Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002
©2002 Society for Design and Process Science

A FRAMEWORK FOR SUPPORTING THE DEVELOPMENT OF COORECT MOBILE APPLICATIONS BASED ON

Fernando Luís Dotti, Lucio Mauro Duarte[+]
[fldotti, lduarte] @inf.pucrs.br
Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática - PPGCC
Av. Ipiranga, 6681 - CEP 90619-900 - Porto Alegre - RS - Brazil

Flávio M. Assis Silva[++], Aline M. Santos Andrade
[fassis , aline] @ufba.br
Universidade Federal da Bahia
LaSiD - Laboratório de Sistemas Distribuídos
Av.Adhemar de Barros, S/N - CEP 40170-110 - Salvador - BA - Brazil

ABSTRACT

The development of complex mobile applications, i.e., applications based on software components that can migrate through a distributed environment, is a difficult task. Mobility increases the complexity of testing applications in real conditions, introducing a new dimension in the behavior of programs. This difficulty becomes even higher when an application is composed of several different mobile components, which might migrate in an autonomous way, independently one from the others. In this context the use of formal methods becomes increasingly important, since they provide a sound and precise way of specifying applications and enable the development of tools for correctness verification. The goal of the project ForMOS - Formal Methods for Mobile Code in Open Systems - is to analyze the application of formal methods to support the generation of correct mobile applications and to provide an integrated infrastructure for the development of correct mobile systems. This paper introduces a framework based on Object-based Graph Grammars that is being developed in the context of this project and describes two of its tools: a simulator and an application code generator.

* This work is partially supported by projects: *ForMOS* (Formal Methods for Mobile Code in Open Systems) - grants CNPq/ProTeM-Brazil and FAPERGS-Brazil; I*Q-Mobile* (Improving the Quality of Open Systems with Code Mobility through Rigorous Development) - grant CNPq/CNR (Brazil/Italy); and *PLATUS* (Formal Development and Simulation of Reactive Systems) - grant FAPERGS-Brazil.

[+] The work of this author was partially supported by CAPES, Brazil

[++] The work of this author was partially supported by CNPQ, Brazil

1. INTRODUCTION

Supported by the current state of development of processing and communication technologies, today's distributed environments such as the Internet allow complex forms of integration and cooperation between systems and organizations. Frequently referred to as *open environments*, these environments are typically characterized by: massive geographical distribution; high dynamism; no global control; partial failures; lack of security; high heterogeneity; and presence of autonomous organizations.

Building distributed applications for such environments is a complex task. Research efforts have been directed to managing this complexity by, for example, developing new paradigms, improving theory, and developing more powerful and flexible technologies for distributed applications. Within this context, *code mobility* (Fuggetta, Picco and Vigna, 1998) emerged as a promising paradigm for designing and building distributed systems.

Code mobility refers to the concepts and technologies that enable pieces of code to be sent through the network to be executed at remote locations. An example of technology enabling code mobility are the *mobile agent* systems. A mobile agent is a piece of software that is able to autonomously migrate through the nodes of a distributed system. A mobile agent might migrate multiple times during its execution. Mobile agents have received special attention of the research community and of the industry due to its flexibility and potential use in various application fields, e.g. network management, electronic commerce, distributed information retrieval, advanced telecommunication services, active networks, and workflow management systems (Fuggetta, Picco and Vigna, 1998). Code mobility is also well suited for the field of physical node mobility because of its support for disconnected operation (agents might execute activities on a network on behalf of a disconnected host).

However, there are important research issues related to the development of mobile agent-based applications for open environments that must still be further investigated. Among these issues are the development of concepts and tools for guaranteeing security, reliability, and - of main interest in this paper - *correctness* of these applications.

Guaranteeing correctness of mobile-agent based applications is a non-trivial task. Mobile applications are inherently distributed and concurrent. Additionally, a mobile application can be structured in parts that can migrate autonomously through the distributed environment, completely independent one from the other. In open environments, not only the mobility aspect makes the task of guaranteeing correctness of an application harder, but also aspects such as the existence of organizational boundaries, the unpredictability of failure patterns in such environments, among others.

The use of formal methods is one of the ways of addressing the generation of correct systems, and we concentrate on this approach in this paper. Formal methods have been traditionally applied to concurrent and distributed systems both as a tool for precisely describing the interactions between parts of a system as well as for enabling the development of automatic verification tools.

