Simulation of Mobile Applications in Open Environments

Eduardo Tavares Rödel[a], Lucio Mauro Duarte[b], Osmar Marchi dos Santos[c], Fernando Luís Dotti

PUCRS – Faculdade de Informática – PPGCC
Av. Ipiranga, 6681 – Prédio 16 – ZIP 90619-900
Porto Alegre – RS – Brazil
[eduardot, lduarte, osantos, fldotti]@inf.pucrs.br

Abstract. In this paper we exploit various aspects of simulation for mobile applications. We briefly describe a formal language and a tool that we use for the simulation of mobile applications, showing how to analyze their performance. Also, we show how we use our language and tool to represent failure behavior that may be present in open systems. This allows one to reason, through simulation, about the robustness of the application and fault detection or tolerance mechanisms. A case study on active networks is presented for performance analysis and to exemplify failure representation.

Keywords. Code mobility, open systems, simulation, formal methods, failure representation.

1. Introduction

The development of mobile applications is a complex task. Such applications are inherently concurrent and distributed, having also components that may migrate during the execution. In particular, guaranteeing the correctness of such applications is far from trivial if we consider the characteristics of open systems, like: massive geographical distribution; high dynamics (appearance of new nodes and services); no global control; partial failures; lack of security; and high heterogeneity [1]. Among other barriers, in such environments (e.g. Internet) it is hard to test mobile applications because we cannot be sure whether an error is caused by the application itself or by the environment in which it runs.

It is therefore necessary that developers of mobile applications have a higher degree of confidence in their solutions during the construction phase. To achieve that, we have developed methods and tools to assist the development phase such that the developer may assure that desired properties of the application are present[*]. In this context, the use of formal methods becomes a good approach to provide a way to create a precise description of the applications. More specifically, we have developed [2] and used [3] a formal specification language suited for mobile applications. Models written according to this formal language can be simulated with a simulation tool [4] as well as code for them can be generated following a straightforward mapping to Java classes considering the use of a mobility support platform [3, 5]. Altogether we have a framework for the development of correct mobile code applications [6]. The innovative aspect of the framework is the use of *Object-Based Graph Grammars (OBGG)* [2] as the underlying unifying formal method for a set of integrated tools [6]. Each of the tools supports a way of addressing the generation of correct mobile applications. While in [4] we have focused on the simulation tool itself, in this paper we will concentrate on simulation of mobile applications for open environments and the various uses of our simulation tool.

Our achievements, so far, can be characterized by the mapping of an OBGG specification into a simulation model and the generation of code for a mobility support platform. Under these assumptions we could not find in the literature projects that address the same points that our environment does address. According to our review, we have found projects that use formal methods to develop applications based on code mobility, some of them providing analysis tools.

The KLAIM [7] (Kernel Language for Agent Interaction and Mobility) project aims at the creation of a language supporting the paradigm of code mobility programming. KLAIM explicitly use localities for accessing data or computational resources, and has a type system to control access rights. In the KLAIM structure there is a net coordinator that has control over the net (there is a special coordination language) and processes. One special feature of KLAIM that resembles our project is the possibility to generate code through the Java implementation of KLAIM, named KLAVA.

There are also projects found in the literature that use formal verification techniques. Although planned, our environment does not address this point, so far. The

[a] This author is partially sponsored by CAPES/Brazil.
[b] This author is partially sponsored by FAPERGS/Brazil.
[c] This author is partially sponsored by CNPq/Brazil.
[*] These activities take place in the scope of projects ForMOS - Formal Methods for Mobile Code in Open Systems - Grants: CNPq-AI, CNPq/ProTeM, FAPERGS (Brazil) and IQ-Mobile - Improving the Quality of Open Systems with Code Mobility through Rigorous Development - Grants: CNPq/CNR (Brazil/Italy).

main projects under this approach are MobiS [8], Mobile UNITY [9] and Mobility WorkBench [10].

MobiS is a specification language whose specifications denote a tree of nested spaces that dynamically evolves in time [8]. There is the possibility of automatic verification of specified properties. Mobile UNITY, unlike MobiS, is a model to reason about key concepts and ideas of mobility. The Mobility WorkBench is a tool, not a specification language. It's designed to analyze concurrent systems over specifications written in the pi-calculus formalism.

One major advantage of using a simulation model is the possibility of validating strategies as well as control algorithms for complex cases where the use of formal verification could lead to a state explosion problem. These models may be used to check if the components of an application behave as expected, if they are independent from each other such that the replacement of a simulated component by a more sophisticated version becomes possible, and also to check the application behavior under various environment conditions. Of special concern for open systems, we have a tool and a method that allow us to interactively reason through simulation about mobile applications in presence of selected failures, helping the developer to formally specify robust applications as well as mechanisms for fault detection and tolerance. Moreover, it is also highly desirable to evaluate the performance of solutions before their implementation. In fact, in many cases this evaluation can even change the use of mobility in the solution under preparation, leading to significant improvements in the application.

The main uses of our simulation environment are discussed in this paper. In order to demonstrate these uses, we have developed, as a case study of significant complexity, an architecture for active networks [11] and a routing algorithm [12] to execute over it.

