Formal Specification of Object-Based Distributed Systems*

Osmar Marchi dos Santos† and Fernando Luís Dotti

Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
ZIP 90619–900 – Porto Alegre, RS – Brazil
{osantos,fldotti}@inf.pucrs.br

Abstract. In this paper we present the main characteristics of a graphical and declarative formal specification language, called Object Based Graph Grammars (OBGG), suitable for the specification of distributed systems. We show how one can use OBGG by modeling the dining philosopher’s problem.

Resumo. Neste artigo são apresentadas as principais características de uma linguagem de especificação formal gráfica e declarativa, chamada Gramática de Grafos Baseada em Objetos (GGBO), apropriada para a especificação de sistemas distribuídos. É visto como utilizar GGBO, através da modelagem do problema dos filósofos glutões.

1. Introduction

The development of distributed systems is considered a complex task. In particular, assuring the correctness of distributed systems is far from trivial if we consider the characteristics of open systems, such as [Silva 1999]: massive geographical distribution; high dynamics (appearance of new nodes and services); no global control; faults; lack of security; and high heterogeneity. Among other barriers, in open environments (e.g. Internet) it is hard to assure the correctness of applications because it is difficult to identify whether a failure is caused by a fault in the system under construction itself or by the environment where it runs. Therefore, it is necessary to provide methods and tools for the development of distributed systems such that developers can have a higher degree of confidence in their solutions.

A visual formal specification language suitable for specifying distributed systems, called Object Based Graph Grammars (OBGG), was proposed in [Dotti and Ribeiro 2000]. Currently, models defined in this formal specification language can be analyzed through simulation [Copstein, Móra and Ribeiro 2000] [Dotti, Duarte, Copstein and Ribeiro 2002], and an approach to analyze models through verification has been proposed [Dotti, Foss, Ribeiro and Santos 2003a] [Dotti, Foss, Ribeiro and Santos 2003b]. Also, starting from an OBGG model, it is possible to generate code for execution in a real environment, following a straightforward mapping from an OBGG

* This work is partially supported by HP Brasil - PUCRS agreement CASCO (24º T.A.).
† This author is partially supported by CNPq (Brazil).
Anais do II Simpósio de Informática da Região Centro / RS – Santa Maria, agosto de 2003.

specification to the corresponding Java code [Dotti, Duarte, Copstein and Ribeiro 2002]. In order to deal with open environments, an approach considering classical fault models for distributed systems, still during the specification phase, allowing one to reason about a given model in the presence of a selected fault behavior was defined in [Dotti, Santos and Rödel 2003].

By using the methods and tools mentioned above, a framework to assist the development of distributed systems was defined. The innovative aspect of this framework is the use of the same formal specification language (OBGG) as the underlying unifying formalism [Dotti, Duarte, Silva and Andrade 2002]. In this paper, we focus on the presentation of the main characteristics of the OBGG formalism, since it is the first step in order to use the methods and tools briefly described above. We show an example on the use of OBGG, by modeling the dining philosopher’s problem.

This paper is organized as follows: Section 2 presents the main concepts of the formal specification language OBGG; Section 3 shows the modeling of the dining philosopher’s problem using OBGG; finally, Section 4 brings us to the conclusions and future works.

2. Object Based Graph Grammars

The use of a formal specification language allows the creation of a precise description of a system with a well-defined syntax and semantics. Since this semantics is based on a mathematical model, it is possible to apply formal verification techniques [Dèharbe, Moreira, Ribeiro and Rodrigues 2000] to show that a system specified in the formal language has some desired properties.

The formal specification language used in this work is based on a restricted form of graph grammars, called Object Based Graph Grammars (OBGG) [Dotti and Ribeiro 2000]. The basic notions of graph grammars [Ehrig 1979] are that the state of a system can be represented by a graph (the system state graph), and from the initial state of the system (the initial graph), the application of rules successively changes the system state. A rule is composed by a name, a left side, a right side, and a condition (Figure 1).

A rule can be applied whenever its left side is a sub-graph of the current system state graph and its condition is satisfied. This is called a match or occurrence of the rule.
When applied, the rule brings the system to a new state defined as [Dotti and Ribeiro 2000]:

- items in the left side not present in the right side are deleted;
- items in the right side not present in the left side are created;
- items present in both sides of the rule are preserved.

Multiple rules can be applied in parallel if there is no conflict between them, i.e. two or more rules can not modify the same item(s). When running into a conflict situation, one of the candidate rules to be applied is chosen non-deterministically.

OBGG is a restricted form of graph grammars with respect to the kinds of vertices, as well as the configuration of rules to represent object-based concepts. This is done by representing entities (classes in object-based systems) that aggregate attributes and react to messages. Messages themselves may have attributes, called parameters.