Although much effort has been applied to develop formal methods for mobile systems (Milner, 1999; Cardelli and Gordon, 1998; Fournet, Gonthier, Lévy, Maranget and Rémy, 1996; Sewell, Wojciechowski and Pierce, 1999), an integrated framework for supporting the development of correct mobile applications is still missing. One of the objectives of the ForMOS project (*Formal Methods for Mobile Applications in Open Environments*) is to develop such a framework. The innovative aspect of the framework we are developing is the use of *Object-Based Graph Grammars* (Dotti and Ribeiro, 2000) as the underlying unifying formal method *for a set of* integrated tools. Each of the tools supports a way of addressing the generation of correct mobile applications. The framework encompasses a simulation tool and a code generator, and efforts are being invested towards a verification tool. In this paper we describe the current state of the simulation tool and of the code generator.

The paper is structured as follows. In section 2 we describe the related work. In section 3 we describe the framework tools (simulation tool and code generator). Finally we conclude the paper in section 4.

2. RELATED WORK

A set of formal methods has been proposed for the specification of mobile systems, such as π-calculus (Milner, 1999), mobile ambients (Cardelli and Gordon, 1998), Distributed Join Calculus (Fournet, Gonthier, Lévy, Maranget and Rémy, 1996), Nomadic π-calculus (Sewell, Wojciechowski and Pierce, 1999), among others. The formalisms differ in the way they model mobility or in their ability to explicitly model specific aspects of the development of mobile applications/protocols.

The π-calculus, for example, allows the specification of changes in the configuration of communication channels between processes. The location of a process at a given instant can be specified by the configuration of connections (communication channels) it has with other processes at that instant. As long as a mobile process migrates in an environment, its connections with other processes might change. Mobility is thus represented implicitly.

In Mobile Ambients, Nomadic π-calculus and Distributed Join Calculus explicit notions of agents and locations exist. Mobile ambients explicitly takes into consideration administrative domains. The main abstractions are the so-called *ambients*, where agents are contained in and which can migrate in the distributed environment. Explicit primitives for modelling permissions in the environment are provided (so-called *capabilities*). The Nomadic π-calculus is a variation of π-calculus in which lower level basic and simple constructs for specifying mobility are defined. This formalism is being used to analyze and develop ways of transparent communication between mobile agents. The Distributed Join Calculus is a formalism that incorporates a simple model of failure. This model defines a simple semantic for the effects of failures on locations.

The framework described in this paper is based on the use of a special kind of Graph Grammars to specify mobile code applications, called *Object-Based Graph Grammars (OBGG)*, as proposed in a previous work of one of the authors of this paper (Dotti and Ribeiro, 2000). In (Dotti and Ribeiro, 2000) OBGGs were extended to become a suitable specification technique for mobile code applications. This extension introduced the notions of locality and mobility in OBGGs. Graph grammars are quite 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 specification formalism for mobile code systems because it helps for a good acceptance of a method in practice.

We are developing a framework where different tools for helping the generation of correct mobile applications coexist (a simulator, a code generator and - under analysis - a verification tool). These methods and tools are all based on a single formalism, object-based graph grammars. Although there exist programming languages and verification tools based on other formal methods that support mobility, such as Pict (Pierce and Turner, 1997), Nomadic Pict (Wojciechowski and Sewell 1999) (programming languages based on π-calculus and nomadic π-calculus, respectively) or the Mobility Workbench (Victor, 1994) (a tool for manipulating and analyzing mobile systems specified in (polyadic) π-calculus), we are not aware of the existence of an integrated system based on a single formalism such as ours.

3. THE FRAMEWORK

In this section, we briefly outline object-based graph grammars (section 3.1), give an example of its utilization (section 3.2), and present the simulation tool (section 3.3) and the code generation tool (section 3.4).

3.1. OBJECT-BASED GRAPH GRAMMARS (OBGGs)

OBGGs are a restricted form of graph grammars, where the notion of *entities* that communicate through messages is introduced (Dotti and Ribeiro 2000). In terms of modelling, an entity is similar to objects in object oriented design. The basic notion of OBGGs, as for any graph grammar, is that the state of a system can be represented by a graph, the *system state graph*. From the initial state of the system, given by an *initial graph*, the application of *rules* successively changes the system state. Since the system state is given by a graph, changing it means to change the graph configuration.

The system state graph and the initial graph are graphs where the nodes are represented by entities or messages. Entities and messages may have *attributes*. An attribute of an entity or message might be an entity or an elementary data type (integer, real, etc.). Attributes are represented as labeled arcs in graph grammars (in the graphical notation adopted in this paper attributes are not always represented by edges, see section 3.2 for further details on the graphical notation).

An OBGG specification is composed by:

- for each entity, a *type graph* and the entity's *initial graph*;
- a specification of the system *initial graph;*
- a set of *rules*.

Each of these components is described in the following.

Entity's Type Graph and Initial Graph

In an application specification, each entity involved is specified by its type graph and its initial graph. A type graph describes the entity, the names and types of its attributes, and the types of *messages* that the entity may send or receive. The entity's initial graph specifies the value of each attribute of the entity when one instance of this entity is created.