This paper is organized as follows: Section 2 discusses about our simulation tool; Section 3 discusses a case study used to demonstrate the uses of the simulator; Section 4 presents some uses of the simulator; and Section 5 brings us to the conclusions and future work.

2. The Simulation Environment

The use of a *formal specification language* allows the creation of a precise description of a system with well-defined syntax and semantics. The formal specification language used in this work is based on a restricted form of graph grammars, called *Object-Based Graph Grammars (OBGG)*[2].

OBGG incorporates object-based concepts, such as encapsulation and communication through message-passing. An OBGG specification is composed by type graphs and an initial graph. A type graph describes an entity, the names of its attributes, and the types of messages that the entity may send or receive. An application specification is composed by the specification of the various entities involved. The behavior of an entity is determined by the rules associated to it. A rule can be applied whenever the left-side of the rule is a sub-graph of the current system state graph – i.e., rules may be applied in parallel - and there is no conflict. A conflict exists when two or more rules need write access to the same attributes. In case of conflict, the choice of which rule will be applied is non-deterministic. The application of a rule must consume a message and may change the internal state of the entity and/or generate new messages. The application of rules successively changes the state of the system, starting from an initial state, described in the initial graph.

To represent mobile systems, OBGG was extended with two specialized entities: *places* and *mobile components*. *Places* (represented by the *Place* entity in our simulation environment) represent the possible locations where mobile components may execute. A place offers basic functionalities like storage and communication facilities and computational power. Furthermore, places offer message passing and migration services to mobile components. *Mobile components* (represented by the *MAgent* entity in our simulation environment) represent software components that can migrate between places during their execution, using resources and services from the places they visit. When creating a mobile application, the user may specialize the entities place and mobile component, as he/she wants.

In order to simulate specifications written in OBGG, we have conceived a simulation environment and described the mapping of specifications in OBGG to a simulation model. The simulation environment, presented in [13], is implemented in Java [14] and is composed by: (i) a library that supports the basic OBGG abstractions of entity, attribute, message, and rule; and by (ii) the kernel of the simulator, which is responsible for message passing and global time control. A simulation entity is basically modeled by an active object (a Java object with an internal thread) where: attributes of the entity are mapped to attributes of the object; a message buffer of this object is used to store the messages delivered by the kernel to the entity; and rules are mapped to associated classes with functionality to: (i) test if a rule is enabled for an entity, and (ii) apply the transformations stated by the rule on the entity. The internal thread of the object selects the rules enabled by messages in the input buffer and the internal state of the entity and triggers their application according to the OBGG semantics. With the constructions offered by the simulation environment, the mapping of specifications in OBGG to simulation models is straightforward. For more details on the structure of the simulation tool please refer to [13].

Based on the mapping created to translate OBGG specifications into simulation models, we also created a mapping to generate code for a mobility support platform. This way, it is possible to specify an application, simulate it and generate code to execute the application in a real environment [3].

3. A Case Study

In this section we describe the case study we used as example throughout the paper. We developed an active network architecture with mobile components. Active networks are very flexible in terms of configuration and demand an intense use of code mobility. Because of its inherent complexity, performance analysis of such cases becomes almost not feasible using other methods, such as analytical ones, due to the state space explosion. Therefore, this is an interesting case for performance evaluation through simulation. Moreover, since active networks are an emerging area, we believe that there is a lack of simulation and analysis tools, and that our work could contribute in this aspect too.

Active networks [11] are named *active* because the routers, in such network, can perform computations according to user code carried by packets received by them. This way, the user has the possibility of "programming" the network, providing the programs to be used by the routers to execute their computations. Programs can be dynamically inserted in the network nodes in order to configure them according to requirements of applications.

The architecture considered is composed by:

- *Active nodes*: are the nodes of the network and can send and receive capsules in unicast and broadcast modes;
- *Capsules*: represent the packets transferred over the network. They can carry data and code. Each type of capsule is handled by a service;
- *Service*: are entities that own the code to be executed with the data of a specific type of capsule, handling it. Services execute in active nodes and serve specific types of capsules;
- *Code bases*: are entities that maintain the available services. They provide instances of services to active nodes.

Each entity of the described scenario was mapped to an OBGG entity. Active nodes were mapped to places (static) whereas capsules, services and code bases were mapped to mobile components. According to the classification presented in [11], this architecture follows the active nodes approach, where the capsules carry only the identification of the service they require to process their data.

On top of this architecture we developed the *Dynamic Source Routing* (*DSR*) [12]. The DSR algorithm was developed to route packets between nodes of an *ad hoc* network, whereby the path through the network that a packet must follow from the origin to the destination node is determined before sending the packet. The algorithm is composed by 2 mechanisms: the route discovery and the route maintenance. *Route discovery* is the mechanism used by a node to dynamically discover a route to a destination node. Discovered routes are stored in a *route cache (RC)*. *Route maintenance* is the mechanism used to detect changes in the network topology, i.e. some known routes become invalid or new routes are available.

For exemplification and explanation purposes, we present an excerpt of the specification of the DSR algorithm, showing some rules. Due to space limitations, we do not present the complete specification (for more information please refer to [3]).

