SPECIFICATION OF MOBILE CODE SYSTEMS USING GRAPH GRAMMARS *

Fernando Luís Dotti
Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul
fldotti@inf.pucrs.br

Leila Ribeiro
Instituto de Informática, Universidade Federal do Rio Grande do Sul
leila@inf.ufrgs.br

Abstract In this paper we introduce a formal approach for the specification of mobile code systems. This approach is based on graph grammars, that is a formal description technique that is suitable for the description of highly parallel systems, and is intuitive even for non-theoreticians. We define a special class of graph grammars using the concepts of object-based systems and include location information explicitly. Aspects of modularity and execution in an open environment are discussed.

Keywords: Mobile code, Formal specification, Graph grammars

1. INTRODUCTION

The fast and continuous evolution of computing and communication capabilities have resulted in massively distributed computational environments (e.g. Internet). These environments are often called *open environments* and are characterized by: massive geographical distribution; highly dynamic environments; no global control; partial failures; lack of security and high heterogeneity due to the diversity of communication links (delay, throughput), cooperating organizations, services offered, etc. Due to these factors, developing applications for such environments is rather complex and therefore research efforts have been

*This work was partially supported by the projects PLATUS (CNPq and Fapergs), Graphit (CNPq and DLR), QaP-For (Fapergs) and Código Móvel: Suporte e Aplicações (Fapergs).

directed to improve support for development of distributed applications. One such efforts is the research area around *code mobility*[9].

In traditional distributed systems, interacting components are primarily static, i.e., a component remains on the same location during its whole life-cycle and interactions with other components take place by exchanging messages through the communications network. Code mobility can be defined as the capability of dynamically changing the location of an executing component. A mobile component is able to stop its execution on a location, migrate through the network carrying its internal state, and resume its execution on another location. Migration is not transparent to the distributed software developer, it is instead explicitly handled by him/her as part of the application's functionality.

A wide field of applications is foreseen to be scenario for mobile code applications. Network management [9], electronic commerce [18], distributed information retrieval [10], advanced telecommunication services and active networks [28], active documents and workflow management systems are among them. Code mobility is also well suited for the ever growing field of physical node mobility because it is possible to launch remote computations, switch off the local node (e.g., a laptop), switch on the node latter, and receive the results of the remote computations.

The ideas around mobile code and its implementations emerged from a practical approach. Most of the standards, platforms and languages currently available and widely used for the development of this kind of systems[22] reflect this fact: they were constructed in an ad hoc way rather than based on corresponding theoretical investigations. Starting with the π-calculus [19][21], there had been some efforts towards computational models for mobile systems, e.g. based on abstract state machines [17], on mobile ambients [3], and on actors [2]. However, to be used in practical applications, high-level specification languages as well as programming languages whose semantics can be described using such models must be provided. There are some proposals of corresponding programming languages (e.g. KLAIM [6], Mobile UNITY [25], Pict [23], Nomadic Pict[29]), but on the level of specification there is still no formal method that is largely used for mobile systems. When considering mobile code systems, complex distribution aspects, like location and mobility, communication, and, in some cases, failures, are not only implementation issues but also part of the functionality of the system. Therefore, it is necessary to provide the user with abstract constructions to specify and reason about those aspects. Process calculi models and corresponding languages offer a level of abstraction based on processes: a system is viewed as a composition of (interacting) processes. Although this point of view may be adequate for some aspects of a system, it lacks a comprehensive representation of data structure and its distribution within a system. Object-based models, instead, join descriptions of data and processes within one object. In such systems, distribution and concurrency appear naturally with the concept of objects as autonomous entities. In the practice, object-based approaches to specification and programming are widely accepted, although most of the specification methods used are informal of semi-formal. In this paper we we advocate the use of a formal description technique, namely graph grammars, to specify mobile code systems.

Graph Grammars [8][26] is a formal specification formalism that has been already used for the specification of concurrent and distributed systems [27][24]. The basic idea of this formalism is to model the states of a system as graphs and describe the possible state changes as rules (where the left- and right-hand sides are graphs). The behavior of the system is then described via applications of these rules to graphs describing the actual states of a system. Rules operate locally on the state-graph, and therefore it is possible that many rules are applied at the same time. In [20] is was shown that it is possible to encode any π-calculus agent into a graph grammar. There, vertices were used to model channels (or names) and (hyper)arcs were used to model processes, thus obtaining a graphical representation of a term (agent). The reduction rules were then described by graph rules. Using this graph representation, it was possible to provide a true concurrency semantics to the π-calculus. Note that, the description of the system is given by one graph, the rules were just used to implement the reduction of the π-calculus. Here we will adopt an approach in which the system will be modeled by a graph grammar, the initial state of the system will be modeled by a graph (representing the distributed structure of data/objects) and possible evolutions of the system will be described by a (user-specified) set of rules. Graph grammars are appealing as a specification formalism because they are formal, they are based on simple but powerful concepts to describe behavior, and at the same time they have a nice graphical layout that helps even the non-theoreticians understand a graph grammar specification. The latter argument was of particular importance for our choice of using graph grammars as a specification formalism for mobile code systems because it helps for a good acceptance of a method in practice. To make our specification language simpler to use within this application domain, we will define a special class of graph grammars using the concepts of object-based systems and include location information explicitly.

