]. The specifications has been accompanied by a number of interesting questions about properties of the system, e.g., concurrent activities or deadlocks. They could often be answered by applying some of the various theoretical results, which are by no means trivial especially when the extended framework of typed and attributed graphs is considered.

The following section 2 will informally introduce the kind of a graph grammar we are using. Section 3 contains a brief summary of the the telephone system and a subsequent modeling using graph grammars. In section 4 we finally sketch the main theoretical techniques which can be applied to the previous specification thus rewarding the current approach. Finally we will conclude our results.

2 Graph Grammars

The following description of a graph grammar tries to be as comprehensive as possible. For corresponding formal definitions of algebraic graph grammars see [Ehr79, Löw93] or [EHK+96, Kor96, Rib96a].

Graph grammars generalize Chomsky grammars from strings to graphs. Unlike Chomsky rules, a graph *rule* $r : L \to R$ does not only consist of graphs L (left hand side) and R (right hand side), but has also an additional part: a partial graph (homo)morphism r mapping edges and vertices in L to edges and vertices in R respectively in a compatible way. Compatibility here means that whenever an edge e_L is mapped to an edge e_R then the source (target) vertex of e_L must be mapped to the source (target) vertex of e_R.

Graph grammars specify a system in terms of *states—modelled by graphs*[2]—and *state changes—modelled by derivations*. The following operational interpretation of a rule $r : L \to R$ provides the basis for this specification approach:

- Items in L which do not have an image in R are *deleted*.
- Items in L which are mapped to R are *preserved*.
- Items in R which do not have a pre-image in L are *created*.

Rather than using plain graphs merely consisting of vertices and edges we actually use typed and attributed graphs making specifications more natural, compact, and easy to survey.

Attributes: Attributes are (algebraically specified) algebras (carrier sets plus operations) that may be used to assign values to the vertices (and edges) of a graph. They are used to integrate some basic data types like natural numbers or strings which shall not be represented graphically. Using term algebras particularly offers to use variables in the specification.

Example. [Attributes] Table 1 shows a list of attributes. As examples consider the attribute(ion set)s *Bool* and *Nat*. *Bool* denotes the algebra of boolean values which may be denoted by On and Off or T and F respectively (plus operations). *Nat* denotes the algebra of natural numbers $0, 1, \ldots$ including standard operations like $+, -, \ldots$. ☺

Attribute	*Value*
Bool	$On(T)\vert Off(F)$
Status	$Mute\vert Free\vert Busy\vert Call\vert Carrier\vert Ring\vert Wrong\vert Speak$
Digit	0|1|2|3|4|5|6|7|8|9
Nat	natural numbers
List	lists of natural numbers
UserId	$\{user1, user2, \ldots, usern, adm\}$

Table 1: Attributes of the PBX System

Type graph: A type graph is a graph in which each vertex and each edge represent some distinct type of vertex/edge in some specification. Each actual graph of the system must then have an interpretation in terms of this type graph. The concept of typing imposes structural restrictions on the graphs that represent states of the system.

Example. [Type Graph] The type graph in Figure 1 uses attributes the from Table 1. It demands that the following types of vertices are distinguished: PHONE, CENTRAL, and ENVIRONMENT(or user), *P:Digit*, *P:Sign*, *C:Digit*, and *E:Act*. Analogously this holds for edges. For example, if there is no edge connecting an `E-Act` message to the central in the type graph *Type* of an specification, there can be no state of this system in which such a message is sent to the central because this state would not be a graph having *Type* as a type graph. ☺

[2] or graph-like structures

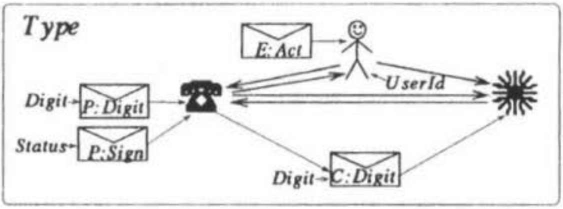


Figure 1: Type Graph

A *graph grammar* consists thus of the following components: *attributes*, *type graph*, *initial graph* and *rules*.

Example. [Grammar] Figure 2 shows a graph grammar comprising the attributes in Table 1, the type graph in Figure 1, the initial state (graph) Ini and 3 rules ($r1$, $r2$ and $r3$).