When the application has started, data capsules need to be routed from one node to another using the DSR sevice. The rule of Fig. 1 defines that when the DSR service receives a message *Packet* from a data capsule (*Packet2*), it checks if it is the destination of the capsule and it looks for a route to the destination node in its cache. If it is not the destination (*hid <> d*), it does not know a route to the destination node (*not rc.isInList(d)*) and this capsule has not been already routed (*rt.isEmpty()*, where *rt* represents the information of a route to follow), then this rule is applied. The rule application causes the generation of a route request capsule (the dashed entities represent entities that are created by the rule application), that is initialized with the identification of the node that is requiring a route, the identification of the destination node to be found, the information of the service that will handle this capsule (DSR service) and other control information. The route request carries the information of which nodes it passed through (attribute *route*). So, the identification of each node visited by the request capsule is put in the route information. The node that generates the request, includes itself in the route (*rt.add(hid)*), meaning that it is the first node in the route.

The route request is sent in broadcast by the node. This way, neighbor nodes in the network receive the request and verify if they have a route to the destination node in their caches. If a node knows a previous route to the destination, it sends a reply to the request source node informing the route. If no route is known, it propagates the request in broadcast to all its neighbors. When the request reaches its destination, the rule *TargetFound* is applied. This rule is presented in Fig. 2.

Once the destination node receives the request, it generates a route reply capsule, which receives the route from the origin of the request to the local node and is sent back to the origin through the inverse route followed by the request.

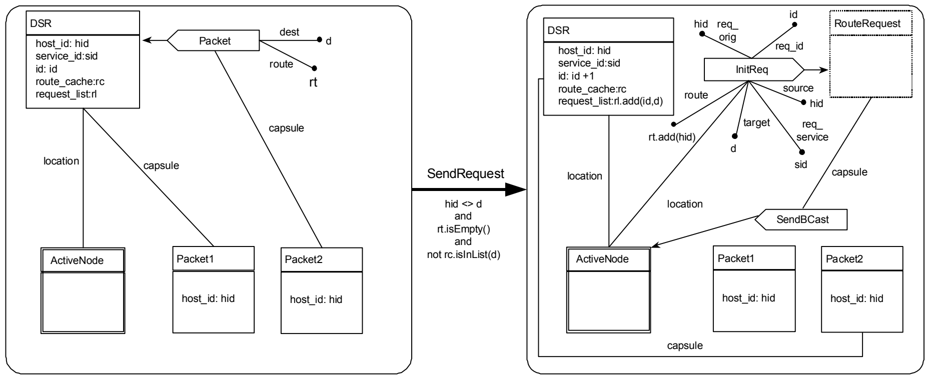

Fig. 1. Rule *SendRequest*

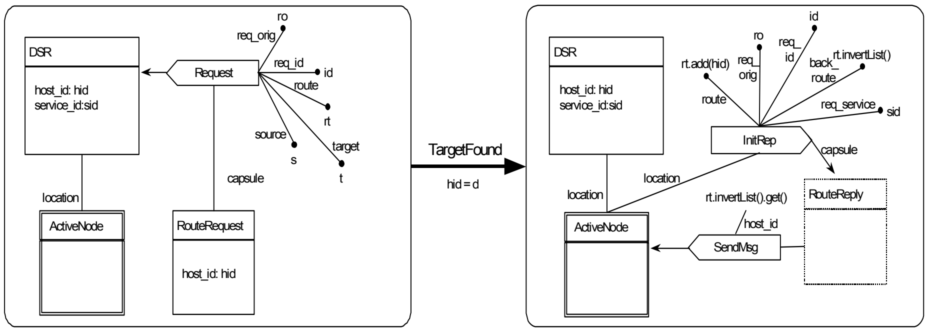

Fig. 2. Rule *TargetFound*

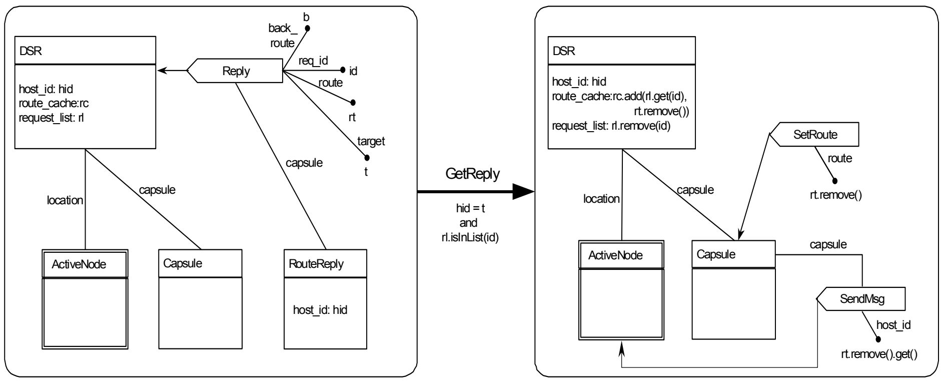

Fig. 3. Rule *GetReply*

The route reply is forwarded by the intermediate nodes until it arrives at the request origin node. When the origin node receives the route to the destination, it sends the data capsule through the route obtained. This route is stored in the local cache, so next capsules with the same destination can be routed without executing the route discovery process. The reception of route reply enables the application of rule *GetReply* (Fig. 3). The capsule is then forwarded through the nodes contained in its route information.

