A Formal Framework for the Development of Concurrent Object-Based Systems ⋆

Leila Ribeiro[1], Fernando Luís Dotti[2], and Roswitha Bardohl[3]

[1] Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
leila@inf.ufrgs.br

[2] Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul Porto Alegre, Brazil
fldotti@inf.pucrs.br

[3] International Conference and Research Center for Computer Science Schloss Dagstuhl, Germany
rosi@dagstuhl.de

Abstract. In this paper we present a framework for the development of concurrent object-based systems. The framework is based on graph grammars, and includes techniques for specification, simulation, animation and verification.

1 Introduction

The development of methods and techniques to aid the construction of correct concurrent software systems has been a challenge for computer scientists for many years. In the 60's and 70's, the interest was more academic, because most of the commercial tools and applications executed on mainframes, only few scientific applications made use of concurrent software constructions. The situation changed in the mid 80's with the boom of intra- and Internet systems as well as the development of highly parallel and vectorial computer architectures: software applications for distributed and concurrent systems started having commercial importance, and therefore the need for correctness gained a new perspective. This triggered a lot of research activities in the areas of semantical models (e.g. CCS [Mil89], event structures [Win89], I/O automata [LT89] – see [WN94] for an overview), formal specification languages (e.g. LOTOS [vEVD89], Estelle [ISO89], TLA [Lam94], Petri nets [Pet62,Rei85], graph grammars [EPS73,Ehr79]), verification techniques and tools (e.g. SMV [McM93], SPIN [Hol97]). Many of such activities resulted in quite powerful models that were able to express and reason about concurrent and distributed systems, and were applied successfully to protocol specifications, that are systems with the characteristics that states are quite simple (usually have no structure), and the

⋆ This work was partially sponsored by ForMOS (FAPERGS/CNPq), PLATUS (CNPq), IQ-Mobile II (CNPq/CNR) and DACHIA (FAPERGS/IB-BMBF) Research Projects, and partially developed in collaboration with HP Brasil.

important events are changes of states triggered by signals or messages. However, in practice languages like the (semi-formal) ones of UML [BRJ98] are the ones that are used, because for commercial software applications typically the states are quite complex structures, and this feature cannot be captured easily and naturally by the existing formal techniques. The formal background of users in industry was rather poor, and they were not comfortable with the required mathematical notation to use formal specification languages to describe systems and their properties. Due to the size and complexity of distributed and concurrent software applications nowadays, the cost of finding and correcting bugs in the implementation is even bigger in this kind of systems. Thus, the software development process would greatly benefict from a good analysis of functional as well as performance requirements already in the specification phase, and this requires a formal model of the application, that is not delivered by most of the UML languages used in industry.

With the aim of bridging the gap between the semi-formal languages used in practice and the formal notation required to prove correctness, an international cooperation project between Brazil and Germany, leaded by Prof. Hartmut Ehrig, started in 1994: the GRAPHIT project (DLR/CNPq) [BBC+00]. Originally, the project involved two universities (TU-Berlin and UFRGS) and two industrial partners (MSB and Nutec), and later other partners joined the project (Universität Stuttgart and PUCRS). The idea was to develop a specification language that, despite being formal, had a graphical layout, preferably following the object-oriented style used in practice, and relying on few simple and powerful concepts. This would allow users to easily understand and build specifications on this language. Graph grammars seemed to be a perfect basis for this aim due to some of its inherent characteristics: graphical description of states (even complex ones can be better understood using a suitable graph representation), changes of states can be easily specified via relationships between graphs (rules), concurrency is naturally described (implicitly, through parallel applications of rules). In the following years there have been a lot of research activities in order to define a suitable specification language as well as structuring and analysis techniques for distributed and concurrent object-based systems in the GRAPHIT as well as in follow-up projects. In this paper we will review the main results we obtained for specification, simulation, animation and verification of concurrent object-based systems, presenting the steps to be followed to use each of them in order to improve the software development process.

One of the results of the GRAPHIT project was the development of the specification language Object-Based Graph Grammars (OBGG) [DR00]. This language have been strongly influenced by the composition operators for graph grammars presented in [Rib96] (a Ph.D. thesis advised by Prof. Hartmut Ehrig). The main idea is that a system is constructed by composing objects belonging to different classes, and each class is specified with a graph grammar (with special characteristics that give the language an object-based style, that is natural for non-academic users). In section 2 we present the main concepts of Object-Based Graph Grammars.

To be really useful in practice, there must be techniques (and possibly tools) for the analysis of a specification. With this aim we have built a simulation tool for OBGG (the PLATUS tool[4]). With this tool it is possible not only to execute the system to validate strategies, but also to make quantitative analysis (e.g. count the number of messages exchanged in the system, the number of created objects). In section 3 we describe how OBGG specifications are simulated.

Another result of the GRAPHIT cooperation was the development of the GenGED concept and environment [Bar00] (a Ph.D. thesis advised by Prof. Hartmut Ehrig). This is a visual environment based on graph grammars for the definition of visual languages. In section 4 we show how to use GenGED to provide animation of specifications written in OBGG[5].

The possibility of analyzing specifications was a main aim since the beginning of GRAPHIT. After defining the specification language OBGG it was possible to define verification techniques that could make use of existing model checking tools in a suitable way (due to the restrictions imposed in OBGG, verification of many properties became feasible – see [LRT02] for examples). We have used different approaches for verification: translation to PROMELA (the input-language of the SPIN model checker [Hol97])[6], translation to π-calculus [Mil99] [7] and use of unfolding semantics to verify properties directly for the OBGG specification [BCK00]. The results about verification[8] are presented in section 5.