The graph Ini shows two phones connected to a central, where each phone have some user and the central has an administrator. The users and the administrator are able to act (modeled by E-Act messages connected to them. Rule $r1$ models the dialing of a digit Pd on a phone by a user. The user and the phone are preserved, the E-Act message of the user is deleted, another E-Act message is sent to the user (created) and a message containing the dialed digit is sent to the phone (created). Rule $r2$ models the forwarding of the digit Pd to the central, and rule $r3$ models the working out of this message by the central: it sends a message to the phone in order to stop the phone's carrier tone.

As an example of using variables as attributes consider the rule $r3 : L3 \rightarrow R3$. $R3$ contains a vertex attributed with the value $Mute$, that is one of the signals that a telephone may receive. In $L3$ we see a variable Pd of sort $Digit$. We could have specified the same situation without using the variable Pd, but then we would have needed 10 rules: one for each possible value that Pd may assume. ☺

The operational behaviour of a system described by a graph grammar is described by applying the rules of the grammar to actual graphs. The application of a rule to an actual graph, called *derivation step*, is possible is there is an occurrence of the left-hand side of this rule into the actual graph. This occurrence, called *match*, is a total graph morphism because one intuitively expects that all elements of the left-hand side must be present at the actual graph to apply the rule. The result of the application of a rule $r : L \rightarrow T$ to a graph IN is obtained by the following steps:

1. Add to IN everything that is created by the rule (items that are in the right-hand side R of the rule but not in the left-hand side L).
2. Delete from the result of 1 everything that shall be deleted by the rule (items that are in the left-hand side L of the rule but not in the right-one R).
3. Delete dangling edges. This step is necessary because it can be that some vertices deleted in step 2 had incoming and/or outcoming edges, and these must be deleted such that the result becomes a graph. This implicit deletion of edges is sometimes a feature and sometimes a problem. Intuitively, one may compare this phenomena of disallocating a variable to which there are pointers to.

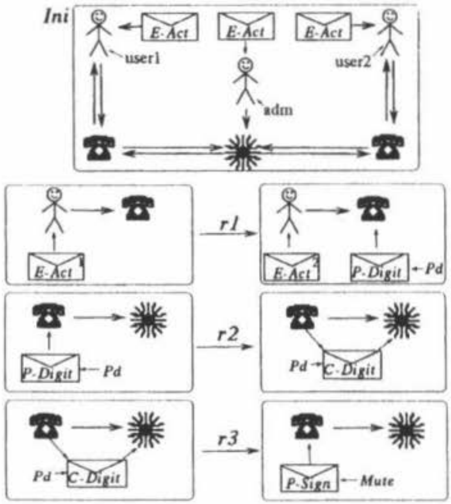


Figure 2: Graph Grammar GG

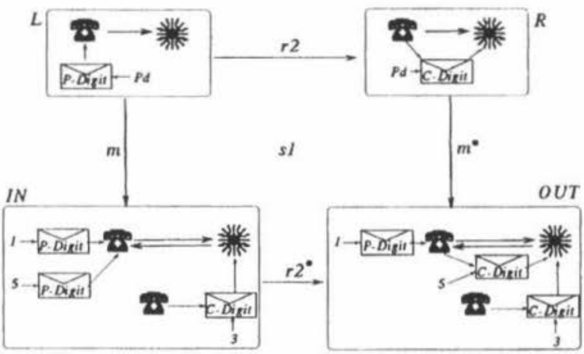


Figure 3: Derivation Step

Example. In Figure 3 IN is transformed into OUT using rule r2 applied at match m. Fixing $m(Pd) = 5$ (i.e., the match maps Pd onto the number 5) uniquely distinguishes m which is required to be a graph morphism. ☺

The sequential semantics of a graph grammar GG is given by all sequences of derivation steps using the rules of GG, starting with the initial graph of GG, and in which the output graph of one step is the input graph of the following one (see Figure 4).