When it arrives at the destination node, the DSR algorithm for this capsule is over.

4. Applying Simulation to our Case Study

Simulation models can be used to achieve an early representation of mobile applications, allowing the developer to reason about their behavior even before their implementation. The ability to resemble various scenarios and conditions involving the components of the application makes it possible to identify errors that would be hardly detected when testing the real system execution, especially if one consider open systems with mobile components. Our simulation environment provides these benefits, allowing one to consider also specific aspects such as locality, mobility and failures, which are of importance for mobile applications in open systems.

While developing our case study (see section 3) relevant specifications errors could be found during the specification phase, like for instance: sending a message to an entity that is not able to handle it; wrong definition of conditions of a rule; dead-lock and live-lock situations, among others.

In the following sections, we discuss performance measurement of mobile applications through simulation, as well as representation of selected failures in the simulation model in order to reason about the system behavior in the presence of failures.

4.1. Performance Evaluation

Simulation can also be used to analyze the expected performance of an application. Although it is possible to develop analytical models for mobile applications to obtain performance indexes, we can analytically represent and analyze only simple applications, with low degrees of concurrency, and abstracting from various aspects that may be of interest. When applications are more complex and we want to analyze their behavior in detail, then it is not trivial to formulate these applications analytically. Having the ability to simulate our specifications makes it possible to investigate more complex situations, in a degree of abstraction that may satisfy our expectations, and turning the analysis process much more comfortable to the developer.

According to the abstractions of our formal specification language, components can be mobile or not, and communicate through asynchronous message passing. Message passing may be local (mobile components in the same place, mobile component communicating with its current place) or remote. Assigning appropriate delays to local and remote messages delivery, as well as to the migration of a component, allows the developer to represent the latency of network links, the amount of time necessary to transfer messages and mobile components and other time consuming operations. This makes it possible to compare the performance of different strategies of distribution and mobility in different situations of the environment. For instance, different delays according to network conditions (congestion, low traffic, light-loaded, etc.).

As discussed in Section 2, the OBGG simulator has a special entity called *kernel* that controls the simulation time and is responsible for passing the messages between entities. Since we have the delays of local and remote communication and migration informed to and handled by the *kernel*, we can use the *kernel* to obtain various performance indicators. Some of them are generic and can be obtained for all mobile applications through the simulator support, without introducing any control in the application. The generic information provided by the *kernel* is:

- the total time consumed in local communications;
- the total time consumed in remote communications;
- the total number of local messages generated;
- the total number of remote messages generated;
- the average time spent in local communications;
- the average time spent in remote communications;
- the total number of migrations occurred;
- the total time spent in migrations;
- the average time spent in migrations;
- the application execution total time.

Communications that occur at the same simulation time (in parallel) are accounted once (the one that lasts longer) for the application. The generic performance information is calculated based on the values given by the developer to the delays associated to the messages.

The delay for delivering local messages and remote messages and the delay for migrating a mobile component are defined by the developer in the initialization of the *kernel*, that is responsible for adding the correct delay to the timestamp of local and remote messages. We are considering approaches to represent random values for message delivering delays, in order to better represent the execution of applications in open environments. For simplicity, we assume that the message delivering delay does not vary and we do not take into account the size of a message, supposing that all messages have the same size. The migration time can be calculated based on the size of the mobile component (size of its code) and on the latency of the network that it will pass through. For the analysis in this paper we have assumed that all mobile components have the same size.

Besides the generic information, the simulator generates a log file containing all events occurred during the simulation of an application. Other specific information for an application can also be obtained. However, since this information is application dependent, the developer would need to include explicit controls in his/her application in order to obtain the expected performance values.

4.1.1. Performance Evaluation Results for the Case Study

The tests with the DSR algorithm working over the active network architecture were done using the logical topology presented in Fig. 4.

The 10 nodes involved in the topology were active nodes. Besides these 10 nodes, a node *N0* was created to host the network code base. An instance of the DSR service was created in node *N0* when the application was started. No other node initially owned this service, so when a node received a DSR capsule for the first time, it had to ask the code base for an instance of DSR service in order to handle the capsule received.

The performance measurements of the DSR algorithm over the active network with the topology of Fig.4 involved the generic information provided by the kernel as well as specific indexes gathered through the introduction of control points in our application. These specific performance indexes were calculated for each node of the network were:

1. Total number of capsules handled;
2. Average time spent to handle a capsule: average interval of time from the reception of a capsule to the time the required service is provided to the capsule;
3. Total time spent to install the DSR service: interval of time from the request of a DSR service instance to the code base until the installation of the service;
4. Utilization: Amount of application time that a node spent handling capsules;
5. Average time to route discovery: average interval of time to find a route for a capsule.

In order to obtain performance indexes for our application, we generated 3 data capsules to be routed from each one of the 10 nodes to other 3 different nodes. We assumed that a local message takes 5 units of time (u.t.), a remote message takes 10 u.t. and a migration of a component is assumed to take 30 u.t. These values were arbitrarily chosen. The simulator allows the developer to select the delays he/she wants to use in the beginning of the simulation. The results of simulation can be seen in Table 1 (generic information) and Table 2 (application specific information).