System Initial Graph

The system initial graph describes the state of the system when a computation specified with OBGG starts. The composition of the system initial graph is done by taking the entity initial graph for each instance that must be present in the beginning of the system and completing them with the expected instantiation values to build the initial state of the system.

Rules

The behavior of the system is determined by a set of *rules*. Graphically a rule is specified by two graphs, connected by an arrow (see, for example, figure 5). A rule can be applied (executed) when the graph at its left side is a subgraph of the current system state graph. Optionally an additional condition can be specified that restricts the execution of the rule (graphically, the condition is specified below the arrow linking the left to the right side of a rule – see, for example, figure 5). If a condition is specified, a rule can only be applied when the condition is true. When a rule is applied (executed), the system state graph is changed according to the right side of the rule. Components of the graph (vertices, edges and messages) that exist at the left side and do not exist at the right side are excluded from the graph, while components that exist at the right side and do not exist at the left side are included.

A computation is triggered by the reception of a message by an entity. So, each rule must always have an incoming message to one entity at its left side, and at the right side this message must be consumed - meaning that the computation described by the transformation rule took place due to the reception of the respective message by the entity. The transformations from the left to the right side may only modify attributes of the entity, which receives the message, to ensure encapsulation. If the computation upon receiving a message should affect other entities, then this should be modeled as the generation of messages to other entities at the right side.

The execution of rules is also dependent on the occurrence of *conflicts*. A conflict exists involving various enabled rules when their transformations modify both a same set of items. Rules can be applied simultaneously when there is no *conflict*. In case of conflict among various enabled rules, the choice of which rule is applied is non-deterministic. Since we are dealing with OBGG and the left side of a rule is restricted to refer to only one entity, conflicts never involve various entities.

Extending OBGGs for specifying mobile applications

To represent mobile systems, OBGGs were extended to include two specialized entities (Dotti and Ribeiro, 2000): *places* and *mobile components*. *Places* represent the possible locations where mobile components may execute, offering basic functions such as storage and communication facilities, computational power, and access to services. Furthermore, places offer message passing and migration services to mobile components. *Mobile components* represent software components that can migrate from place to place during their execution, using resources and services from the places they visit.

In the following section an example specification in OBGG is given. In this example we focus on the specifier point of view, and assume the basic functionality of *Place* and *Mobile Component* as provided, i.e., we assume that Mobile Components may move from Place to Place using the move service that Places offer. For further details on the formalization of this basic service as well as on the message passing characteristics, please refer to (Dotti and Ribeiro, 2000).

3.2. AN EXAMPLE: AN AGENT LOOKING FOR THE BEST PRICE OF A PRODUCT

In this example we consider an electronic market scenario composed by a set of places that can be visited by mobile agents. Each place has an *Information Service* that provides the price of a product. In this scenario, a mobile agent is used on behalf of a customer to look for the place, out of a set of places to be visited, that has the best price for a given product.

This example involves four entities: Place, MC (Mobile Component), IS (Information Service) and Customer. Figures 1 to 4 show the type graphs of these entities. As can be seen, we adopt a convention for specifying type graphs:

- an entity is represented by a rectangle, with its name on the top;
- entities that extend Places have double border lines;
- entities that extend Mobile Components have single border lines;
- messages are represented by the shaded symbol shown in the figures;
- attributes of an entity that are nodes of the graph are connected to the entity by an edge, while attributes that are elementary data types are listed inside the rectangle;
- each entity is associated with a number, written inside a circle. This number can be used to refer to the entity in different parts of the specification;
- the messages that the entity might generate, as well as other entities used (referred to by numbers) in the specification of its type graph, are represented below the dotted line in the figures.

Thus, in Figure 1 it is specified that the entity *Place* has two attributes: *hostedIS* and *nextPlace*. The attribute *hostedIS* refers to the Information Service of the Place (the number 3 in the circle is the number associated with the entity IS, shown below the dotted line). The attribute *nextPlace* is a reference to another place (the number 1 in the circle is the number associated with a *Place* entity). This parameter is used to build a sequence of places, used by the agent to visit the places. Figure 1 shows also that a *Place* entity might receive *NextVisit* and *GetIS* messages (represented by the shaded symbols). Each of these messages might be sent by an *MC* entity. A *NextVisit* message is sent by an MC to the place when it wants to move to the place indicated by the attribute *nextPlace*. The message *GetIS* is sent by an MC to a place to get a reference to its IS.

Please note that a message can carry information about its sender as one of its attributes, but it is not necessary. A message that only reports some information to the receiver will not need to carry data about the sender. On the other hand, a service request message needs to identify the sender such that the response can be addressed to the requiring entity.