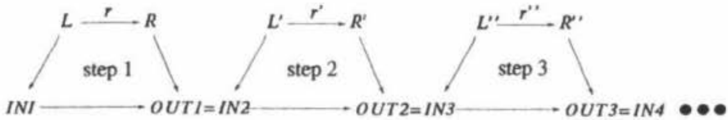


Figure 4: Sequential Derivation

3 Example: Specification of a Telephone System

3.1 Description of the PBX System

A PBX provides an intelligent connection between a (small) telephone pool – as it can typically be found in companies – and several external lines giving access to an already existing (public) telephone net. The heart of such a system is a piece of hardware—often called a CENTRAL. The CENTRAL controls the (internal) communications between the PHONEs and manages the connection of PHONEs inside the system with PHONEs outside belonging to a second (external) telephone CENTRAL. For simplicity, we ignore any additional features as e.g., programmable keys, last number redialing, etc., and restrict to the internal side of such a telephone system, i.e., one CENTRAL connected to several standard PHONEs. Therefore, the main aim of a PBX system presented here is to control the calls between the telephones that are connected to it. The messages it receives from its phones are usually informations about the state of the hook of the phones and the digits dialed by the users of the phones. The reaction to these messages is to send appropriate tone/ring signals to the phones and establish a connection between phones. It should have become clear that a telephone system is characterized by a high communication traffic and that, in particular when more than one telephone is involved, there is a high degree of desirable parallelism.

3.2 Specification of the PBX System

To build an specification for the PBX system using graph grammars we followed the following steps:

1. Define the active objects involved in the system, their attributes (internal structure) and interconnections with other objects.
2. Develop the interface of services offered by each object.
3. Specify the services.

As the first two steps refer to stactical aspects of a system, they will be specified by the type graph and attributes of a graph grammar. The specification of the services will be done via rules and the initial graph of a grammar.

3.2.1 Step 1: Define Objects and Interconnections

The attributes of elements of the PBX system are summarized in Table 1. We will have as types: PHONE, CENTRAL, ENVIRONMENT and messages (that is, these elements will be modelled graphically). The choice of which items shall be represented graphically or textually is left to the specifier. Usually, basic data types like natural numbers, booleans and lists are better represented as attributes (a graphical representation of them is possible, but not so understandable as the textual one). Figure 5 shows the type graph of the PBX graph grammar.

The vertices drawn as ☎, ✹ and 🯅 correspond to PHONES, CENTRALS and ENVIRONMENTS respectively, and the arrows between them correspond to "knows" relationships.The result of this development step is actually this graph without the message vertices and corresponding edges.

The internal structure of a phone modelled by attributed vertices (not drawn) carrying the phone's internal variables. These dots are connected to the phone via edges. The internal state of the phone consists of the following attributes: *P.st* (telephone status), *P.h* (status of the hook), *P.ph* (pending hook message) and *P.pd* (pending digit message).

The attributes of the CENTRAL include a TAB component that carries informations about each PHONE that is connected to the CENTRAL. These informations are modelled by *C.nr* (number of the PHONE), *C.st* (status of the PHONE), *C.dn* (dialed number), *C.pd* (pending dialed digit) and an established connection between two PHONES (indicated by the c-edge). Moreover, the CENTRAL has another attribute, namely a list of still free phone numbers (that is used to check whether a dialed number is valid and for the insertion of new phones in the net).

3.2.2 Step 2: Define Interface

Message	*Target*	*Parameters*	*Description*
`E-Act`	ENV	—	Indicates that the user is ready to act.
`E-Sign`	ENV	$s : Status$	Indicates that the user had become the signal s
`P-Hook`	PHONE	$h : Bool$	Indicates that the hook of the phone was lifted (Off) or put down (On).
`P-Digit`	PHONE	$d : Digit$	Indicates that the digit d has been dialed on this telephone.
`P-Sign`	PHONE	$s : Status$	Indicates that the telephone shall send the audio signal s to its environment.
`C-Hook`	CENTRAL	p : PHONE, h : *Bool*	Indicates that the hook of the phone p was lifted (Off) or put down (On).
`C-Digit`	CENTRAL	p : PHONE, d : *Digit*	Indicates that the digit d has been dialed on phone p.
`C-NewPh`	CENTRAL	p : PHONE	Indicates that the telephone p shall be connected to the net.

Table 2: Messages of the PBX System

The services that are offered by each of the object components can be ordered via messages. The kinds of messages needed in the PBX system are listed in Table 2. A service offer is modeled by graph grammars by connecting in the type graph the message corresponding to this service (together with its parameter types) to its target object. This will

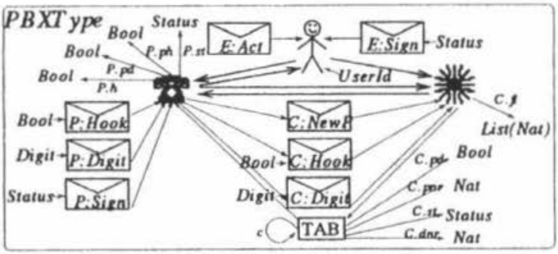


Figure 5: Type graph of PBX

have the effect that, during the execution of this grammar, a message can only be sent to the object that offered the kind of service ordered by this message. By including all the messages listed in Table 2 into the graph resulting of step 1, we get the type graph $PBXType$ of the graph grammar PBX.