We can observe in the results provided by the simulator one of the advantages of using mobility: the number of remote communications is much smaller than the number of local communications (see Table 1). In our scenario, we had a great number of migrations, what was expected once that our application demand intense mobility.

Looking at the nodes utilization (Table 2), we can identify which nodes are more visited by the capsules. In our testing scenario, we could see that node N3 is

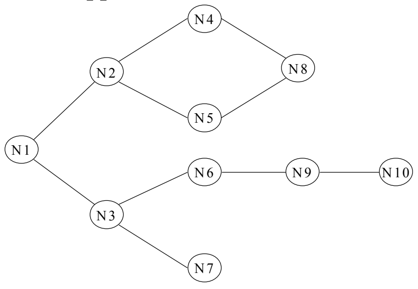

Fig. 4. Logical topology used to simulate the execution of the DSR algorithm.

Table 1. Results of generic performance evaluation of the mobile.application.

Generic performance indexes	
Total number of local communications	23447
Total time spent in local communications	14160 u.t.
Average time for local communications	5 u.t.
Total number of remote communications	3114
Total time spent in remote communications	11234 u.t.
Average time for remote communications	10 u.t.
Total number of component migrations	673
Total time spent in migrations	9515 u.t.
Average time to migrate a component	30 u.t.
Application execution total time	34909 u.t.

Table 2. Results of application specific information.

Performance indexes for nodes	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10
Total number of capsules handled	96	109	118	63	61	74	37	54	54	26
Average time to handle a capsule (u.t.)	103,9	118,9	118,98	103,33	103,28	103.58	87,57	103,06	101,11	86,54
Average time to install service (u.t.)	9975	12960	14040	6510	6300	7665	3240	5565	2730	2250
Node utilization (%)	28,57	37,13	40,22	18,65	18,05	21,96	9,28	15,94	15,64	6,45
Average time to route discovery (u.t.)	3875,67	5879,33	3971,67	6246	3131	5663,33	4331,67	6010	7180	6716,67

the node with the highest utilization. Considering a real environment, this information could be used to detect possible bottlenecks in the topology.

Another interesting information is the time spent to install a service. In our scenario we have one code base, which owns the services available in the network, at a separate node, called N0. We suppose this node to be equidistant from all network nodes. So, in this case, the distance between the node and the code base does not influence the time spent to install the service. Therefore, the time spent to install the service is basically determined by the concurrent requests to the code base. Nodes will receive the requested service faster or slower depending on the number of requests the code base has to answer. Using simulation, we could analyze the performance gain of a strategy for code base replication.

The average time of a route discovery process (Table 2) is totally dependent of the target node. So, the results are a consequence of the origin and destination nodes we chose for the capsules we generated.

4.2. Failure Simulation

In order to be able to reason about mobile applications for open systems, we have first to formalize important features of open systems and then consider our application under development in the presence of these features. One important feature of open systems is the presence of partial failures. In this section we show that it is possible to represent various classical failure models in terms of our formal specification language. Also, we show that we can introduce the behavior of selected failure models in an application specification. With this, we achieve a specification that combines the behavior of the desired application in the presence of selected failures, allowing to reason about the robustness of the application as well as detection and tolerance mechanism.

The formal definition of *failure*, according to [16], is based on the observation that a faulty behavior of a process is just another kind of a (programmable) behavior. Moreover, a failure can be seen as an unwanted state transition of a system. These unwanted state transitions can be modeled through the use of additional (virtual) logical variables, acting like guards to activate specific commands (guarded commands). In this case, a group of guarded commands represents a specific *failure model* (the manner components of a system can exhibit a faulty behavior), being activated whenever its associated guard is satisfied, by the assignment of the *true* value to it.

Formally, a failure model is a function that maps a program *A* without failures into a program *A'* with failures [16]. This mapping can be done through the insertion of guarded commands, where each of these guarded commands, when activated, adds some particular failure behavior to the original program *A*.

The failure models considered in our work are:

- Crash;
- Fail-stop;
- Send Omission;
- Receive Omission;
- General Omission;
- Byzantine.

4.2.1. Representation of Failure Models

Now we explain the failure representation methodology we have developed in our work, giving examples as means to clarify important concepts and show how to insert failures into an application. Although we consider the failure models mentioned above, due to space restrictions we will show the specifications only for the *Crash* model. The other failure models are similar and will be described textually. *Crash* means that a process fails by halting [17 apud in 16], and all other processes are not warned about the failure.

We adopted the same approach in [16] of mapping a to map a program *A* into an A' with failure behavior. Here, a graph[1] *G* representing a system without failures is translated into a graph *G'* representing a system with a selected failure model. Since our specification formalism supports implicit parallelism and is declarative, it is very suited to represent guarded commands (as introduced in [16]): the left-side of a rule corresponds to the guard of a command; applying the rule transformation (according to the right side) corresponds to executing the guarded command.

In order to control the activation and deactivation of failures in the various entities of our mobile applications, we have extended the basic *Place* and *MAgent* entities with functionality for failure representation and named these extensions *FPlace* and *FMAgent*. Fig. 5 shows the *FPlace* type graph and Fig. 6 shows the *FMAgent* type graph. Only the messages and attributes relevant to the *Crash* model are shown.