2 Object-Based Graph Grammars

In this section we propose to use a formal method, namely graph grammars, to describe concurrent object-based systems. The main characteristics of this formalism are its graphical layout, rube-basedness, natural description of complex states, inherent concurrency and non-determinism, object-basedness.

The specification of each kind of object (class) that compose an object-based system is done via an (object-based) graph grammar. Before discussing in more details the description of a class, we will present the kind of graphs and rules that will be used for the specification of object-based systems (Sect. 2.1). These graphs are called *object-based graphs* and were introduced in [DR00]. Then we show how classes are specified (Sect. 2.2 and how they are put together to form an object-based system (Sect. 2.3). Finally, we discuss behavior of a system defined using OBGG (Sect. 2.4) and present an example (Sect. 2.5).

[4] The underlying concepts of the simulator, as well as a prototype of the tool were developed in the PLATUS project (CNPq).

[5] The use of GenGED for animation of OBGG was a result of the German-Brazilian cooperation project DACHIA (IB-BMBF/FAPERGS), involving the universities TU-Berlin, Universität Stuttgart, UFRGS and PUCRS, and the industry MSB.

[6] Investigations on this topic were carried out in the scope of the ForMOS (FAPERGS/CNPq) and CASCO (in collaboration with HP-Brazil) projects.

[7] Results obtained in the IQ-Mobile (Italian-Brazilian cooperation CNPq/CNR) and ForMOS projects.

[8] Investigations on verification were obtained in the scope of the ForMOS, CASCO, IQ-Mobile and and DACHIA projects.

2.1 Object-Based Graphs and Rules

Each graph in an object-based graph grammar is composed by instances of the vertices and edges shown in Figure 1(a). The vertices represent object identities/classes and abstract data types, whereas messages and attributes of objects are modeled as hyperedges (edges with one destination and many source vertices). We defined a distinguished graphical representation for these graphs to increase the readability of the specifications (see Figure 1(b)). Elements of abstract data types are allowed as attributes of classes and/or parameters of messages. Note that the graph in Figure 1 defines only a scheme of which kinds of vertices and edges may occur in a specification, and does not oblige classes or messages to have attributes. For example, this graph specifies that, if an class has attributes, they must be either of type ADT or of type Class.

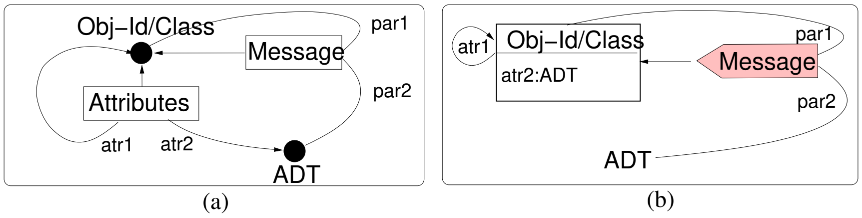

Fig. 1. (a) Object-Based Graph Scheme (b) Graphical Representation

A rule expresses the reaction of an object to the receipt of a message. A rule of an object-based graph grammar consists of:

- *a left-hand side L*: describes the items that must be present in the current state to enable the application of the rule. The restrictions imposed to left-hand sides of rules are:
  - There must be exactly one message vertex, called trigger message (this is the message treated by this rule);
  - Only attributes of the class that is the target of the trigger message may appear (in the graphical representation, not all attributes of this class must appear, only the ones necessary for the treatment of this message);
  - Items of type ADT may be variables, that will be instantiated at the time of rule application. Operations defined in the ADTs may be used.
- *a right-hand side R*: describes the items that will be present after the application of the rule. It consists of:
  - Objects: all objects and attributes present in the left-hand side of the rule, as well as new objects (created by the application of the rule); The values of attributes may change, but attributes can not be deleted;
  - Messages to all objects appearing in R are allowed.
- *a condition*: that must be satisfied for the rule to be applied. This condition is an equation over the attributes of left- and right-hand sides.

Formally, we use typed attributed hypergraphs and rule is a (partial) graph homomorphism with application conditions. See [EHK$^+$97] for more details on this graph transformation approach.

2.2 Specification of Classes

A *class* is composed of a *type graph, a set of rules* and an *initial graph.*

Type Graph : a graph containing information about all attributes of this class (either of ADT types or references to other classes) and messages sent/received by objects of this class. This graph can be seen an instantiation of the object-based graph scheme described above.

Set of Rules : these rules specify how the objects of this class will behave when receiving messages. For the same kind of message, we may have many rules specifying the intended behavior. Depending on the conditions imposed by these rules (on the values of attributes and/or parameters of the message), they may be mutually exclusive or not. In the latter case, one of them will be chosen non-deterministically to be executed. Note that the behavior of an object when receiving a message is not specified as a series of steps that shall be executed, but rather as an atomic change of the values of the object attributes together with the creation of new messages to other (or the same) objects. That is, there is no control structure to govern the application of the rules that specify the behavior of an entity. Our approach is data driven. This has the advantage that unnecessary sequentializations of computation steps are avoided because the specifier only has to care about the causal dependencies between events.

Initial Graph : this graph can be seen as a *template* for the creation of objects. It specifies the initial values of attributes of the objects of this class, as well as messages that must be sent to these objects when they are created. These values can be either concrete or variables. If they are variables, they shall be seen as instantiation parameters (see discussion on object creation below).

There are two ways of creating objects in an object-based system: either they are created in the beginning (initial state) or they are created during the execution of the system. These are called *static* and *dynamic creation*, respectively. In both cases, the created objects must use the initial graph template of the corresponding class.