In Sect. 2 we recall the main concepts of graph grammars according to the algebraic approach [7] and introduce the concept of doubly-typed graph, that will be used to model object based and mobile systems.

In Sect. 3 we show how to describe mobile code systems using graph grammars, and in Sect. 4 we discuss (informally) the semantics of such systems. Final remarks can be found in Sect. 5.

2. GRAPH GRAMMARS

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. A graph grammar is composed by a *type graph* (representing the types of vertices and edges allowed in the system), an *initial graph* (representing the initial state of the system) and a *set of rules* (describing the possible state changes that can occur in the system).

We will use the algebraic Single-PushOut (SPO) approach to graph grammars [16][7]. This approach is based on categories of graphs and partial graph morphisms. The kind of graph we will use is called doubly-typed graph. In this Section we will first present the definition of doubly-typed graphs and (partial) morphisms, and then show how the use of this kind of graphs allows for a nice description of object-based systems. Then, in Sect. 3 we will expand this model for mobile code systems. We assume the reader is familiar with basic notions of category theory.

2.1. BASIC NOTIONS

Graphs and homomorphisms: We will consider graphs with vertices labeled by attributes, that are values belonging to carriers sets of an algebra [15]. A graphs is a tuple $G = (V, E, A, s, t, a)$ consisting of two sets V and E (vertices and edges), one algebra A (attributes) and three total functions $s, t : E \to V$ and $a : V \to U(A)$ (source and target of edges, and the attribution function), where $U(A)$ is the disjoint union of carrier sets of A. A homomorphism $g : G \to H$ is a triple $g = (g_V, g_E, g_A)$ consisting of two partial functions g_V and g_E mapping vertices and edges from G to H and one total algebra homomorphism g_A mapping the attribute algebra of G into the attribute algebra of H such that the diagrams below commute (the diagram on the left represents two diagrams, one for the compatibility with the source and other with the target functions). Let **GraphP** be the category of graphs and morphisms as described above (for the formal definition see [15], [14]). Colimits in **GraphP** can be constructed componentwise in the categories of sets and of algebras followed by a construction to remove edges whose source or target vertices have been deleted. For details see [14].

$$
\begin{tikzcd}
E_G \arrow[d, mapsto, "{s^G,t^G}"'] & dom(g_E) \arrow[l, mapsto, "g_E?"'] \arrow[r, mapsto, "g_E"] \arrow[d, phantom, "="] & E_H \arrow[d, mapsto, "{s^H,t^H}"] \\
V_G \arrow[rr, "g_V"'] & & V_H
\end{tikzcd}
\qquad
\begin{tikzcd}
V_G \arrow[d, mapsto, "a^G"'] & dom(g_V) \arrow[l, mapsto, "g_V?"'] \arrow[r, mapsto, "g_V"] \arrow[d, phantom, "="] & V_H \arrow[d, mapsto, "a^H"] \\
A_G \arrow[rr, mapsto, "g_A"'] & & A_H
\end{tikzcd}
$$

Notation 2.1 *The $\mapsto$-arrows denote total morphisms, whereas $\rightarrow$-arrows denote arbitrary morphisms (possibly partial).For a partial function f, $dom(f)$ represents its domain of definition, $f?$ and $f!$ denote the corresponding domain inclusion and domain restriction. Each morphism g in the category* **GraphP** *can also be factorized into corresponding components $g?$ and $g!$.*

Typed Graphs and morphisms: A ***typed graph*** G^T is a tuple $G^T = (G, type^G, T)$ where G and T are graphs and $type^G : G \to T$ is a total graph morphism. A ***typed graph morphism*** $g^T : G^T \to H^T$ between typed graphs G^T and H^T is a pair of graph morphisms $g^T = (g, id_T)$ with $g : G \to H$ and id_T is the identity on T such that $type^G \circ g? = type^H \circ g!$. A typed graph morphism g^T is called an injective/total if g is injective/total. The category of typed graphs and typed graph morphisms over a type graph T is denoted by $\mathbf{TGraphP(T)}$. Colimits in $\mathbf{TGraphP(T)}$ can be constructed componentwise in **GraphP**[14].[1]
Doubly-Typed Graphs and morphisms: As the definitions and results for doubly-typed graphs are analogous to the ones for typed graphs, we will just sketch them here. To get the category of doubly-typed graphs, one has just to take the corresponding definition of typed graphs and substitute the category **GraphP** by $\mathbf{TGraphP(T)}$. A doubly-typed graph $(G^T, type^{G^T}, TG^T)$ will be denoted by $G^{TG \nearrow T}$. The graph TG is called ***(application) type-graph*** and the graph T is called ***model type-graph*** (these names will be motivated in Sect. 2.2). Again, colimits are constructed componentwise in the basis category, i.e., $\mathbf{TGraphP(T)}$ (see [24] for the proofs).