The *FPlace* and *FMAgent* type graphs show the messages needed to activate and deactivate the crash behavior on those entities. Also, the *down* attribute represents the guard for a *Crash* failure model.

Now we show how these messages and guard attributes are used in the rules describing the crash behavior. The rules are showed in Fig. 7 - 9. The *MACrash* rule (Fig. 7) specifies that, when receiving a *Crash* message with a specific time parameter *f*, the *FMAgent* updates its *down* attribute to *true* and programs an *UnCrash* message for itself for the current simulation time + *f*, where *f* represents the time interval

[1] Whenever we mention "graph" in this section, we are referring to an OBGG graph.

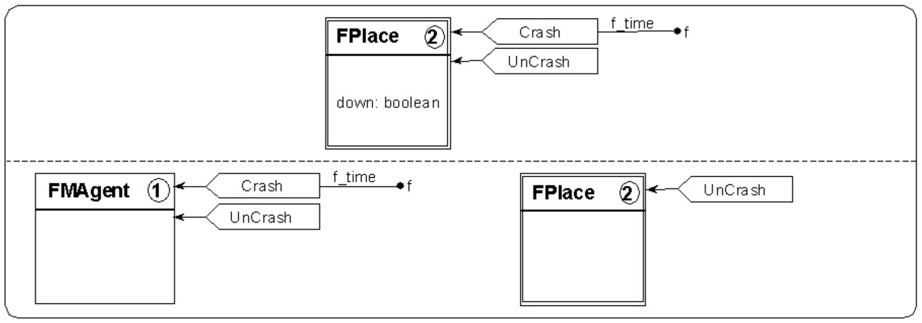

Fig. 5. *FPlace* type graph.

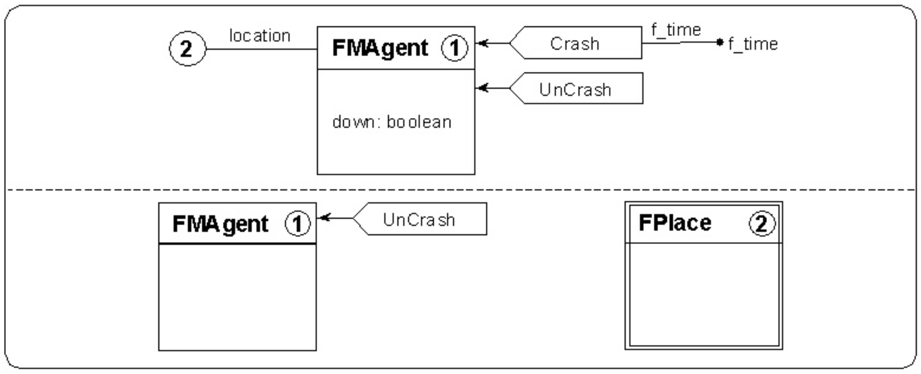

Fig. 6. *FMAgent* type graph.

the entity will be crashed. We can optionally represent a crash forever if needed. After the simulated time interval *f*, the failure behavior will be turned off, which is the goal of the *MAUncrash* rule (Fig. 9). That rule specifies that, when receiving an *UnCrash* message, the *FMAgent* deactivates its guard *(down = false)* proceeding its execution without the crash failure behavior.

While the *down* guard is true, i.e. the failure is activate, the guarded command represented by the *rule scheme MACrashed* (Fig. 8) may apply if a message arrives. A rule scheme is a rule structure that is expected for a set of concrete rules that follow that scheme. Here we are modeling crash in a way that is application independent. However, in order to specify the behavior of an entity when it receives a message *and* it is crashed, we need somehow to represent the application messages that an entity may receive, independently of what messages a specific entity may handle. In rule scheme *MACrashed*, *Message_In* represents any application message that the developer may specify for an entity, stating that when the crash is activate the behavior will be to discard the application messages.

When translating graph *G* into *G'* with failures, for each message *Mi* that an entity may receive (*Mi* specified by the developer of the mobile application), there will be a rule derived from this scheme where message *Mi* will take the place of *Message_In* in the scheme.

In order to clarify the concept of rule scheme and show how to insert failures into an application, we exemplify the creation of a concrete rule following the *MACrashed* rule scheme in Fig. 8. The rule of Fig. 10 states that if a *Reply* message is received by *DSR* and down is true, then nothing happens in *DSR* and no messages are generated. Other rules like Fig. 10 should be inserted in the model for each message that *DSR* may receive.

Up to now we considered the representation of a failure and the insertion of additional rules to the application that represent the behavior when the failure is activate. Since the semantics of our formalism (OBGG) is declarative and inherently parallel and our simulation environment supports these features, the additional rules are simply put together with the application rules. The environment is responsible to find the appropriate matches and to apply the appropriate rules.

We still have, however, to introduce an additional guard in each application rule. This is because, in case of a crash and a message received by the application, up to now, we have both the application rule (representing the application behavior on reception of that message), for example the rule in Fig. 3, and the generated rule enabled (representing the failure behavior on reception of that message), presented in

Fig. 8. In this case, in our environment, the choice of which rule is applied is non-deterministic.