Static object creation : This situation occurs when the user composes the initial state of a system, by creating instances of various different classes and linking them. To create an object one must instantiate the attributes that correspond to variables in the definition of the initial graph of the corresponding class, as well as assign concrete objects to the attributes that are references to other objects. All messages that belong to the definition of the initial graph of the class will be automatically sent to this object in the initial state of the system.

Dynamic object creation : During the execution time of the system, new objects may be created. Objects of any class may request the creation of objects of other (or the same) class, the only requirement is that the the structure of the *creation message* of the class of the newly created object is known by the object that requests the creation. Each class has an associated *creation* *message*, that is a message sent to objects of this class that has as parameters values that are needed to create the initial state of an object of that class (instantiation parameters). Moreover, there is a rule, called *creation rule*, that treats this creation message: it removes the message and creates the internal structure of the object according to the definitions of the initial graph template of the class and the parameters of the creation message. Only after the application of this rule the object is executable (that is, before the application of this rule, the vertex corresponding to the object may be part of the system, but no message will be neither treated nor sent by this object). The creation message and creation rule are automatically obtained from the definition of the initial graph template of each class, and can not be changed by the user.

2.3 Specification of Object-Based Systems

To build an object-based system, the user shall choose all classes that will be compose his/her system, connect them in a suitable way, and create an initial state for the system consisting of objects of these entities. More concretely, the steps to be followed are:

Step 1 : Choose the classes that will compose the system.

Step 2 : Connect the classes chosen in step 1. This is accomplished by relating the entity vertices and message edges in the type graphs of the involved graph grammars(classes). This is necessary because the names of classes/messages in each type graph may not agree with each other, and with this relationship we say which items shall be the same and which shall be different, regardless of the names used in the specification of the classes.

Step 3 : Generate a grammar that corresponds to the whole system, using the grammars chosen in step 1 and the connection defined in step 2[9]. Note that the relationship defined in step 2 over the type graphs induces a relation on the initial graphs and rules of the classes.

Step 4 : Create the desired objects of each of the classes that compose the system. This corresponds to the *static creation* of objects discussed above.

2.4 Behavior

Each *state* of a computation described by a graph grammar is a graph that contains objects (with their internal structure) and messages to be treated. A rule r is *enabled* in a state S by a message m belonging to the state S if m is the message that triggers r (the message deleted by r), each attribute that is

[9] Formally, this step is implemented as a composition (via a universal construction) of the corresponding graph grammars using the morphisms induced by the relationship defined in step 2. More details about this composition operator can be found in [Rib96,Rib99].

preserved by the rule have the necessary value in S[10], each attribute in that is changed by the rule have the necessary value in S, and the condition of the rule is true in S. The effect of this *rule application* is that all attributes of object that are not preserved will receive the new values as defined in the rule, message m will be deleted and the new messages that appear in the right-hand side of the rule will be created (new objects may be created as well). Nothing else in the state is modified by this rule. Formally, this corresponds to a pushout in the corresponding category [DR00].

At each execution state, several rules may be enabled (and therefore they are candidates for execution at that moment). Rule applications only have local effects on the state. However, there may be several rules competing to update the same portion of the state. To determine which set of rules will be applied at each time, we need to choose a set of rules that is consistent, that is, in which no two or more rules have write access to the same resources (non-conflicting rule applications). Due to the restrictions imposed in object-based graph grammars, conflicts can only occur among rules of the same class. When such a conflict occurs, one of the rules is (non-deterministically) chosen to be applied. The semantics of the whole system is the set of all possible computations [Kor96] (or, equivalently, a tree containing all computations [Rib96]).

2.5 Example: Dining Philosophers

In this section we model the dining philosophers problem using OBGG. Traditionally the dining philosophers problem is described in the following scenario. There are N philosophers sitting in a table that has N forks (a fork between every philosopher). The philosophers spend some time thinking, and from time to time a philosopher gets hungry. In order to eat a philosopher must, exclusively, acquire its left and right forks. After eating a philosopher release both left and right forks and starts thinking again. Using OBGGs we modeled the problem with two classes: *Fork* and *Phil*, depicted in Figures 2 and 3[11]. Each *Fork* object is composed of a boolean attribute (*acquired*) that determines if the fork is currently in use by a philosopher (*acquired* true) or not (*acquired* false). Each *Phil* object is composed by two reference attributes to *Fork* objects (modeling its left (*leftFork*) and right (*rightFork*) forks) and five boolean attributes: *acquire* (the philosopher is trying to acquire the forks), *eat* (the philosopher is eating), *release* (the philosopher is releasing its acquired forks), *asym* (indicates if the philosopher starts getting the left fork (false), or the right fork (true)), and *forks* (used to control the number of acquired forks). The rules for the *Phil* and *Fork* objects represent the behavior of these objects (creation rules are omitted). A philosopher starts execution, rules *AsymStart* or *SymStart*, trying to acquire its left

[10] A rule may require that an attribute has a certain value, for example, rule *SymStart* of Figure 2 can only be applied if `acquire` has the value `true` in state S

[11] The numbers inside the circles are used to allow a clearer graphical representation of the type graph (preventing the some arrows to be very long and/or cross the picture) as well as to indicate how the objects that appear in rules and initial graphs are mapped to the type graph.