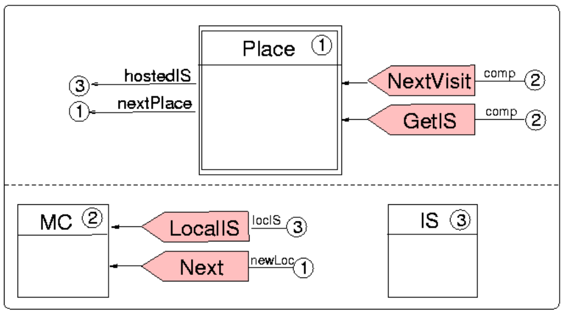

Fig. 1: Type Graph for entity Place

Figure 2 shows the type graph for the *Mobile Component* entity. An MC has to visit *numPlaces* places to discover the best price (stored in *bestPrice* - real) for a product with name *prodName*. The attribute *bestPrLoc*, which refers to a node of type *Place* of the graph, is used to store a reference to the place with the best price. An MC also has parameters to indicate its origin place (*origin)*, its current location (*location)*, and the customer to which it has to report the results of the search on the market (*respCustomer)*.

The *location* of an MC is an attribute of every mobile component, independently of its application. It is modified when a move operation takes place.

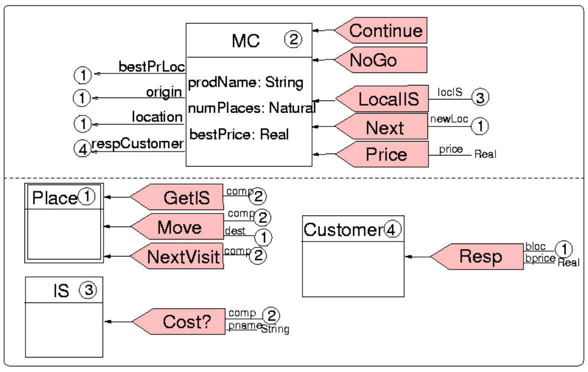

Fig. 2: Type Graph for a Mobile Component

As can be seen in Figure 2, beyond a *GetIS* message (explained above), an MC might issue a *Move* message to the place where it resides, asking to move to a destination *dest* (parameter of the *Move* message), and referencing itself as the component to be moved. An MC might also send messages to the IS and Customer entities (*Cost?*, *Resp)*. When these message are used is explained below, when the type graphs of their receiving entities are described. Four messages can be received by an MC (*Continue*, *LocalIS*, *Next*, and *Price*). The roles of all these messages are explained below, when the rules are described.

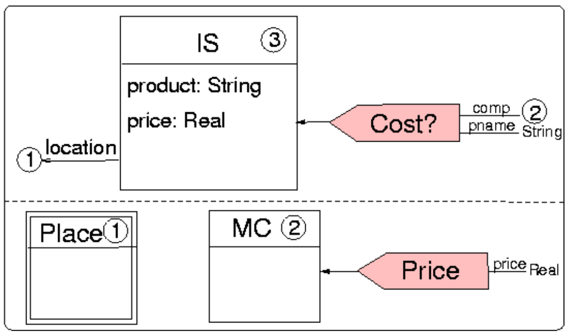

Fig. 3: Type Graph for an Information Service (IS)

The type graph of IS is depicted in Figure 3. An IS has as attributes the place where it resides (*location*), the identification of a product (*product*), and the price of this product (*price*). The IS is a simple service that, when inquired about the cost (message *Cost?*) of *product* by a component MC, it answers with the value of *price* (sending message *Price*).

Finally, the entity *Customer* (see Figure 4) has attributes for storing the best price found (*price*), the location which offers the best price *(bestPriceLocation)* and the location where the customer is (*myLocation*). A Customer entity might receive the message *Resp*, which is the response of MC, after it has finished the search.

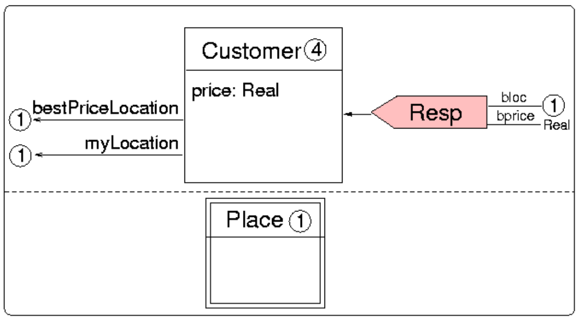

Fig. 4: Type Graph for entity Customer

The specification of the behavior of each entity is made by defining rules. Due to space limitations we do not provide here the complete set of rules for the example, but only some of them to illustrate the use of the formalism. These rules are shown in figures 5 to 8.

The rule in figure 5 states that if an MC receives a *Continue* message (indicated in the graph at left side of the rule) and the number of places to visit (*numPlaces*) is not zero (condition under the arrow linking the left and right side of the rule), then MC will consume that incoming message and generate a message (*GetIS*) to the current place (*P_A*) asking for the IS in that place.