3.2.3 Step 3: Specify services

For each message that may be sent (whose interface was specified in step 2), there shall be at least one rule to work it out. All rules express the reaction of the system to some message. If there are more than one rule with the same left-hand side, this indicates non-deterministic choice. The rules of the telephone system may be grouped into 3 kinds according to the target of the messages that are worked out by each rule. For space reasons, not all rules will be shown here. They can be found in [Rib96b]

- Environment rules: As they do not belong directly to the specification of the PBX system, the environment rules will not be drawn here. However, it should be said that the specification of such rules turned out to be a very good basis for the elaboration of a user's manual for the system.

  $r1$: The user takes the hook off.
  $r2$: The user puts the hook on.
  $r3$: The user dials the digit Pd.
  $r4$: The administrator creates a message to include a new telephone to the net.

- Phone rules (Figure 6):

  $r5$: Forwarding of a hook off message from the telephone to its central.
  $r6$: Forwarding of a hook on message from the telephone to its central.
  $r7$: Forwarding of the first dialed digit from the telephone to its central (in this case, the phone has a carrier tone). Notice that the phone is only allowed to send a message `C-Digit(Pd)` to the central if there are no pending digit messages from this phone on this central. This way of modelling guarantees that the digit messages are received by the central in the same order they were sent by the corresponding phones. This can also be modelled without synchronization, but then a complex queue structure is needed.
  $r8$: Forwarding of the second dialed digit from the telephone to its central (in this case, the phone has no tone).

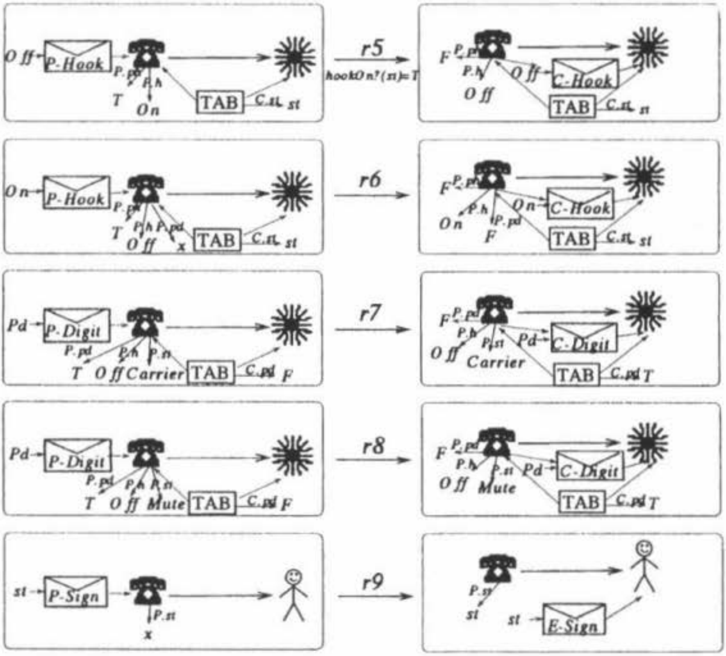


Figure 6: PBX: PHONE Rules

$r9$: Forwarding of an audio signal from the telephone to its user.

- Central rules (Figure 7):

  $r10$: Starting of a telephone call: The central notices that the hook of a phone is off and send a carrier tone to this phone.
  $r11$: Establishment of a connection between two phones.
  $r12$: A phone gives up a call (put the hook on without having a connection).
  $r13$: A phone interrupts a connection.
  $r14$: The central stores the first number dialed by a phone.
  $r15$: The phone called by another one is already busy.
  $r16$: The phone corresponding to a dialed number is called (rings).
  $r17$: There is no phone in the net that corresponds to the dialed number.
  $r18$: A new phone is connected to the net.
  $r19$: A new phone is not connected to the net because there is already another telephone with the corresponding number.