The behavior of an entity when reacting to a message is given by a (set of) rule(s). Therefore the left side of a rule always specifies the reception of a message by a specific entity. At the right side, that message is consumed and the effect of applying the rule is defined, which may be:

- change of attributes;
- creation of objects (instances of entities);
- generation of new messages.

This way, the application of a rule may leave the system state graph in a configuration such that other matches may occur. The system evolves until there is none possible match. Analogously to an object-based system composed by classes and objects, an OBGG specification is composed by different entities and its instances (objects).

Each entity, has a type graph (specifying attributes of the entity and messages it may receive), and a (set of) rule(s) that specify its behavior upon reception of a message. It is also possible to define different entities in the same type graph (as defined for the dining philosophers problem presented in the next Section). The specification of a system where several entities are involved is given by the specification of each separate entity complemented by the system initial graph that may have different objects and messages (addressed to these objects) in order to start the model.

3. Modeling the Dining Philosophers Problem

In this Section we model the dining philosopher’s problem using OBGG. The type graph for the entities that compose the specification is present in Figure 3. The rules for the entities are depicted in Figures 4, 5 and 6. By using the same entities presented below we define two different models for the problem, given by two different initial graphs, describing symmetric (Figure 7) and asymmetric (Figure 8) solutions for the dining philosophers problem.

Traditionally the dining philosopher’s problem is described in the following scenario. In a table there are $N$ philosophers and $N$ 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 his left
and right forks. After eating, a philosopher release both left and right forks and starts
thinking again. This behavior continues repeatedly.

In OBGG we model the problem with two entities: Fork and Phil. The
messages that these entities can receive and their attributes are described in Figure 3. It
is important to notice that the numbers inside the circles of the entities are used to
indicate the type of each entity. They are defined in the type graph and used as type
information for the instances that appear in the rules and initial graphs. Thus, entity
Fork is marked with number 1, and entity Phil is marked with number 2.

The Fork entity represents a fork, being composed by a boolean attribute
(acquired) that determines if the fork is currently in use by a philosopher (acquired
true) or not (acquired false). The Phil entity represents a philosopher, being composed
by 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). Each Phil object also
has two references to Fork objects: its left (leftFork) and right (rightFork) forks.

The rules for the Fork entity are shown in Figure 4. The behavior of a Fork
object is simple, and is mainly defined by the attribute acquired. When a Fork is not
acquired (acquired false) it can receive AcqFork messages and be acquired by
executing the Acquired rule. On the other side, when a Fork is acquired (acquired
true) it can not receive AcqFork messages and can not execute any kind of rule.
However, when a Fork is acquired it can receive RelFork messages and return to a
state where AcqFork messages can be received and applied.
The rules for the Phil entity are depicted in Figure 5 and 6. The behavior of a Phil object is as follows. A philosopher starts its execution, rules SymStart or AsymStart, trying to acquire its left (asym false) or right (asym true) forks (rules AcquireLeft and AcquireRight). If the philosopher can acquire the fork, 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, by sending itself an Eat message and applying, at the reception of this message, a Eating rule. After eating, the philosopher release its forks and starts the process all over again (rules ReleaseForks).
In Figure 7 we show an initial graph for a symmetric solution of the problem. This solution is symmetric because all the philosophers have their \textit{asym} attribute set to
false, meaning that all of them will try to acquire the left fork first. This is not a valid solution for the problem, since the philosophers will enter in a deadlock situation.

Figure 7. Initial graph for symmetric solution

Figure 8 illustrates a valid solution to the dining philosopher’s problem. This solution is called asymmetric because one philosopher breaks the symmetry of the problem by first requesting a different fork. That is, in contrary to philosophers Phil1 and Phil3 that requests the left fork first, we set the asym attribute of Phil2 to true, meaning that it will try to acquire the right fork first.

Figure 8. Initial graph for asymmetric solution

4. Final Remarks

In this paper we have presented the main characteristics of the formal specification language OBGG, which are: message passing, non-determinism in the application of rules, and implicit parallelism. We have also exemplified the use of OBGG, by modeling the dining philosopher’s problem. However, in this paper, we have not presented the use of methods and tools currently available for OBGG.

Currently, for OBGG, there are three implemented tools: a simulator for OBGG models, a code generator, and an interface for the edition of OBGG models. We intend to expand the interface, used for the edition of models, in order to create a development environment for distributed systems.
Another feature, not implemented yet, is a verification tool for OBGG models. This tool is under development and its root consists on the translation of OBGG specifications to the input language of the SPIN model checker [Holzmann 1997].

References