2.2. OBJECT-BASED GRAPH GRAMMARS

In this paper we consider an object-based system as being a system consisting of autonomous entities called *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

[1]If we had used total graph morphisms, the category defined above could have been defined as a comma category (as it was done in [4]). We use an analogous construction for categories with partial morphisms called generalized graph structures [14].

of an object is described through its *reactions* to the receipt of messages (triggers). This reaction may be to change the object's internal state and/or send messages to other objects. An object may perform many (re)actions in parallel.

A way to model object-based systems using graph grammars have been presented [13] inspired in the actor model [1]. The basic idea was to use graph grammars as a (graphical) language to specify actor systems. We will follow the same ideas, although using definitions based on doubly-typed graphs.

To describe an object-based system using graph grammars, the first step is to identify the entities of this specification model and represent them as a graph, called model graph. This is shown in Figure 1. Objects, messages and attributes will be modeled as vertices. A message must have as destiny an object (modeled by the dashed edge) and may have as arguments other objects and/or attributes of data types. An object may know other objects and may have attributes. Each attribute vertex is attributed with a value belonging to an attribute algebra.At this level of abstraction, we will use a variable as attribute to describe that the ◯-vertex may assume any value from the carrier sets of the attribute algebra in some instance graph of this type.[2] 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.

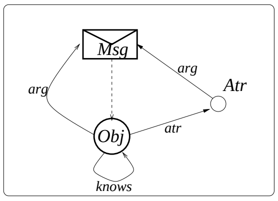

Figure 1 Object Model Graph *OG*

The existence of the model type-graph makes it easier to relate specifications based on different models: once we have related the models at the abstract level (for example, via a morphism between two model type-graphs) we can relate specifications of concrete applications based on that models.

[2]This can be formally described, for example, by using as attribute algebra a term algebra over an order-sorted specification. As this topic is not central for this paper, we will not discuss it in detail.

For each specific object-based system we may have various types of objects and messages that are relevant for that application. Thus, to build a specification for an object-based system using graph grammars one must define the application type-graph. In the next section, Figure 2 shows the model type graph for mobile code systems, and Figure 5(a) shows the application type graph corresponding to one specific mobile code application.

A rule $r : L \to R$ specifies a state change of a system in the following way: all items that are in L must be present at the current state to allow this state change, all items that are mapped from L to R (via the morphism r) will be preserved, all items that are not mapped from L to R will be deleted from the current state and all items that are in R and are not in L (not in the range of r) will be added to obtain the following state. For an object-based graph grammar we will only allow rules that consume elements 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 by 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. 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 exactly one destination. Analogously, each object must have all its attributes.

The following definition is given in a semi-formal way because the corresponding formal definitions require a number of concepts that are not needed elsewhere in this paper and have been omitted.

Definition 2.2 Rule. *Let AG^{OG} be a finite typed-graph, where OG is the object-based model type-graph. A morphism $r : L^{AG\nearrow OG} \to R^{AG\nearrow OG}$ is a **rule scheme** iff L and R are finite and r is injective. A **rule** is a rule scheme which satisfies the following conditions:*
i) *There is exactly one message vertex 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 and all necessary arguments. This is called **message-completeness**. Moreover, all items in L must be connected. This latter condition is to avoid non-local side effects.*
iv) Only the attributes corresponding to the internal state of the destination of a message may appear in L and R.

Given a rule scheme rs and a rule r, a **specialization** *is a pair of total homomorphisms $(left, right)$ mapping the left- and right-hand sides of the rule scheme rs to corresponding left- and right-hand sides of one of the rule r such that $r \circ left = right \circ rs$.*

Now we can define a graph grammar.

Definition 2.3 (Object-based) Graph Grammar. *A* ***graph grammar*** *is a tuple $GG = (AG^{OG}, I, Rules)$ where OG is the object-based model graph, AG^{OG} is a finite typed-graph, called the* **type** *of the grammar, I is a finite, message-complete, object-complete (each object has exactly all attributes specified in AG^{OG}) doubly typed-graph according to AG^{OG}, called the* **initial graph** *of the grammar, and $Rules$ is a finite set of rules according to AG^{OG}.*

Notation 2.4 *When the model graph OG is clear from the context, we will write G^{AG} instead of $G^{AG \nearrow OG}$.*

3. MOBILE CODE SYSTEMS AS GRAPH GRAMMARS

With Mobile Code, the distributed software developer can build applications taking location into consideration. Migration is not transparent, but rather specified explicitly. In order to discuss mobility, some notions have to be first established. The distributed environment is assumed to be a set of places and a set of mobile components.

Places work as possible locations where mobile components can run. They offer basic facilities and the possibility of accessing other components having well defined interfaces (e.g. naming service, event service, etc.). Basic facilities are communications, storage, and processing power. The basic functionality of a place is to accept an incoming mobile component, launch it and support it during run time until it leaves.

Mobile components are software components that may migrate during their execution from place to place to use other local components and basic facilities. A mobile component has internal data or state, code, and a set of meta-data or attributes (e.g. identifier, credentials, originator, operational status, etc.). These components are designed without location transparency and may create other mobile components to run concurrently in the same or different places. Mobile applications can be build from various mobile components.