To complete the specification we still need to specify the initial state of the system. We could have specified here a state consisting only of the CENTRAL and its administrator and the first steps of the execution of this grammar would build a telephone net. Here we will rather start with a concrete net consisting of two PHONES. The initial graph of this system is the graph $PBXIni$ shown in Figure 8.

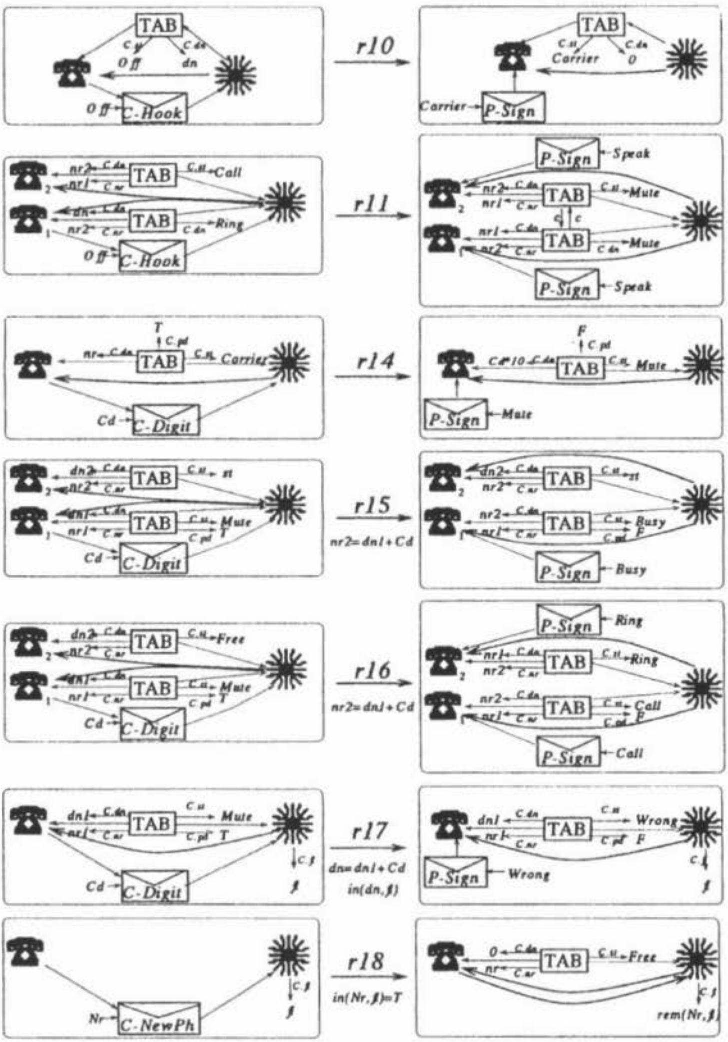


Figure 7: *PBX*: CENTRAL Rules

4 Concurrency Aspects of Graph Grammars

Depending on the aspects of a system we are interested in, one semantical model may be more appropriate than others. For the telephone system, the main aspect we are interested in is concurrency. Therefore semantical models that describe concurrency seem to be more adequate in this case. These models do not only focus on the reachable states but they

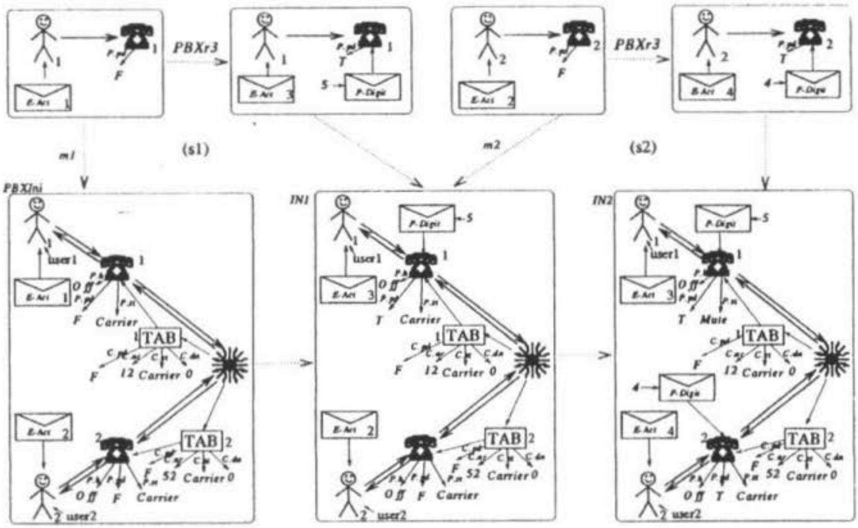


Figure 8: Derivation Sequence $\sigma 4$

rather emphasize on the *way* these states are reached. A suitable semantics for concurrent systems provides means for reasoning about computations: which actions may happen in parallel, what are the relationships between different computations and between actions of the same computation, etc. To understand which kinds of relationships may occur between different actions of a system, we will give a small example. These relationships are described in different ways by different semantical models.