Since in the crash failure model we want to have the failure behavior and not the applications behavior, we introduce an additional guard in each application rule, stating that it matches only if there is no crash. Fig.11 shows the result of adding a guard (*down* = *false*) to the rule of Fig.3.

The presented process of rule creation *(FailedGetReply)* and transformation (*GetReply* to

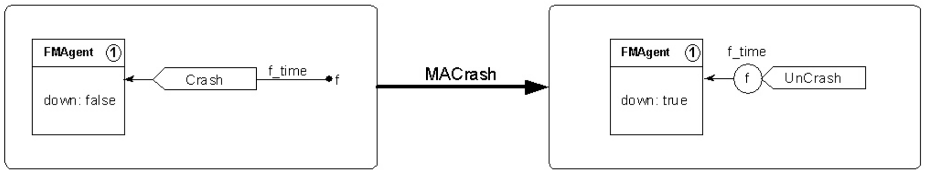

Fig. 7. *MACrash* rule.

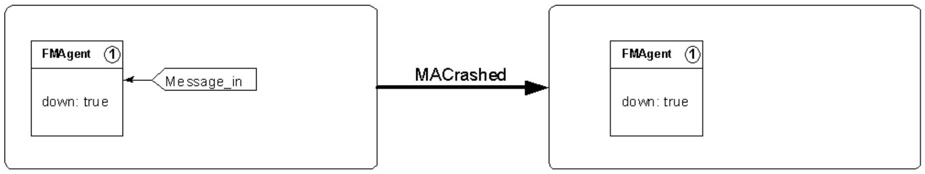

Fig. 8. *MACrashed* rule sheme.

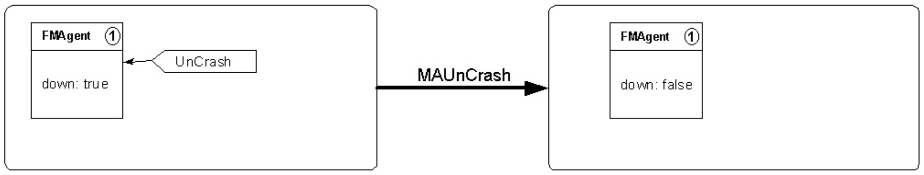

Fig. 9. *MAUncrashed* rule.

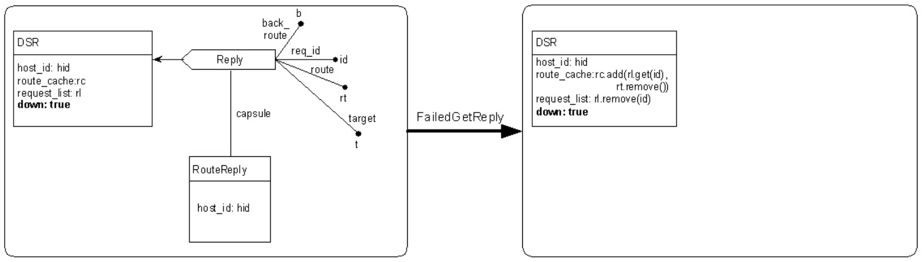

Fig. 10. *FailedGetReply* rule.

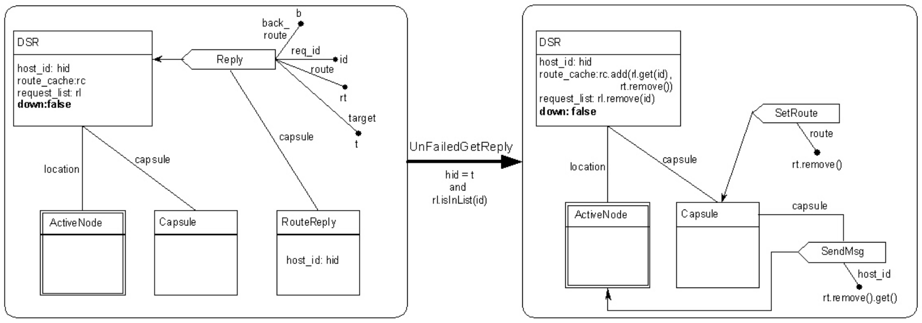

Fig. 11. *UnFailedGetReply* rule.

UnFailedGetReply) exemplifies the failure insertion process for a *FMAgent or any derived entity, like DSR*. Rule creation should take place for each incoming message, while rule transformation should take place for each original rule of the application. For inserting failures in a *Fplace or any entity that extends it*, the same process is applied. However, there are some intrinsic differences among the way a *FMAgent* and a *FPlace* drive the failure process. For a failed *FMAgent*, representing the crash failure behavior is just as described before.

Since a *FPlace* may host *FMAgents*, then it has to propagate the failure to all of them before it turn itself failed. This is done by sending a *Crash* message to each *FMAgent* hosted. After sending the last *Crash* message, *FPlace* activates its guard and starts the failure behavior.