Example 3.1 *As an example of a mobile code application, consider shops that support places to which mobile components can move to and from. The place of each shop can be configured following its objectives, with services* *relative to the shopping activity, e.g., to answer queries about offers and prices, to reserve and sell products or services, etc.*

Consider that various shops exist in the distributed environment and that a user wants to know which shop has the cheapest price for a product. Various approaches can be followed to achieve that. Here we adopt a simple one based on mobile code.

The user launches a mobile component informing a list of shops (addresses of the places of the different shops) and the product to search for (details of how to obtain those place addresses will not be considered here). The component sequentially visits the various places informed by the user, and in each place interacts with a query service of that shop to discover the price of the product informed by the user. After visiting all places defined by the user, the component returns to the origin place and informs the user where the cheapest price for the desired product was found.

We can model mobile code systems as object-based systems having two kinds of objects: places and components. The specification of places suggested here will encompass only the functionalities that must be provided by a place (and can not be implemented as components). Everything else will be seen as a component. Obviously, some of these components may implement the services provided by the place, and some may be user defined applications executing on that place. This way we will get a flexible model that can be seen as a first step towards a formal specification of active networks: services of a place may be upgraded dynamically without having to recompile or make changes to the kernel of the place.

A component may send messages to other components it knows and also to its own place (for example, requesting to move somewhere else). Places may send messages to the components lying on it and to other places. But if we analyze carefully, we recognize that places have the power to act over components: messages among components can only be sent if the corresponding places agree, the same holds for move requests; if a place crashes, all components running on it are not accessible anymore and stop execution (if the place has recovery schemes, components may resume when the place re-initiate, otherwise, the components disappear). Moreover, the attribute describing in which place a component is in can not be changed by the component itself (like its object identity). As we wish to have a high-level specification language for mobile code systems, we will abstract from this kind of details. For example, a move will be modeled as an atomic operation in which a place is able to modify the argument of a component that corresponds to its placement information. Figure 2 shows the graph that will be used as model type graph to mobile code applications. The arc labeled with *is_in* carries the information about the place a component resides. This argument will be mandatory for each component of a mobile code system specification. The arcs labeled with *to* indicate the receiver of a message. In an instance of type *Msg* the only outgoing arc shall be of kind *to* because each message in the system must have exactly one receiver, and the ingoing arcs represent the parameters of this message (that, according to the model type graph MG may be places, components and/or attributes). The outgoing arcs of components and places represent their internal state. The arcs/vertices are labeled/have different shapes such that we can distinguish the instances of them without having to show explicitly the typing morphism.

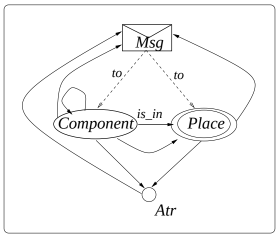

Figure 2 Mobile Code Model Graph MG

As we may see components and places as objects, we can define rules and graph grammars for mobile code systems analogously to the corresponding definitions for object-based systems (Defs. 2.2 and 2.3), obtaining the concept of mobile code graph grammars. In this case, we would have to relax condition v of Def. 2.2 to allow the deletion/insertion of *is_in* attributes of components by the places they are in.

When designing a mobile code system, a user specifies components. This specification is based on his/her expectations about the behavior of the places involved (because places are responsible for handling communication and move requests). Thus, it is imperative that he has an abstract and precise description of this behavior. In the following we will provide such a description, and then show how (mobile) components may be specified using graph grammars and illustrate this by an example.

3.1. SPECIFICATION OF PLACES

A place basically provides physical resources (like memory, CPU) and communication services to the components running on it. Internally, a place manages the resources and communication, that is, it implements a network operating system. There are two kinds of communication that involve places: communication between components and places (place →

component: messages to start and finish execution, etc.; component $\rightarrow$ place: messages to move, find place name server, etc.), and communication between places (messages that form the communication protocol between places).

The *resources and their administration* can be modeled very abstractly using an algebraic specification. One of the operations, *accept* : *State*, *ReqList* $\rightarrow$ *Bool*, may evaluate whether the place may or may not accept a new component trying to move to it (based probably on the number of applications actually on this place, the properties of the place and the requirements of the component).

Communication can be modeled using graph rules. Figure 3 presents rule schemes to model this, actual communication will be represented by a rule that is a specialization of one of these rule schemes. Rule scheme *Send* specifies that components lying on the same or different places may communicate (note that this rule scheme specifies local and remote message passing, since the specialization homomorphism is not required to be injective). Rule scheme *ServiceRequest* describes that components may send messages to the place in which they are executing.