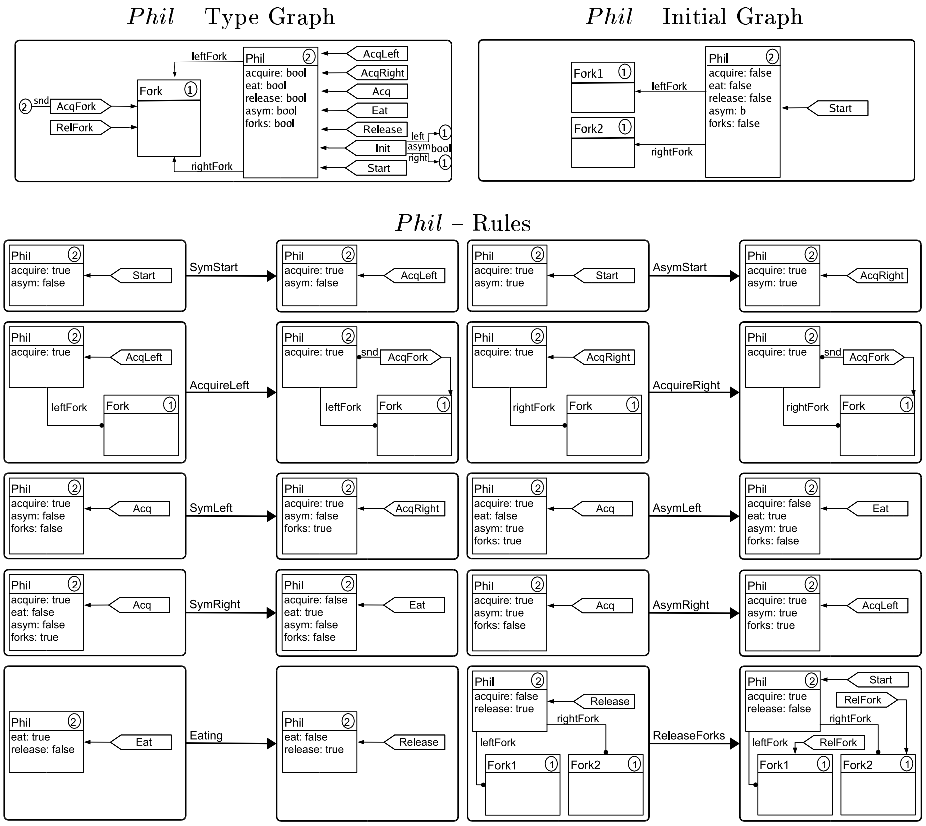

Fig. 2. *Phil* Class

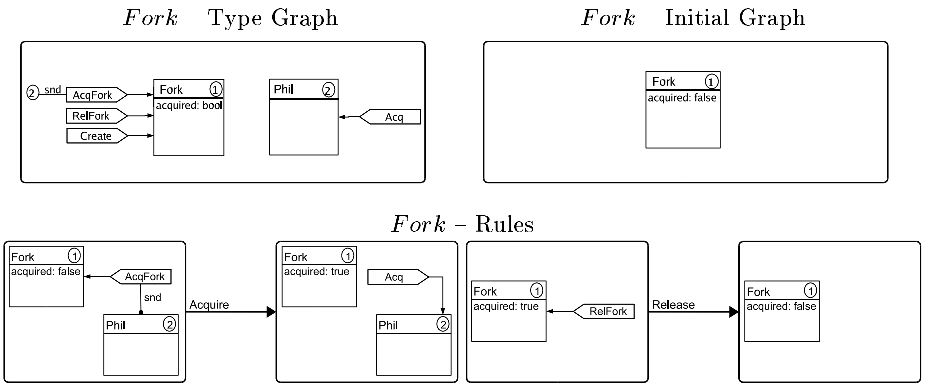

Fig. 3. *Fork* Class

(*asym* false) or right (*asym* true) forks (rules *AcquireLeft* and *AcquireRight*). If the philosopher can acquire the fork (rule *Acquire*), he tries to acquire the other fork (rules *SymLeft* or *AsymRight*). If the philosopher can acquire it too (rules *SymRight* or *AsymLeft*), he starts eating (rule *Eating*). After eating the philosopher releases his forks and starts all over again.

When building a model for the dining philosophers using OBGG, after choosing the two classes above (step 1) and suitably connecting them (step 2), we can generate a grammar for the complete system (step 3), and choose an initial graph for this system (step 4). In Fig. 4 we show an initial graph for an asymmetric solution for the problem. This solution is asymmetric because the philosopher *Phil2* has its *asym* attribute set to true, meaning that he will try to acquire the right fork first, differently from the other philosophers.

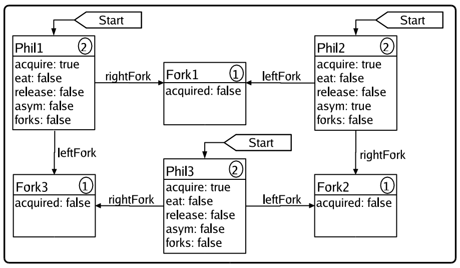

Fig. 4. Initial Graph for an Asymmetric Solution

3 Simulation of OBGG Specifications

One of the main advantages of using simulation models while developing an application is the possibility to *validate* strategies as well as control algorithms even before their implementation. For instance, simulation 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; check the applications behavior under various environment conditions (e.g. failure simulation); obtain quantitative results such as number of messages exchanged, number of rules applied for each object, number of objects created dynamically, among others.

The simulation of OBGGs must ensure that the formalism is faithfully represented. Analyzing the model characteristics we conclude that an OBGG model is a discrete-state system: the state of the objects change as rules are executed in response to the messages. The use of discrete-event simulation is thus a natural approach to model evaluation of such systems.

An OBGG specification is mapped to a simulation model in two main parts: one is the simulation kernel, the other is the representation of internal activities of objects. The kernel is responsible for passing messages among objects (i.e. dispatching events to simulation entities) and managing the global simulation time. The simulation time is needed to ensure the progress of the simulation process and to simulate the parallel execution of rule applications. OBGG messages are not timed or ordered in any way. The chosen kernel algorithm follows a conservative approach and the simulation time has a centralized control. The kernel keeps a list of messages (or events), known as event list. The process of simulation in the kernel consists in selecting event(s) (message(s)) according to the OBGG semantics; delivering those events (messages) to the target simulation entities (objects); await each recipient entity (object) to handle the message; advance the simulated time and start selecting events again.