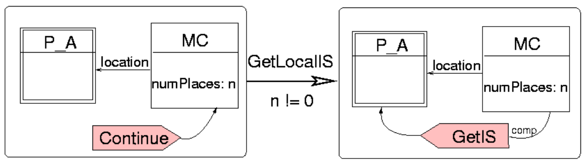

Fig. 5: GetLocalIS rule

As a result of the *GetIS* message sent by the MC in the *GetLocalIS*, a message *LocalIS* is eventually sent to the MC. This message has as parameter the IS in the local place. The rule that generates this message is not shown here. As depicted in rule *QueryPrice* in Figure 6, when an MC receives the *LocalIS* message, it asks the IS about the cost of the product it is searching for (message *Cost?*).

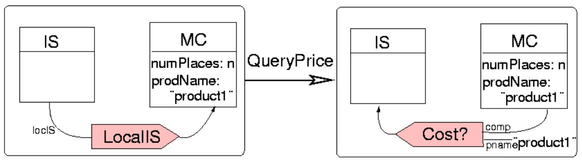

Fig. 6: QueryPrice rule

When an MC discovers a lower price for the product, then it has to update its *bestPrice* attribute to the new price found out and the best price location *bestPrLoc* to refer to the current place, as depicted in Figure 7 (*Px* and *Py* are places). Also, MC asks the current place for the next place to visit (for simplification purposes, in our scenario the current place has a reference to the next place to be visited).

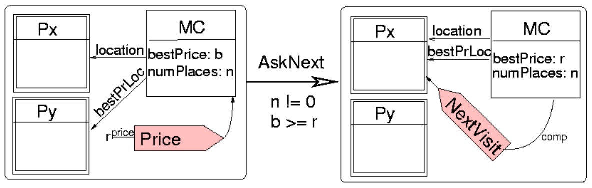

Fig. 7: AskNext rule

The current place informs the next place to continue the search with the message *Next* - see Figure 8.

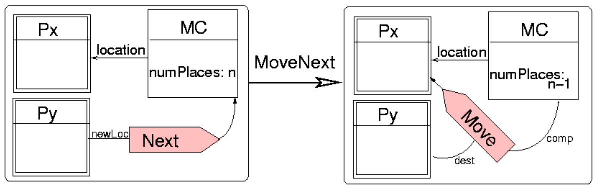

Fig. 8: MoveNext rule

The result of the *Move* request is either a *Continue* message, as appears in Figure 5, left side, or a *NoGo* message. The *NoGo* message means that the MC can not be hosted in the destination place for some reason like, for instance, some of the necessary resources are not available. The rules that describe the movement were not shown here since we have only focused on the

aspects that the specifier has to deal with. These rules define the handshake performed between the origin, the destination place and the MC in order to move MC - see (Dotti and Ribeiro, 2000) for further details. From the point of view of the specifier a mobile component will issue a *Move* request at the left side of one or various of its rules. The result of the movement is returned to the mobile component through either the *Continue* or *NoGo* messages. These messages must be present in the left-side of rule(s) of the mobile component in order to continue the computation.
As described in section 6.1, a complete specification of an entity includes the specification of its initial graph. The initial state graph of entity MC is given by the *MC-InitGraph* in figure 9 (the elements in gray denote instantiation variables).

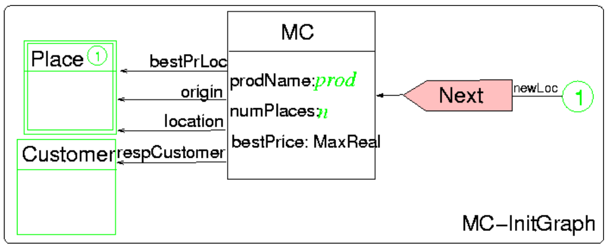

Fig. 9: MC InitGraph

According to Figure 9, when an instance of an MC is created, values for the attributes *bestPrLoc*, *origin*, *location*, *respCustomer*, *prodName* and *numPlaces* are provided, and the message *Next* is sent to MC to start its computation.
The initial state of the whole system specifies all entities that must exist in advance and the relationships among them. The initial state graph of the system is achieved by including the appropriate initial state graph of the entity for each instance needed in the beginning of the system. In this process, instantiation variables receive the appropriate values for the initial state of the system.
Figure 10 depicts the initial state of a system comprised by four places (one original place, *P_Orig*, and three additional places, *P_1*, *P_2* and *P_3*), where the agent will search for the best price of a product, an MC and a Customer. Each of the places has its own *Information Service (IS_Orig*, *IS_1*, *IS_2* and *IS_3*, respectively).