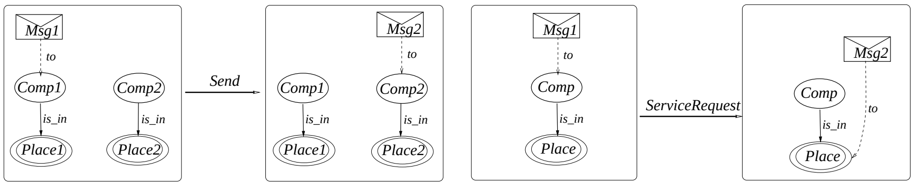

Figure 3 Message Passing Rule Schemes

Besides computational resources and communication, a place may be able to provide move services. The rules describing such service are shown in Figure 4. Message $Move$ can be sent by a component to a place ($Orig$) asking to do a move to place $Dest$. The attribute req represents the requirements of the component. This is a list of services/attributes needed for the execution of this component. When a place ($Orig$) receives a $Move$ message from a component, it sends a $MoveReq$ message to the place that shall receive the component having as attributes the component that wants to move, its requirements and the origin place (rule $RequestMove$). Rule $Move$ specifies what happens when a place receives a $MoveReq$ message and decides to accept the component[3]:

[3]The condition written under the rule arc describe conditions that has to be satisfied for the rule to be applied. Formally this behavior is obtained by considering as attribute algebra for the left- and right-hand side an algebra that satisfies this condition.

the place sends a message to place $Orig$ to inform that the component $Comp1$ has successfully moved, sends a message to $Comp1$ telling it to continue its execution, and changes its internal state to include this new component. Messages $DenyMove$ and $NotMove$ specify the actions to be performed when a move request does not succeed: the destination place informs the origin place the move was denied, and the latter forwards this information to the corresponding component.

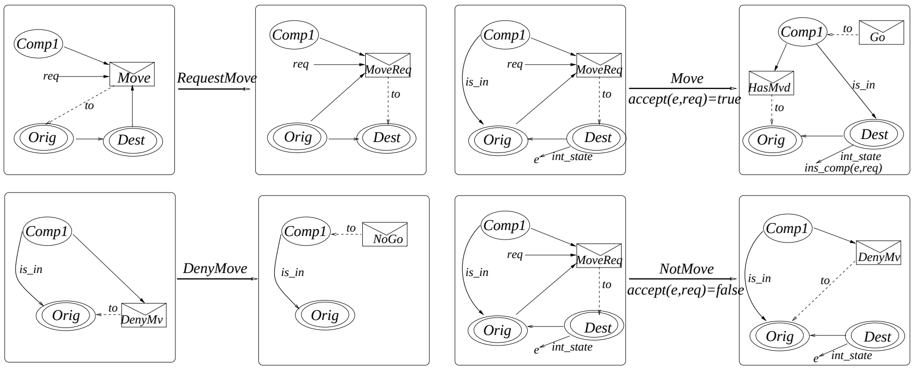

Figure 4 Move Rules $MRules$

3.2. *SPECIFICATION OF MOBILE COMPONENTS*

To build a specification of a mobile component, the first step is to define the application model graph AG to be used. This graph will be typed over the mobile code model type graph MG (Figure 2) and contains the types of components that are necessary for this application, the messages that are exchanged, and the internal structure the component being developed. Then we can define the initial state of the component (that is a doubly-typed graph over AG^{MG}), and the set of rules that describe how this component may evolve. These components typically rely heavily on message passing, and therefore the behavior of the application can be suitably described by graph rules [24][11]. To be able to model mobile code systems faithfully, this grammar must fulfill additional requirements: each component vertex must be connected to exactly one place by an *is_in* arc, and each rule that generates messages must be a specialization of one of the rules of Figure 3 (because these rules describe the possible message passing strategies of the places).

Each component in an open system may use other components and be used by them. When choosing a component to perform some task, it is important that the behavior of this used component is in accordance to the desired by the user component. To model this we will include in the specification of a component an abstract description of the desired behavior of the components that are used by this component. This will be called *import* interface.

Definition 3.2 *Mobile Component. A mobile component is a tuple* $MC = (T, Ini, Rules, Imp)$ *where* $(T, Ini, Rules)$ *is a graph grammar with type graph* T^{MG}, Imp *is a set of rules over* T^{MG} *such that the following conditions are satisfied:*
i) Each component vertex must be connected to exactly one place by an is_in *arc.*
ii) Each rule that generate messages must be a specialization of one of the rule schemes of Figure 3.
iii) Each rule in Imp *contains no attributes for places or components.*

Example 3.3 *Now example 3.1 will be modeled using graph grammars. The application type graph (see Figure 5(a)) shows the types of entities involved. A mobile component* (MC) *searches for the* $bestPrice$ *of a product with name given by* $prodName$ *in a number of places* $numPlaces$. MC *holds also registers to each place to be visited* ($visit$)*, to the* $origin$ *place, to where it* $is - in$ *currently, and to the place with the* $best$ *offer. The scenario described also involves information services* (ISs) *that the* MC *may use (consult) during its journey. Details of* IS *are not given here. When* MC *completes its task it should also give back an answer to the user* U. *The identifiers* $real$, $natural$ *and* $string$ *stand for variables of the corresponding data types (we use a term algebra as attribute algebra). Actually all messages (together with their arguments) involved in this application are also part of the application type graph (although not drawn in Figure 5).*