Example. The following actions are possible in the PBX system:

1. PHONE 12 gets a `Digit(5)` message.
2. PHONE 52 gets a `Digit(4)` message.
3. PHONE 12 gets a `Digit(3)` message.
4. PHONE 12 forwards the `Digit(3)` message (received in action 3.) to its central.

Obviously, actions 1 and 2 may occur in parallel because they involve different telephones. Actions 1 and 3 are in conflict because only one digit may be dialed at each time (phone numbers are *sequences* of digits). Action 4 depends on action 3 (PHONE 12 can only send a digit that was dialed to the central). One derivation sequence of the PBX system, namely derivation $\sigma 4$, is shown in Figure 8. The matches used for the applications of the rules are indicated by corresponding indices. In this derivation sequence the user of PHONE 12 generates a `P-Digit(5)` message (step $s1$) and then the user of PHONE 52 generates a `P-Digit(4)` message (step $s2$). Let derivations $\sigma 5$ and $\sigma 6$ be defined as follows: In derivation $\sigma 5$ these two messages are sent in the inverse order; and in derivation $\sigma 6$ the first step ($s5$) represents the generation of a `P-Digit(3)` message on PHONE 12 and the second step ($s6$) represents the forwarding of this digit to the CENTRAL. By considering the sequential sematics of graph grammars described in Sect. 2, we would have the following set of derivation sequences (";" denotes sequential composition): $\sigma 1 = (s1), \sigma 2 = (s3), \sigma 3 = (s5), \sigma 4 = (s1; s2), \sigma 5 = (s3; s4), \sigma 6 = (s5; s6)$ ☺

As the telephone system is highly parallel, many derivation steps may occur concurrently. In the next Sections 4.1, 4.2 and 4.3 we will discuss the modeling of concurrency features via graph grammars. In Sect. 4.4 we will shortly describe a way of building the specification of a concurrent system as the parallel composition of the components using graph grammars.

4.1 Parallelism

Graphs describe explicitely the topological distribution of a state. Therefore it is natural that many actions acting on this state may occur in parallel. These actions are modeled by rule applications. To know whether two rules $r1$ and $r2$ may be applied in parallel on some graph G we have to consider also concrete matches $m1$ and $m2$ (because a rule may have many different matches on a graph and each of them may lead to a different situation). Then we may have 4 cases:

a) Matches do not overlap in G: In this case $r1$ and $r2$ can be obviously applied in parallel as they act disjointly.
b) Matches overlap on preserved items: Here the rules can also be applied in parallel because the shared items are 'read-only' (preserved) by both.
c) Matches overlap on items that are preserved by one rule and deleted by the other: Here we may or not allow the parallel application of these rules. Allowing it means that we allow one 'read-only' and one 'writing' rule to act together on shared items.
d) Matches overlap on items that are deleted by both rules: Here we may also allow or forbid the parallel application of these rules. Allowing means that two 'writing' rules may act together on shared items.

Graph grammars give us the possibility to choose which kinds of parallelism shall be possible in our system. Each choice will probably lead to a different kind of concurrent semantics for this system. Usually cases **a)** and **b)** are allowed. These cases represent what we call *strong parallelism* because it is a symetric kind of parallelism: if two derivation steps $s1$ and $s2$ may occur in parallel the may also occur sequentially in any order and vice versa. Case **c)** is called *weak parallelism* and it represents asymetric parallelism: if two derivation steps $s1$ and $s2$ may occur in parallel they may also occur at least in one order. Case **d)** is usually forbidden, although under some restrictions it may have applications. In this case, there may be parallel situation for which there is no corresponding sequential derivation.