As for the behavior of objects, in an OBGG model we may have concurrent activity in different objects or within a same object. A multi-threaded approach is natural to model concurrency among objects. Each object will have an associated thread to independently process messages (events) from its input buffer. The internal thread of each object is responsible for processing the incoming messages and applying appropriate rules, respecting the OBGG semantics. Within an object, all non-conflicting enabled rules may be applied in parallel[12]. For more details on the internal behavior of a simulation entity (representing an OBGG object) see [CMR00,DDCR02].

A simulation environment for OBGG was constructed [CR98,CMR00,DDCR02] and used to simulate various applications. According to the ideas above, the simulation environment is composed by a library that offers: the kernel functionality; the basic internal behavior of an object (presented above); the basic structure for a rule; the basic structure for a message. More concretely, this library is offered in an object-oriented language (Java) and these are the main classes comprising the library. OBGG models are mapped to simulation models in Java as follows: i) each OBGG class maps to one class specializing class Entity from the library, which will have the corresponding attributes and an internal thread performing as discussed above; ii) each rule specifying part of the behavior of an OBGG class is mapped to a class specializing class Rule; iii) messages handled by OBGG classes map to specializations of class Message; iv) the initial graph is mapped accordingly to a initial state of the simulation environment through the creation of the appropriate entity instances, messages and attributes initialization.

With this, to simulate an OBGG specification we have the following steps:

Step 1 – Translation of the OBGG specification to a simulation model : According to the ideas discussed above, a defined algorithm is used to translate an OBGG specification to a Java program extending pre-defined classes and interacting with the simulation kernel;

Step 2 – Simulate : Once the initial state is built, the simulation takes place generating a log with all messages exchanged among instances and the kernel and rules applied in each instance, along with the simulation time these events occurred.

Step 3 – Analysis of results : The user may then use the log to analyze the phenomena of interest. Simulation was shown to be very useful to find speci-

[12] Due to the encapsulation imposed by the object-based modeling, there can be no conflicts among objects.

fication errors (e.g. missing behavior (rules), wrong rule left-hand side specification leaving messages blocked, etc.) as well as to estimate the communication behavior in terms of total number of exchanged messages and helping to estimate the number of messages in a round (number of causally-dependent messages generated/deleted to complete a service). Alternatively, the simulation log may be submitted for animation using the ideas described in the next section.

4 Animation

In order to have a better visualization of the states and behavior of a system, it is useful to define an animation for the specification, if possible, using symbols that are close to the application domain to ease the understanding. To animate the specification of an object-based system $OBSys$ using OBGGs, there are 4 steps to be followed:

Step 1 – Construction of Animation Grammars : The specification of each class C of $OBSys$ is extended by animation information, that are attributes that are used during the animation and/or by messages that trigger animation events. This new grammar is called animation grammar for class C ($AnimC$). Note that this information shall not interfere in the behavior of the original OBGG C (just extra attributes may change and extra messages may be sent at each rule application).

Step 2 – Definition of Animation Interfaces : Now the user must define the attributes/messages that shall be visible for the one that will define the concrete animation module for the application. This is an abstraction process, resulting in a grammar for each class, called animation interface. This interface is a tool for the users of this class: they do not have to know the internal structure and behavior of objects of this class to choose the desired graphical representation (this guarantees encapsulation). In fact, the same class may have many different graphical representations, depending on the way it is used (see next step).

Step 3 – Definition/Choice of Animation Modules : An *animation module* consisting of concrete graphical representations for the items in each class must be defined (or chosen, if a library of animation modules is available). An animation module can be specified using GenGED, an approach with tool support for the visual definition of visual languages based on graph grammars and graph transformation, respectively. Using GenGED, the animation of each class is considered as visual language. Note that, in order to build the animation of a class, only the information present in the animation interface is necessary (other attributes and messages can only be seen by the developer of a class).

Step 4 – Animate : For the animation, we first have to generate the initial visualization of the system, based on the initial state of the object-based system (generating visualizations for all objects of this initial graph). Then, while simulating the OBGG corresponding to the whole system, animation messages shall be sent to the animation view generated by GenGED(these messages trigger the changes in the graphical representation).

The specification of a visual language using GenGED [Bar00,BEW02] is given by an alphabet and a grammar. An alphabet defines the language vocabulary, i.e. the symbols and how they are connected: symbol descriptions consist of a textual name and a corresponding graphic, links are given by arcs between textual (symbol) names and graphical constraints due to the (symbol) graphics involved. The type graph of a class can be used to generate an alphabet, i.e., it establishes a type system for all instances: vertices are modeled as attributed graph nodes where the attributes describe the graphics and arcs are modeled as graph edges where for each edge there exists at least one graphical constraint. The animation using GenGED will be triggered by message passing. For this purpose, messages have to be assigned to symbols in the alphabet. Since messages should not be visualized, they are assigned according to the textual names occurring in an alphabet only. Such an alphabet has to fit to the type graph of a class. Moreover, the initial graph and the rules defined using GenGED have to fit to the corresponding components of the interface. Each rule describes how the animation shall change when receiving messages defined in the animation interface of the module. Further rules than those declared by the interface may be defined for animation purposes.