The initial graph shown in Figure 5(b) is one of many possible initial graphs for this example. In this case MC *will look for the best price of product* product1 *in places* P1 *to* P3, *departing from* P_orig. *In each of these places there are* ISs *located, which* MC *can use.*

Figure 6 shows the rules that describe the dynamic behavior of the system. Considering the initial state described above, the first rule that will be applied is rule $MoveNext$. *This rule can be applied in three different ways, depending on the place chosen to be visited. In the graph grammar semantics (that will be discussed in the next section), this choice is nondeterministic. The application of this rule will generate a* $Move$ *message having as argument the place chosen to be visited. The reader may verify that now rules* $MoveNext$, $QueryPrice$ *and* $Update\&Proceed$ *or* $Proceed$ *will be applied until there are no more places to visit. Note that rules that belong to the specification of places and other components, for example,*

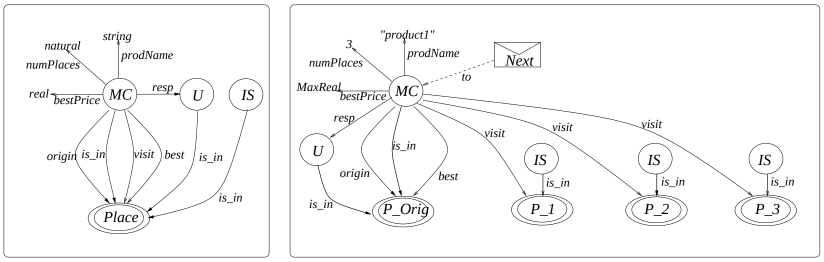

Figure 5 (a)Application Type Graph - (b) Initial Graph

IS, were not described here. The move rules belong to the basic functionality offered by places. The import rules for this component(not shown here) would describe, for example, that in response to a $cost?$ message the IS component will eventually deliver a $price$ message.

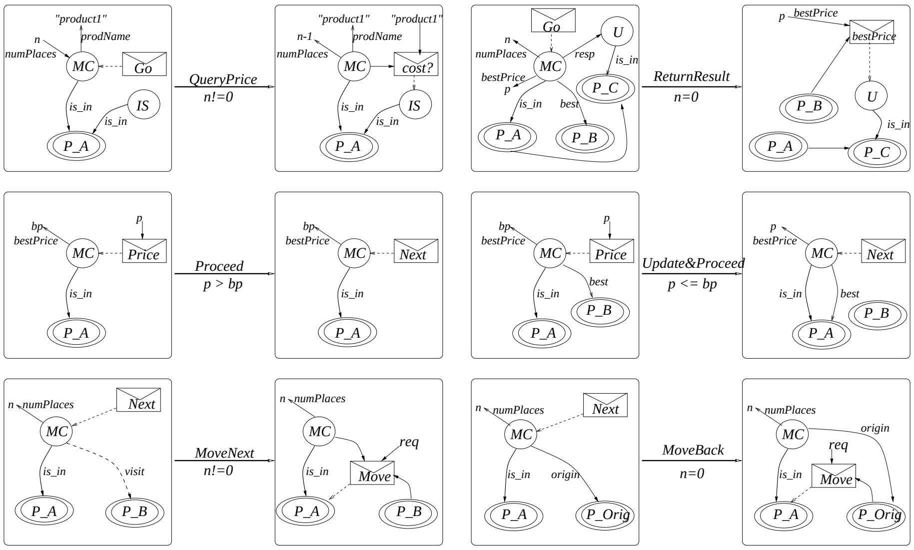

Figure 6 Rules

4. SEMANTICAL ASPECTS

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 preserves 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 to a graph 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.

Definition 4.1 Match, Derivation Step. *Let $r : L^{AG} \to R^{AG}$ be a rule and G^{AG} be a (doubly-typed) graph. A* ***match*** *of r in G is a total doubly-typed graph morphism $m : L^{AG} \to G^{AG}$. A* ***derivation step*** *of a rule r at match m, denoted by $G \Rightarrow^{r,m} H$, is the pushout of r and m in the category* $\mathbf{DTGraphP}(\mathbf{AG^{MG}})$.

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 7(a) Illustrates a *sequential derivation* for a grammar with starting graph I. In this derivation we have a total order ($<$) on derivation steps ($s1 < s2 < s3$) that denotes the sequence in which they have occurred in this computation. If we make a suitable gluing[4] of all intermediate graphs of this derivation, we obtain a structure called *concurrent derivation* (Figure 7(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). A true concurrency semantics for graph grammars can be also described by an unfolding construction that gives us a partial order of derivation steps [24]. The unfolding construction encompasses informations about all possible computations that are described by the given graph grammar.

Open systems are highly dynamic, in the sense that places and services may be created, deleted, become reachable and unreachable at any moment. Thus, a suitable semantics for such systems must take these characteristics into account. Note that, to define a semantics for a mobile component, it is not enough to consider the behavior of this component

[4]This gluing is actually a colimit of a diagram in the category of doubly-typed graphs [14, 24].