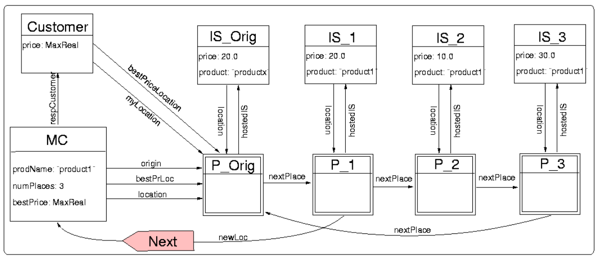

Fig. 10: System Initial Graph

Figure 10 shows also a message *Next* being sent to MC which will trigger the computation in the system. As its next step, MC will apply the rule *MoveNext*, shown in Figure 8 (since the left side of this rule is a subgraph of the graph in Figure 10). The computation then continues according to the specified rules.

3.3. SIMULATION TOOL

The main advantage of using simulation is the possibility of testing the behavior of a system and finding errors in its conception before doing the implementation. In our environment we use an extension of the simulator developed in the PLATUS project (Copstein, Móra and Ribeiro, 2000). This simulator is written in the Java language (Gosling and McGilton, 1996) and simulates models described in OBGG. It works with entities that communicate with each other by exchanging messages.
A special module called *kernel* is responsible for message delivery and for the global time control. The kernel algorithm follows a conservative approach and the simulation time has a centralized control. Conservative approach means that the simulation time is advanced to the next one only when all entities finished processing the messages relative to the current time.
Considering the functions of the kernel, OBGG specifications are mapped to the simulation environment in a straightforward way:

- OBGG entities, with their attributes, map one-to-one to simulation entities, with corresponding attributes;
- each rule specifying part of the behavior of an OBGG entity is mapped to a class (at the implementation language level) representing it. A a class that describes a rule has attributes and methods to:
  - i) test if the left-side of the rule is true for the associated entity;
  - ii) test if the condition (issued on the arrow from left to right side graphs) holds for the associated entity; and
  - iii) apply the rule, whereby: attributes of the associated entity may be changed; new entity instances may be created; messages to other entities may be created.
- messages handled by OBGG entities~~,~~ with the respective attributes, map to simulation messages with their corresponding attributes;
- the initial state of the system is mapped accordingly to a initial state of the simulation environment through the creation of the appropriate entity instances, messages and attributes initialization.

An entity is basically modeled by an active object (a Java object with an internal thread). A receive buffer of this object is used to store the messages delivered by the kernel, which correspond to the messages at the specification level. The internal thread of the entity selects the rules enabled by messages in the input buffer and trigger their execution, respecting the non-deterministic behavior given by graph grammars.

The internal thread of the entity follows the algorithm of Figure 11. The general idea of the algorithm is as follows: for each message in the input buffer (line 6) all candidate rules to handle that message are collected (line 9). For each of these rules it is checked whether it conflicts with other rules already selected to be applied (lines 10 to 12). If there is no conflict then the rule is selected to be applied, handling message m (line 13). If no rules are selected for a message, the message is postponed to be a candidate for execution in the next simulation time (line 15 to 19). After checking all messages from the input buffer, the set exec has all non conflicting rules that may then be applied concurrently. Separate threads are started to apply each rule in exec (line 21). After the end of these threads, a new state of the entity has been constructed (line 22) and new rules may be selected for application considering the new state (returning to line 2).

The non-deterministic behavior is modeled through a randomic choice of rules among those that are enabled, in line 11: choose rule r of R.

The function conflict(rule, exec) encodes the decision if a rule may or may not be applied concurrently with the rules in exec, which is a set of rules chosen for execution with the respective messages they handle. Ideally, only rules that write on the same attributes are not allowed to proceed. Currently we have implemented a version that performs such control on an entity (or object) based and not based on the attributes of the entity, i.e. we have a notion of blocking at the object level.

```
1  While the end of simulation was not notified by the kernel
2      construct the set M of all messages posted by the kernel
3      // Choose rules to treat each message,
4      // building the set exec={(r1,m1),(r2,m2),...}
5      error := false;  exec:={};
6      while (M!=∅  and  (not error))
7          choose a message m in M; M:=M-m;
8          chosen := false;
9          construct set R with all enabled rules that handle m;
10         while ((R!= ∅)and(not chosen))
11             choose rule r of R;
12             if not conflict(r, exec)
13                 then exec:=exec U (r,m); choosen := true;
14             R := R-r;
15         If (not choosen)
16             then if (tmax(m)>T) then
17                 send the message back to the kernel with
18                   timestamp (T+tunit);
19                 else error := true
20     if error abort the system;
21     launch processes to execute all rules in exec;
22     construct the next state with the results of all processes;
```