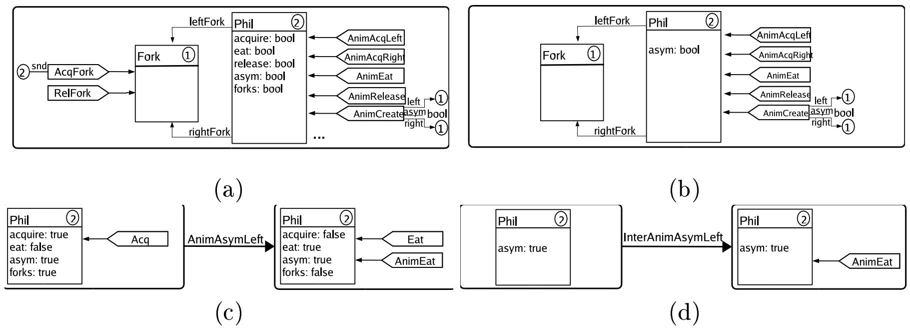

Fig. 5. (a)(c) Animation Grammar (b)(d) Animation Interface

Figure 5 shows an example of animation components for the dining philosophers. In (a) we can see the type graph of the animation grammar for the *Phil* class (this type graph consists also of all messages that appear in Figure 2, but these were omitted here). A rule of this grammar can be seen in (c) – it is an extension of a corresponding rule of the *Phil* grammar. In (b) we can see the type graph of the animation interface, consisting only of items that shall be visible for the users of the class and developers of animation modules, and in (d) we see a rule of the animation interface that corresponds to the concrete rule (c).

5 Verification of OBGG Specifications

Although simulation is a useful analysis method, it does not allow one to make definitive assertions about the behavior of a system. When building concurrent systems, often distributed, it is not obvious that such systems will obey certain properties. When dealing with critical systems, this becomes mandatory. Verification is thus an important analysis method.

In this sense, methods and tools to allow the model checking of OBGG specifications were developed. Model checking is an appealing verification method since it does not require advanced users know-how during the verification process (in contrast to theorem proving). Instead of developing a model checking tool for graph grammars or OBGG we focused on translating our models to formal languages that could serve as input for existing model checking tools, which is a common practice. Our approach seeks to reuse the existing implementations of model checkers, as well as take immediate advantage of their enhancements.

Starting with [FR04], efforts were invested to translate OBGG specifications to π-Calculus. The π-calculus [Mil99] is a well known and established formalism for description of semantics of concurrent systems. There are some automatic checkers for this formalism, for example, HAL [FGM$^+$98] and MWB (Mobility Workbench) [VM94]. Objects and messages are defined as processes of the π-calculus (agents) that communicate through a local channel. The source object and the parameters of each message are represented as parameters of message agents. The reactions of the objects when receiving a message are described by rule agents, that compose the object agent. These rule agents are identified by a kind of treated message and by a "rule" identifier. A "rule" describes the procedures to treat a message. Each kind of message can have several procedures for treating it, so we may have several rule agents that describe the treatment of the same kind of message. The choice of procedure to be executed is non-deterministic. This is described by composition of rule agents (for a same kind of message) with the Sum operator (+), without guard. The concurrency between objects is modeled by parallel composition of object agents and message agents. So each object can treat its messages in parallel. The internal concurrency is modeled by recursion of object agents. Although the semantic compatibility of the translation could be shown, two main problems were faced: i) models had to be hardly abstracted/restricted such that OBGG objects had no internal state (the overhead to model attributes in π-calculus would be great); ii) when using existing π-Calculus model checkers, we found some limitations, specially while supporting the replication operator of π-Calculus. Moreover, the π-calculus tools were not very optimized, and thus only small examples could be translated.

Therefore, we invested efforts in a second translation, from OBGG to PROMELA (PROcess/PROtocol MEta LAnguage) [pro03], the input language of SPIN (Simple Promela INterpreter) [Hol97]. PROMELA is based on processes and provides shared memory as well as (CSP-like) channels for interprocess communication. According to the proposed translation, presented in [DFRS03a], OBGG objects are mapped to PROMELA processes. For verification purposes, attributes of OBGG objects are restricted to the types supported in PROMELA (boolean, char, int, array). Moreover, OBGG messages are translated to PROMELA messages, and the receipt of messages is done through an asynchronous channel (that is defined for every translated object), which is also used as reference to translated objects. Rules for OBGG objects are mapped to a condition structure inside the translated object, and the OBGG initial graph becomes a initial process in PROMELA. Concurrency among objects is naturally preserved by the concurrency between translated objects (processes). More details about the translation can be found in [DFRS03a], including a discussion of the semantic compatibility for the generated PROMELA model and the original OBGG model.

Complementary to the translation of the model to the input language of a model checker, an approach is needed to the specification of properties using a given temporal logic. Although it is quite appealing to think about graphical properties when using graph grammar specifications, this subject is still to be investigated. Up to now, in our work, we have considered properties about the history of rule applications starting from the initial state.

In [DFRS03a] and [SDR04] we define how to specify properties over OBGG models using Linear Temporal Logics (LTL) – the same temporal logic used in SPIN. LTL properties are defined over events of the model, where an event is a rule application. Since rule applications are local to objects (i.e. transform the internal state of an object), an event is composed by the identification of the object, the name of the rule applied, and any internal attributes of interest, for verification, of the object that suffered the transformation. Moreover, the graphical presentation of counter-examples in terms of OBGG abstractions, instead of the translated PROMELA model, is provided.