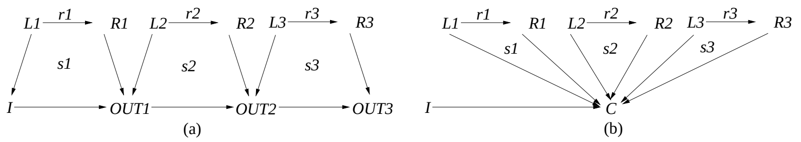

Figure 7 (a) Sequential derivation (b) Concurrent Derivation

in every possible network configuration, but we must also consider its behavior in changing network configurations, that is, in a configuration that may change while the component is executing. To capture the dynamics of the environment we will use a set of rules that describes such changes. Some of these rules are depicted in Figure 8. Rules $createPlace/createComp$ and $destroyPlace/destroyComp$ are used to create and delete places between places/components from the network. Moreover, a rule that changes (consistently) the internal state of a place is needed. The semantics of a mobile component shall consider not just the rules that describe the behavior of the component itself, but also and the rules that describe (abstractly) the behavior of the used components (described in the Import interface of this component), the move rules of the place, and the rules that may create/destroy/modify places and used components. As a mobile component only uses the kinds of components that are described in its specification, the fact that other kinds of components may exist in the network are irrelevant for its behavior and must therefore not be considered. With these rules we may obtain all possible computations of this system taking as initial state a graph consisting of the initial graph of the definition of the component itself plus an initial state for the places involved. We can consider a trivial initial state because other possible initial states may be obtained by the dynamic rule that changes the state of places.

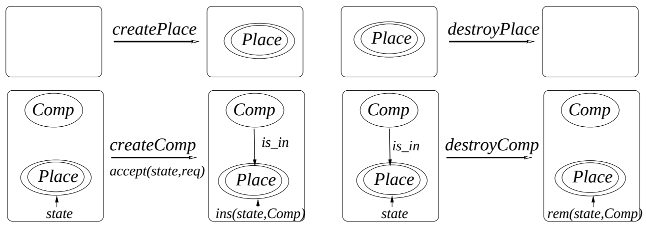

Figure 8 Dynamic Rules

5. CONCLUSION

In this paper we have introduced a formal approach for the specification of mobile code systems. This approach is based on graph grammars, and the main idea is to model places and components as vertices (with attributes), and their behavior as rules specifying the desired actions that shall be performed when a place/component receives some kind of message. Graph grammars seem to be well suited for the specification of such systems as they match a series of important characteristics:

Concurrency: Applications are composed by various mobile components which may be active. Therefore a mobile application is inherently concurrent. Graph grammars is a formal description technique that is suitable for the description of highly parallel systems. Concurrency is not specified explicitly, as, for example, in process calculi, but rather obtained as a consequence of independence of actions (rule applications). As discussed in [20], this approach can be useful to guide an efficient implementation of the system (since it allows to derive information concerning the maximal parallelism taking into account the use of shared data), as well as allow for better strategies to debug a system (since it permits to trace back all the actions that have influenced the occurrence of an undesired state). From the concurrency semantics of a graph grammar we can obtain causal dependency and conflict relationships that can be used to reason about the parallelism of a system (actually, due to the possibility of read-access to items, we need also a third relationship, called weak – or asymmetric – conflict [12]). Even without need to consider all computations of the semantical model, an analysis of the potential and conflict relationships among rules may already lead to useful statements about the system.

No global state: As the environment is very dynamic, mobile code components rely less on a global notion of state and more in a local notion of state, tending to have more autonomy to decide the actions to perform in a wider range of situations as if compared to fixed systems. This is naturally modeled with graph grammars as the distributed state is modeled by a graph and each rule application affects only a part of this graph (rule applications may have only local effects). The description of the state as a graph highlights the relationships among the components (since these can be explicitly represented as edges connecting the involved components). Such a representation considerably eases the understanding of the specification.

Openness: In the massively distributed environment considered, places can be created and deleted (shut down) following policies from different organizations and offer different sets of services. Components have independent life-cycles and enter and leave places, becoming reachable or unreachable to other components. This dynamic aspects were considered by the discussion about semantics in Sect. 4.

Fault tolerance: As the environment considered is a massively distributed one, partial failures may be present/arise. Many times it is important/necessary to consider the possibility of infra-structural failures in the applications. As the sketched semantical model takes into consideration that the environment may fail, we believe that is will be possible to reason about the fault tolerance aspects of a component by analyzing the computations of the corresponding graph grammar. This is a very interesting subject of future work.

Modularity: While modeling applications for open systems it is very important to have a sound definition of the functionality offered by the application and used from other applications. In our approach this was considered using the notions of import interface, which is part of the specification of each component. However we still have to consider the impact of this interface in defining semantical models for mobile applications (here we have only discussed the semantics of one mobile component). Moreover, it is also important to have an export interface describing the services offered by a component at an abstract level. This would require a notion of satisfaction of a specification (rule) by a graph grammar, that is, we would have to guarantee that the rules specifying the behavior of a given component would always lead to a certain state/trigger some message. This topic is currently under investigation, as it is also needed to assure that an imported component presents the required behavior. Some work in this direction have been done in [20], where different notions of observational equivalences for graph grammars were defined. Another topic would be to consider whether richer place hierarchies, as provided by mobile ambients [3], can be integrated to our approach to improve the structuring of the system.