Fig. 11: Internal Algorithm of an Entity

The simulator allows one to represent minimum and maximum timestamps for a message, meaning that the message has to be treated at the destination entity between current time + minimum and current time + maximum. The minimum time allows one to represent delays of real channels, if the case. The maximum time is supported for situations where one wants to specify systems with temporal restrictions, i.e. if a message is not consumed in a certain time interval, the system fails. Since we are looking at an asynchronous scenario, we do not use maximum times. Minimum times may still be used to model minimum delays. Lines 15 to 19 means that if a message m has not been chosen to be processed by some rule, then it will be postponed to be considered for the next simulation time (lines 16-18). If a message has a maximum time which is less than the current simulation time, then the simulation is stopped raising an error, since it was not possible to keep the time restrictions stated in the model.

Graphic tools are under development to allow the graphical creation of simulation models and their automatic conversion to the corresponding simulation code.

In order to support the simulation of mobile applications specified using OOBG, the simulator was extended by creating special entities that represent the behavior of places and mobile components. We have used the simulator to test the behavior of various sample applications. The tool proved to be valuable since relevant specifications errors could be found during the specification phase - e.g., messages sent to entities that did not handle them; live-lock or dead-lock situations caused by wrong definitions of rule conditions; synchronization errors, among others.

3.4. CODE GENERATION

To accomplish the objective of translating a formal specification into executable code for mobile applications, we have further extended the translation process usedin the PLATUS simulator, discussed in the previous section. In a simulated situation, the time, and therefore all events in the system, are ruled by the simulation algorithm. In this process, the kernel plays the important role of passing events (messages) between entities, and of giving a temporal coherence to these events. The goal is to resemble the time of a real situation. In a real situation the time is implicit, i.e. the events of the system are naturally ordered in time as they occur. Messages can be passed directly from entity to entity as they are generated.

So, our first step to generate a real application out of an OBGG specification is to take the same mapping of OBGG entities to Java classes established for simulation, but excluding the simulation time control aspects and the message passing functionality of the simulation kernel, having entities communicating directly. To do this for a distributed scenario, we decided to adopt an underlying platform that ensures distribution transparency and FIFO (first-in-first-out) semantics for message delivery, therefore keeping the kernel semantics for message passing. The chosen platform was Voyager (ObjectSpace, 2000). This platform provides the necessary support for distributed communication with location transparency and component mobility with reference resolution. Although we have used this specific platform, we have conceived the modifications in the code to easily allow the implementation to be ported to other support platforms with analogous functionality.

The platform plays the same role of message passing provided by the simulation kernel. Concerning time aspects, the simulation time can not be considered anymore at the entity algorithm of figure 11. This algorithm is therefore revisited.

The main modification is that lines 17 and 18 do not handle message timestamps since they do not further exist, the message is simply let in the input buffer of the entity to be further treated.

Also, the handling of error situations does not exist anymore (affecting lines 5, 6, and 16 to 20) since we are dealing with asynchronous systems that have no maximum time for their messages to arrive.

The next important aspect to be introduced is the implementation of real entity migration. This can be decomposed in the following sub-aspects:

1) stop the entity in the original place;
2.a) save the internal state as well as
2.b) execution state of the entity;
3) transfer it to the destination;
4) resolve references, i.e. make the migrating entity continue to be addressed from other entities;
5) resume the activities of the entity at the destination.

Voyager is able to migrate passive Java objects (without internal activities) between distinct locations. The platform supports steps 2.a (it saves only the internal state of the object that represents the entity), 3 and 4. In order to handle active objects (objects with internal activities - one or more threads), which is the case for entities, we have to build the support for steps 1, 2.b and 5. Step 1 guarantees that, when the migration process begins, no internal activity is being performed and the entity is in a consistent state. After migration (using the support platform), step 5 resumes the entities execution, which starts to process the messages available in the input buffer. Since the platform assures that messages are delivered, without losses, even if the destination entity is moving, the behavior of the entity remains the same. The behavior described above was implemented in the specialized entities Place and Mobile Component. The mobile components and places defined by the developers are mapped to the above described implementations of Place and Mobile Component.

4. CONCLUSIONS AND FUTURE WORK

We have developed some case studies with our framework, including: a simple market application; the specification of an active network architecture (Duarte, 2001); the dynamic source routing protocol for active networks (Duarte, 2001); as well as other test situations involving concurrency of mobile components on places and other important features. The development of these case studies included the specification of the application in OBGG, the translation of the specification into simulation code, the simulation of the application to test its behavior, the translation of the specification into executable code and its execution. The results obtained in the execution of the generated code are coherent with the simulation results and with the formal specification of the applications. Although this is a positive indication about the correctness of these tools and methods, a formal proof that both simulation and real execution of entities keep the semantics of OBGGs is still missing. We are working on this topic since it is necessary in order to give formal guarantees that the analyses (verifications) performed on the specification hold for the simulation as well as for the real execution.