Considering the above discussed ideas, we have the following steps for model checking of OBGG specifications:

Step 1 – Translate the OBGG specification to a verification model: An OBGG specification is translated, following a defined algorithm, to a PROMELA model which serves as input to the SPIN model checker;

Step 2 – Definition of properties in temporal logic: Currently, LTL formulas can be proposed taking into consideration the application of rules transforming the internal attributes of OBGG instances, as well as resulting attribute values of the transformation;

Step 3 – Model checking: With the model and the properties to verify, the model checking tool can be initiated;

Step 4 – Analyze results: Step 3 may have different outcomes: i) the model satisfies the property; ii) the model does not satisfy the property and a counter example is generated which can be visualized in terms of OBGG abstractions (instances, messages and rule applications); iii) the verification terminates abnormally due to insufficient size of the channels. To respect the OBGG semantics, channels must always have room for incoming messages since otherwise the sending process would block and this would mean blocking an OBGG instance while sending a message, which is not acceptable. In such cases the specifier may define larger channels and try again; and

iv) the verification terminates due to insufficient resources, such as memory. It is known that model checking techniques can be very time and space consuming.

For the dinning philosophers example the following properties were shown:

- Liveness: the asymmetric version is shown to be deadlock free while deadlocks may occur in the symmetric version. To prove this, an LTL formula was used to specify that "it is always possible that some philosopher will eat". More concretely it is proven that it is always possible, in the future, to apply the *Eating* rule.
- Safety: in a setting with up to three philosophers it is sufficient to prove that "no two philosophers might be eating at the same time". The approach taken is to prove that, if a philosopher starts eating (rule *Eating*) then until he releases the forks (rule *ReleaseForks*) no other philosopher will start eating (rule *Eating*).

6 Conclusions and Future Work

The development of concurrent and distributed systems is challenging. This paper presented how to use OBGG for the modeling and analysis such systems. Beyond the basic abstractions provided by the language, OBGG is attractive due to the diversity of development methods and tools supported (see [DDF$^+$04] for a description of the tools). To validate this specification language, we carried out a series of case studies: mobile code applications [DR00] - analyzed via simulation; active networks [DD04] - analyzed via simulation; the pull-based failure detector [DRS03] - analyzed through simulation; the readers and writers problem [SDR04] - analyzed through verification, where it is shown that writers can starve while readers proceed; the distributed election in a ring algorithm [DFRS03b] - analyzed through verification, where properties could be shown: "eventually there will be one element of the ring elected" and "there can be no two elements of the ring simultaneously elected"; message ordering [DPS04] - analyzed through verification where it could be shown that a message ordering mechanism delivers the messages in the expected order.

In order to allow the specification and reasoning about distributed systems in the presence of faults, in [DSR03,DRS03] an approach for the insertion of classical fault behaviors in OBGG models of distributed systems has been presented. It was shown that, due to the abstractions provided by OBGG it is possible to transform an OBGG model to embed the behavior of a given fault model. While we have analyzed distributed systems in the presence of faults using simulation, currently we are investing efforts to analyze such systems using the model checking approach supported.

The object-based nature of the language and its application to open distributed systems naturally calls for verification methods for reasoning over partial systems. Thus, efforts are being invested towards methods for the verification of partial systems described in OBGG (first results can be found in [DPS04]).

Moreover, the OBGG language itself can be improved, for example, by adding inheritance and module concepts. This would greatly simplify an integration with UML, in the sense that OBGG can be used as a semantical model for many UML languages, integrating in a smooth way the different views provided by UML. For this, translations from these UML languages into OBGG should be defined. In this way, we could use all techniques and tools available for OBGG also for specifications using the UML languages.

Acknowledgments
Altogether the series of achievements (and cooperation projects) presented in this paper were influenced (directly or indirectly) by the ideas and stimulus of Hartmut Ehrig. Besides his own technical contributions to Computer Science and therefore qualified technical advisoring of advanced students, Hartmut is also concerned with the academic maturity of his students, encouraging them to present their results to the academia and giving them the opportunity to work in cooperation projects to exercise how to combine efforts to reach more relevant contributions to Computer Science. We would like to thank him heartly for the technical discussions (that led to many results presented here), carrier advices (that helped us to continue the work started in Berlin) and friendship (encouraging us to pursue our ideas).

References

[Bar00] R. Bardohl, *Visual Definition of Visual Languages based on Algebraic Graph Transformation*, Ph.D. thesis, Technical University of Berlin, Germany, 2000.

[BBC+00] B. Bardohl, R. Bardohl, P. Castro, B. Copstein, H. Ehrig, M. Korff, A. Martini, D. Nunes, L. Ribeiro, and H. Schlebbe, *GRAPHIT: Graphical support and integration of formal and semi-formal methods for software specification and development*, 6th German-Brazilian Workshop on Information Technology, 2000.

[BCK00] P. Baldan, A. Corradini, and B. Koenig, *Verifying finite-state graph grammars: an unfolding-based approach*, Proc. of CONCUR 2004, LNCS, Springer, 2000.

[BEW02] R. Bardohl, C. Ermel, and I. Weinhold, *Specification and Analysis Techniques for Visual Languages with* GenGED, Tech. Report 2002–13, Technical University Berlin, Dept. of Computer Science, September 2002, ISSN 1436-9915.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language user guide*, Addison-Wesley, 1998.

[CMR00] B. Copstein, M. C. Móra, and L. Ribeiro, *An environment for formal modeling and simulation of control systems*, 33rd Annual Simulation Symposium (USA), IEEE Computer Society Press, 2000, pp. 74–82.

[CR98] B. Copstein and L. Ribeiro, *Specifying simulation models using graph grammars*, 10th European Simulation Symposium (UK), SCS, 1998, pp. 60–64.

[DD04] L. Duarte and F.L. Dotti, *Development of an active network architecture using mobile agents - a case study*, Tech. Report TR-043, PPGCC-FACIN-PUCRS, Brazil, 2004.