Example. For the PBX system we consider strong and weak parallelism. If we consider the situation described in the example above we will have a situation in which only actions 1 and 2 or actions 2 and 3 may be occur in parallel (case a). Now consider a situation in which a phone, say 52, is trying to call phone 31 (that is not connected to the net). This would lead to the application $a16$ of rule $r16$ (that checks in the internal list of the central whether a phone is or not connected to the net). This action may occur in parallel with an action $a18$ that puts a new phone on the net (rule $r18$) only if we allow weak parallelism: the list of free phone numbers is updated by $a18$ while its read by $a16$. ☺

4.2 Concurrent Derivations

The semantics of graph grammars is usually given by sets of sequential derivations. In Sect. 4.1 we saw that informations about which actions may occur in parallel can be obtained from the sequential derivation. However, this analysis may become quite hard to do if the actions we want to compare are not subsequent in the sequential derivation. A way to have parallel actions represented explicitely is to construct a *concurrent derivation*. It is constructed by gluing all intermediate steps of a sequential derivation into a *core graph*. In this way, the sequence of (sequential) rule applications turns into a partially ordered set of concurrent rule applications (or actions). The partial order is induced by the overlappings of the different actions in the core graph. It gives us a dependency relation among actions. Two actions are concurrent, i.e., they may occur in parallel, if and only if they are mutually independent.

Example. For example, the sequential derivation $\sigma 4$ gives raise to the concurrent derivation $\kappa 4$, written $\sigma 4 \rightsquigarrow \kappa 4$. In this concurrent derivation $\kappa 4$, we can not say which of the actions $a1$ or $a2$ shall occur "first" (in a corresponding sequential derivation). This is because the pre- and post-conditions of these actions do not overlap in the core graph, i.e., the images of the pre- and post-conditions of these rules are disjoint. Moreover, $\kappa 4$ is also the concurrent derivation of the sequential derivation $\sigma 5$, i.e., $\sigma 5 \rightsquigarrow \kappa 4$. This stresses the fact that $\sigma 4$ and $\sigma 5$ represent in fact the same computation if we abstract from the sequential order. Let $\kappa 6$ be the concurrent derivation generated from $\sigma 6$ ($\sigma 6 \rightsquigarrow \kappa 6$). In the concurrent derivation $\kappa 6$, the pre-condition of action $a6$ overlaps with the post-condition of action $a5$ on the item `C-Digit(3)` of the core graph, and this item was created by the action $a5$. This implies that action $a6$ is causally dependent of action $a5$, written $a5 \leq a6$, and thus there is only one possible sequential order in which these action can be observed: $a5; a6$. Thus, the following concurrent derivations are included in $ConcSem(CGV)$ ("," denotes that two actions are causally unrelated and $\leq$ denotes causal dependency): $\kappa 1 = (a1), \kappa 2 = (a2), \kappa 3 = (a5), \kappa 4 = (a1, a2), \kappa 5 = (a5 \leq a6)$ ☺

4.3 Unfolding Semantics

As concurrent derivations are obtained by gluing the intermediate graphs from sequential derivations, they can not describe non-deterministic (conflict) situations explicitly, such situations are described by the non-existence of a derivation including the two "conflicting ones". The interplay between non-determinism and concurrency gives a very rich description of the behaviour of a system. A well-accepted way to describe this interplay is by modeling a system using a causal and a conflict relationships (as it is done in event structures [Win87]). The *unfolding semantics* of a graph grammar [Rib96a] is able to express these relationships in a natural way. Moreover, these relationships are defined not only between actions but also between items from the state graphs (this gives us a good basis for analysis of a grammar).

The unfolding is constructed inductively starting with the initial graph of the grammar and in each step all possible applicable rules are applied to the results of the last step. As each item of the core graph can be created by at most rule, the unfolding is an acyclic grammar (each rule of the unfolding – that represents an application of a rule of the original grammar – can be applied at most once). In fact, the unfolding constructed inductively is actually the union of all concurrent derivations of a grammar.

Example. The part of the unfolding of the graph grammar of the PBX system corresponding to the actions described in the example consists of 4 actions (that are exactly the actions involved in the concurrent derivations). From the unfolding we can derive (among others) the following relationships between actions:

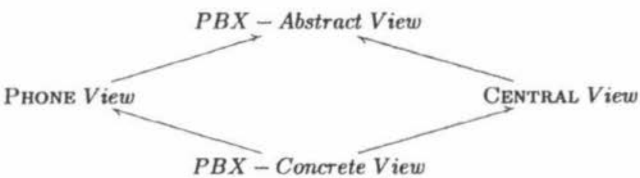
- **Causal Dependency**: $a5 \leq a6$ ($a5$ creates an item that is needed by $a6$.)
- **Conflict:** $a1 \overset{\#}{\Longleftrightarrow} a5, a1 \overset{\#}{\Longleftrightarrow} a6$ ($a1$ and $a5$ delete the same item, as $a1$ is in conflict with $a5$ and $a6$ depends on $a5$, $a1$ in also in conflict with $a6$.)