In addition to all those aspects, graph grammars are intuitive even for non-theoreticians. Besides being an unambiguous description of the system, a graph grammar allows for formal analysis of properties of this system, like complexity (for example, in terms of number of rule applications necessary to perform some task), relationships (causality, conflict) among rules, correctness, etc. In the project PLATUS [5] we are implementing an environment in which it will be possible also to make simulation of graph grammar specifications. This environment may be used for the specification of mobile code systems as described here.

References

[1] Agha, G. *Actors: a model for concurrent computation in distributed systems,* MIT Press, 1986.

[2] Agha, G. and Kim, W. *Actors: A unifying model for parallel and distributed computing.* Journal of systems architecture 45, 1999, pp. 1263–1277.

[3] Cardelli, L. and Gordon, A., *Mobile ambients,* Foundations of Software Science and Computational Structures, Lecture Notes in Computer Science, vol. 1378, Springer, 1998, pp. 140–155.

[4] Corradini, A., Montanari, U. and Rossi, F. *Graph processes*, Fundamentae Informatica, vol. 26, no. 3-4, 1996, pp. 241–265.

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

[6] De Nicola, R., Ferrari, G. and Pugliese, R. *KLAIM: A kernel language for agents interaction and mobility*, Transactions On Software Engineering, IEEE , vol. 24, 1998, pp. 315–330.

[7] Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A. and Corradini, A., *Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach*, in [26], pp. 247–312.

[8] Ehrig, H., *Introduction to the algebraic theory of graph grammars*, Lecture Notes in Computer Science, vol. 73, Springer, 1979, pp. 1–69.

[9] Fuggeta, A., Picco, G. and Vigna, G., *Understanding Code Mobility*, Trans. On Software Engineering, IEEE, vol. 24, 1998, pp. 342–361.

[10] Knudsen, P., *Comparing Two Distributed Computing Paradigms - A Performance Case Study*, M. Sc. thesis, University of Troms, 1995.

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

[12] Korff, M. and Ribeiro, L., *True concurrency = Interleaving + weak conflict*, Electronic Notes in Theoretical Computer Science, vol.14, 1998.

[13] Korff, M., *True concurrency semantics for single pushout graph transformations with applications to actor systems*, Information Systems - Correctness and Reusability, World Scientific, 1995, pp. 33–50.

[14] Korff, M. *Generalized graph structures with applications to concurrent object-oriented systems*, Ph.D. thesis, Technical University of Berlin, 1995.

[15] Löwe, M., Korff, M. and Wagner, A., *An algebraic framework for the transformation of attributed graphs*, Term Graph Rewriting: Theory and Practice, John Wiley & Sons, 1993, pp. 185–199.

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

[17] Maia, M. and Bigonha, R. *Interaction based semantics for mobile objects*, In Proc. of the III Brazilian Symposium on Programming Languages, 1999.

[18] Merz, M. and Lamersdorf, W., *Agents, Services and Electronic Markets: How Do They Integrate?*, In Proc. of the International Conference On Distributed Platforms,IFIP/IEEE, 1996.

[19] Milner, R. and Parrow, J., *A calculus for mobile processes I,* Information and Computation, vol. 100, 1992, pp. 1–40.

[20] Montanari, U., Pistore, M. and Rossi, F. Modeling concurrent, mobile and coordinated systems via graph transformations, *The Handbook of Graph Grammars, vol. 3: Concurrency, Parallelism and Distribution*, World Scientific, 1999, pp. 189–268.

[21] Milner, R., Parrow, J., *and Walker, D., A calculus for mobile processes II,* Information and Computation, vol. 100, 1992, pp. 41–77.

[22] Perdikeas, M., Chatzipapadopoulos, F., Venieris, I. and Marino, G., *Mobile Agent Standards and Available Platforms.*, Computer Networks, vol. 31, 1999, pp. 1999-2016.

[23] Pierce, B. and Turner, D., *Pict: a programming language based on the pi-calculus*, Tech. Report 476, Indiana University, 1997.

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

[25] Roman, G., McCann, P. and Plun, J., *Mobile UNITY: reasoning and specification in mobile computing*, ACM TOSEM, vol. 6, no. 3, 1997, pp. 250-282.

[26] Rozenberg, G. (editor), *The Handbook of Graph Grammars, vol. 1: Foundations*, World Scientific, 1997.

[27] G. Taentzer, *Parallel and distributed graph transformation: Formal description and application to communication-based systems*, Ph.D. thesis, Technical University of Berlin, 1996.

[28] Tennenthouse, D. L., Wetherall, D. J., Smith, J. M., Minden, G. J. and Shiaskie, W., *A Survey of Active Network Research*, IEEE Communication Magazine, January 1997.

[29] Wojciechowski, P., Sewell, P., *Nomadic pict: language and infrastructure design for mobile agents*, In Proc. of the ASA/MA'99, 1999.