[DDCR02] F. L. Dotti, L. M. Duarte, B. Copstein, and L. Ribeiro, *Simulation of mobile applications*, 2002 Communication Networks and Distributed Systems Modeling and Simulation Conference (USA), The Society for Modeling and Simulation International, 2002, pp. 261–267.

[DDF+04] F. L. Dotti, L. Duarte, L. Foss, L. Ribeiro, D. Russi, and O. Santos, *An environment for the development of concurrent object-based applications*, ENTCS (International Workshop on Graph-Based Tools) (2004).

[DFRS03a] F. L. Dotti, L. Foss, L. Ribeiro, and O. M. Santos, *Verification of object-based distributed systems*, 6th International Conference on Formal Methods for Open Object-Based Distributed Systems (France), LNCS, vol. 2884, Springer-Verlag, 2003, pp. 261–275.

[DFRS03b] F.L. Dotti, L. Foss, L. Ribeiro, and O.M. Santos, *Specification and formal verification of distributed systems (in portuguese)*, 17th Brazilian Symposium on Software Engineering (SBES), 2003, pp. 225–240.

[DPS04] F.L. Dotti, F. Pasini, and O.M. Santos, *A methodology for the verification of partial systems modelled with object based graph grammars (in portuguese - accepted for publication)*, 18th Brazilian Symposium on Software Engineering (SBES), 2004.

[DR00] F. L. Dotti and L. Ribeiro, *Specification of mobile code systems using graph grammars*, 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (USA), IFIP Conference Proceedings, vol. 177, Kluwer Academic Publishers, 2000, pp. 45–64.

[DRS03] F. L. Dotti, L. Ribeiro, and O. M. Santos, *Specification and analysis of fault behaviours using graph grammars*, 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (USA), LNCS, vol. 3062, Springer-Verlag, 2003, pp. 120–133.

[DSR03] F.L. Dotti, O.M. Santos, and E.T. Rödel, *On the use of formal specifications to analyse fault behaviors of distributed systems*, First Latin-American Symposium on Dependable Computing, LNCS, vol. 2847, Springer, 2003, pp. 341–360.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini, *Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach*, Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations (G. Rozenberg, ed.), World Scientific, 1997.

[Ehr79] H. Ehrig, *Introduction to the algebraic theory of graph grammars*, 1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology (Germany), Lecture Notes in Computer Science, vol. 73, Springer-Verlag, 1979, pp. 1–69.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider, *Graph grammars: an algebraic approach*, 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 167–180.

[FGM+98] G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, and G.Ristori, *Verifying Mobile Processes in the HAL Environment*, International Conference on Computer Aided Verification (Vancouver, CA), LNCS, vol. 1427, Springer, 1998, pp. 511–515.

[FR04] L. Foss and L. Ribeiro, *A translation of object-based hypergraph grammars into π-calculus*, Electronic Notes in Theoretical Computer Science **95** (2004), 245–267.

[Hol97] G. J. Holzmann, *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), no. 5, 279–295.

[ISO89] ISO, *Information processing systems — Open systems interconnection — Estelle — a formal description technique based on an extended state transition model*, 1989.

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

[Lam94] Leslie Lamport, *The temporal logic of actions*, ACM Trans. Program. Lang. Syst. **16** (1994), no. 3, 872–923.

[LRT02] A. B. Loreto, L. Ribeiro, and L. V. Toscani, *Decidability and tractability of a problem in object-based graph grammars*, 17th IFIP World Computer Congress - Theoretical Computer Science (Canada), IFIP Conference Proceedings, vol. 223, Kluwer Academic Publishers, 2002, pp. 396–408.

[LT89] N. Lynch and M. Tuttle, *An introduction to input/output automata*, CWI-Quarterly **2** (1989), no. 3, 219–246.

[McM93] K. L. McMillan, *Symbolic model checking*, Kluwer Academic Publishers, 1993.

[Mil89] R. Milner, *Communication and concurrency*, International Series in Computer Science, Prentice Hall, London, 1989.

[Mil99] R. Milner, *Communicating and mobile systems: the π-calculus*, Cambridge University Press, USA, 1999.

[Pet62] C.A. Petri, *Kommunikation mit Automaten*, Ph.D. thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[pro03] *Promela language reference*, http://spinroot.com/spin/Man/promela.html, 2003.

[Rei85] Wolfgang Reisig, *Petri nets*, EATCS Monographs on Theoretical Computer Science, vol. 4, Springer-Verlag, 1985.

[Rib96] L. Ribeiro, *Parallel composition and unfolding semantics of graph grammars*, Ph.D. thesis, Technical University of Berlin, Germany, 1996.

[Rib99] L. Ribeiro, *Parallel Composition of Graph Grammars*, Applied Categorical Structures **7** (1999), no. 4, 405–430.

[SDR04] O. M. Santos, F. L. Dotti, and L. Ribeiro, *Verifying object-based graph grammars*, ENTCS (2nd Graph-Transformation and Visual Modeling Techniques) (2004).

[vEVD89] P.H.J. van Eijk, C. A. Vissers, and M. ((editors) Diaz, *The formal description technique lotos*, Elsevier Science Publishers, 1989.

[VM94] B. Victor and F. Moller, *The Mobility Workbench - a tool for the π-calculus*, Proc. International Conference on Computer Aided Verification, CAV (David Dill, ed.), LNCS, vol. 818, Springer, 1994, pp. 428–440.

[Win89] G. Winskel, *An introduction to event structures*, Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Springer, 1989, pp. 364–397.

[WN94] G. Winskel and M. Nielsen, *Models for concurrency*, Tech. Report BRICS RS-94-12, University of Aarhus, 1994.