Our approach to the generation of an executable application is through a direct mapping of its OBGG specification, without further complementing the specification during this mapping. At the one side, this is a positive aspect because we eliminate the possibility of introducing errors in the application during the mapping process. At the other side, all aspects of the application must be resolved at the specification phase. Some aspects of an application, however, may be non natural to be represented with the adopted formalism. For instance, the representation of sequences of activities needs an additional control by the developer. Although this is not a prohibitive aspect, we will investigate it as we gain experience with various application scenarios. A possible alternative approach would be to specify with OBGG to some abstraction level, which can be simulated since the simulation model may abstract from various details. At this abstraction level we would have the various entities with their interface behavior defined, however some might not have the internal behavior specified completely. In an additional step a complementary approach would be needed to describe the internal behavior of the entities and then show the equivalence of the internal behavior with the interface behavior already specified in OBGG. The complementary approach adopted could then be engineered to ease its mapping to current programming languages and run-time environments.

Among the additional aspects being investigated in the context of the framework are also the introduction of inheritance in the formalism (what would allow better reuse and specialization of entities) and the modeling of failures. The need for inheritance, or even a way of parameterizing defined modules, is clearly needed. Since OBGGs as defined are strongly typed, the simple reuse of an entity in a scenario where it would have to exchange the same messages, but with other entities, leads to modifications in the entities definition. This is because often we have as message attributes the originator or destination entities. As an example consider that in our example of section 3.2 MC does not answer back to Customer the best price it found, and where, but it answers to another mobile component, say MC-X, which is responsible for gathering the results of various searches it launched. Then we would need to modify the definition of MC to send the answer message to MC-X and instead of having attribute *respCustomer* of type Customer it would have to be defined as of type MC-X. This is too strong a restriction if we consider the specification of applications to work in open environments. Introducing inheritance would help to deal with this problem.

As described in this paper, the place and mobile component abstractions define the expected communication and migration characteristics. The definitions of these abstractions reflect the expected environment for mobile applications. We could modify such abstractions to represent other types of environments. One possibility is to represent the existence of some kinds of failures. Although this is an issue we are working in, we believe it is possible to represented faulty environments and therefore allow specifiers to reason about specifications of fault tolerance mechanisms.

REFERENCES

Fuggetta, A., Picco, G. P., Vigna, G, 1998, “Understanding Code Mobility”, *IEEE Transactions on Software Engineering*, Vol. 24, pp. 342-361.

Milner, R., 1999, “Communicating and Mobile Systems: the Pi-calculus”, Cambridge University Press.

Cardelli, L., Gordon, A., 1998, “Mobile Ambients - Foundations of Software Science and Computational Structures”, *Lecture Notes in Computer Science*, Vol.1378, Springer-Verlag, pp. 140-155.

Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., Rémy, D. A., 1996, “Calculus of Mobile Agents”, *Proceedings of CONCUR'96, Lecture Notes in Computer Science*, Vol. 1119, Springer-Verlag.

Sewell, P., Wojciechowski, P.T., Pierce, B.C., 1999, “Location-Independent Communication for Mobile Agents: a Two-Level Architecture”, Technical Report 462, Computer Laboratory University of Cambridge.

Gosling, J., McGilton, H., 1996, “The Java Language Environment - A White Paper”, Sun Microsystems.

Pierce, B., Turner, D., 1997, “Pict: a programming language based on the pi-calculus”, Technical Report 476, Indiana University.

Wojciechowski, P., Sewell, P., 1999, “Nomadic Pict: language and infrastructure design for mobile agents”, *Proceedings of the ASA/MA'99*.

Dotti, F. L., Ribeiro, L., 2000, “Specification of Mobile Code Systems Using Graph Grammars”. *Formal Methods for Open Object-Based Distributed Systems IV*, Kluwer Academic Publishers, Stanford, USA, pp. 45-63.

Copstein, B., Móra, M. C., Ribeiro, L., 2000, “An Environment for Formal Modeling and Simulation of Control Systems”. *Proceedings of the 33rd Annual Simulation Symposium,* SCS, pp. 74-82.

ObjectSpace, Inc., 2000, “Voyager ORB 4.0 Developer Guide”, Objectpace, Inc.

Victor, B., 1994, “The Mobility Workbench User's Guide: Polyadic Version 3.122”, Available at http://www.docs.uu.se/~victor/mwb.shtml

Duarte, L.M., 2001, "Desenvolvimento de Sistemas Distribuídos com Código Móvel a partir de Especificação Formal (Developing Distributed Systems with Code Mobility from Formal Specifications)". M.Sc. Thesis, in Portuguese, PUCRS - PPGCC.