☺

4.4 Parallel Composition

The *parallel composition* of graph grammars introduced in [Rib96a] is based on a top-down development of the system: first an abstract description of the components and their interconnections is fixed, then each component is specialized separately and at the end they are put together. The parallel composition requires that both specializations done in the components do not change the behaviour of the abstract view; then the composed system is obtained by gluing the specializations. The main result of the parallel composition is that the behaviour of the composed system is completely defined by the behaviours of the components (there is no unexpected behaviour in the composed system). This concept appears quite useful for the development of concurrent/reactive systems. It also serves as a basis for the communication between members of different teams. Moreover, it makes explicit which changes may affect other components.

Example. For the PBX system, we can recognize 3 components: CENTRAL, PHONE and ENVIRONMENT. The third component corresponds to the users. Assuming that it is enough to give only an abstract description for the ENVIRONMENT component, we will have only two local components for the PBX system: CENTRAL and PHONE. This idea is summarized in the picture below. The arrows between the components mean *specialization* (or refinement) relationships. Both the PHONE and the CENTRAL views are specializations of the abstract view, and the concrete view can be seen as the smallest specialization of the abstract view including the specializations described by the PHONE and the CENTRAL components. A specification of the PBX system based on these components can be found in [Rib96b].



☺

5 Conclusion

We have presented (an outline of) a case study in which graph grammars have been used in order to specify a PBX system. Computations are massively based on communication. Their effect is very local. There is a high degree of desirable parallelism. We observed that this kind of system can naturally and advantageously be modelled using graph grammars.

States are modelled by graph(like structure)s — state changes are modelled by graph rule applications. It appears that object Identities and references may naturally captured by nodes and edges; basic data types may be integrated via attribution algebras. Similar to an entity relationship diagram in the area of databases a fixed (type) graph may be defined to jointly represent all objects and their static relations (structure). Services are represented by rules. The left hand side (precondition) specifies an operations' head including attribute and parameter values in which it applies. The right hand side (post-condition) specifies the resulting effects, like sending new messages and changing attribute values.

Using graph grammars for corresponding system specications is not only rewarded by their simple and suggestive appearance, but also supported by a considerable number of further arguments: their formal basis provides

- an implementation independent definition of system behaviour,
- correctness proofs of critical sections.
- tools investigating basic system properties (liveness, invariants).

On the other hand it should be noted that a (rule-based) graph grammar specification does not directly support an implementation using common imperative or procedural languages like C, Java, or Perl. Besides, there is still a great need for application-oriented specification methodologies and drawing conventions. Thus the advantages of using graph grammars may fully be exploited only when environments and tools have sufficiently been developed possibly offering all or some of the following features:

- filterable display of statical and behavioural aspects of system.
- semi-automatic analysis and graphical visualization of conflicts and independencies;
- high-level behavioural (simulating) interface for users and program designers.
- implementation independent, formal definition of system behaviour,

Apart from this there should be a deeper investigations of how graph grammars may be integrated into the object-oriented analysis and design process. This becomes even more challenging when theoretical compositionality results are considered which allow a feasible specifications on different levels of abstraction.

References

[Cor95] A. Corradini, *Concurrent computing: from Petri nets to graph grammars*, Eletronic Notes in Theoretical Computer Science **2** (1995), 245–260, Proc. of the SEGRAGRA'95 Workshop on Graph Rewriting and Computation.

[EHK+96] 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*, Handbook of Graph Grammars, Volume 1: Foundations, World Scientific, 1996, to appear.

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

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

[KR96] M. Korff and L. Ribeiro, *Formal relationship between graph grammars and Petri nets*, Lecture Notes in Computer Science **1073** (1996), 288–303.

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

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

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

[Win87] G. Winskel, *Event structures*, Proc. of the Advanced Course on Petri Nets, Springer Verlag, 1987, Lecture Notes in Computer Science 255, pp. 325–392.