Deactivating a *FPlace* failure behavior is done analogously. *FPlace* will propagate *UnCrash* messages to all the *FMAgents* it hosts and then deactivate its own guard. In our work, we represented other failure models beyond the *Crash* model. These failure models are *Fail-stop, Send Omission, Receive Omission, General Omission* and *Byzantine*. Generally speaking, the representation of these other models follows the same scheme as for the presented *Crash* model. That means that for each model, we used a distinct guard and guarded command(s) to represent the faulty behavior. Note that, because *General Omission* is a combination of the *Send* and *Receive Omission* models, it can be simply performed through the simultaneous activation of the two last models, dispensing an own guard and guarded command(s). The same is true for the Byzantine model, which is the combination of all other failure models mentioned [16].

5. Final Remarks

We presented some uses of a simulator that offers a mapping from OBGG specifications to simulation models. The formal proof of this translation is a current work. We developed several mobile applications and the simulated behavior was always coherent to the formal specification, indicating that the simulation code is correct. The simulator here discussed has demonstrated to be very suitable not only to be used as tool to test specifications in OBGG, but also to provide performance information. We could also note that we could introduce selected failure behavior into an application in a straightforward manner. Currently, we are making experiences on the use of the simulation to analyze faulty applications.

Through the analysis of the case study developed, we could identify interesting aspects of the application (such as possible bottlenecks) and we could also take conclusions such as the presented in the previous section.

We are currently working towards an integrated environment supporting the graphical specification, simulation and code generation for OBGG. Also, there is current work in the translation of OBGG to pi-calculus. This could be a starting point to provide verification of OBGG specifications, since we could use verification environments for pi-calculus. This topic needs further investigation.

6. References

[1] ASSIS SILVA, F.M. A Transaction Model based on Mobile Agents. PhD Thesis. Technical University Berlin. FB-Informatik, 1999.

[2] DOTTI, F. L., RIBEIRO, L. Specification of Mobile Code Systems Using Graph Grammars. Formal Methods for Open Object-Based Distributed Systems IV, Kluwer Academic Publishers, Stanford, USA, 2000. p. 45-63.

[3] DUARTE, L. M. *Desenvolvimento de sistemas distribuídos com código móvel a partir de especificação formal*. Dissertação de mestrado, Programa de Pós-Graduação em Ciência da Computação – Mestrado, PUCRS, 142 f., 2001.

[4] DOTTI, F. L., DUARTE, L. M., COPSTEIN, B., RIBEIRO, L. Simulation of Mobile Applications. In *Communication Networks And Distributed Systems Modeling And Simulation Conference 2002*, Part of the 2002 SCS Western Multiconference on Computer Simulation, San Antonio, Texas, 2002.

[5] DUARTE, L. M., DOTTI, F. L. Desenvolvimento de Aplicações Móveis. In *III Workshop de Comunicação Sem Fio 2001*, Anais do WCSF 2001, Recife, Brasil, 2001, p. 10-17.

[6] DUARTE, Lucio Mauro; DOTTI, Fernando Luis; SILVA, Flávio M Assis; ANDRADE, Aline M Santos. A Framework for Supporting the Development of Correct Mobile Applications based on Graph Grammars. In: INTEGRATED DESIGN &PROCESS TECHNOLOGY, 2002, Pasadena. Proceedings of IDPT 2002. 2002.

[7] DE NICOLA, R.; FERRARI, G.; PUGLIESE, R KLAIM: a Kernel Language for Agent Interaction and Mobility. IEEE Transactions on Software Engineering, IEEE Computer Society Press. Vol. 24(5), 1998, pp. 315-330.

[8] MASCOLO, C. Mobis: A specification language for mobile systems. Proceedings of Coordination Languages and Models, Coordination'99, Lecture Notes in Computer Science 1594, pp. 37-52, Springer-Verlag, 1999.

[9] ROMAN, G-C. & MCCANN, P.J. An Introduction to Mobile UNITY. In proceedings of the International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA 98), in Parallel and Distributed Processing. LNCS 1388, pp. 871-880, April 1998.

[10] VICTOR, B. & MOLLER, F. The Mobility Workbench a tool for the pi-calculus. In D. Dill, editor, Proceedings of CAV'94, vol. 818 of Lecture Notes in Computer Science, pages 428-440. Springer Verlag, 1994.

[11] PSOUNIS, K. Active Networks: Applications, Security, Safety and Architectures. *IEEE Communications Surveys*, 1999.

[12] JOHSON, D. B., MALTZ, D. A., HU, Y., et al. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (Internet Draft), *MANET Working Group, IETF*, 2001, 60 p.

[13] COPSTEIN, B., MÓRA, M. C., RIBEIRO, L. An Environment for Formal Modeling and Simulation of Control Systems. In *Proceedings of* 33rd *Annual Simulation Symposium*, SCS, 2000. p.74-82.

[14] GOSLING, J., MCGILTON, H. *The Java Language Environment - A White Paper*. Sun Microsystems, 1996.

[15] DOTTI, F. L.; DUARTE, L. M.; COPSTEIN, B.; RIBEIRO, L. Simulation of Mobile Applications. In: WMC2002-CNDS2002. Western Multiconference on Computer Simulation – Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002).

[16] GÄRTNER, F. C.. *Specifications for fault-tolerance: A comedy of failures*. Technical Report TUD-BS-1998-03. Darmstadt University of Technology, Germany. 1998.

[17] HADZILACOS, V., TOUEG, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425. Department of Computer Science, CornellUniversity, Ithaca, NY. NJ, 177–186.