Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 184 (2007) 113–131
www.elsevier.com/locate/entcs

Code Generation for Parallel Applications Modelled with Object-Based Graph Grammars [1]

Fábio Pasini, [2,3] Fernando L. Dotti[4]

Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil

Abstract

During the development of a parallel application, besides being able to analyze performance aspects, it is highly desirable to be able to assure functional properties as early as possible. Assuring functional properties about a model of the parallel applicationcan lead to important savings since it reduces the time spent in application development and debugging. In this direction, model-checking and automatic code generation can be used as complementary tools during the development, making possible to analyze the system behavior and allowing the fast generation of corresponding code. In this paper we propose the use of Object-Based Graph Grammars (OBGG) for the specification of parallel applications. OBGG is a formal, visual language suited for the description of concurrent systems based on asynchronous message passing. Models described using OBGG can be verified through model checking. Following this approach, a translation from OBGG models to C code using MPI (Message Passing Interface), which is suited for clusters, is presented. To illustrate the contribution, a sample parallel application is modelled in OBGG; functional properties of the model are proven by model-checking; the C/MPI corresponding model is presented and performance results of the translated model are discussed and compared with an analogous C/MPI application built by hand.

Keywords: Graph Grammars, Parallel Programming, MPI, Model Checking.

1 Introduction

Concurrent systems can present behaviors that are difficult to predict. Even when the presented algorithmic solution inspires a high confidence degree, aspects such as race conditions and deadlocks, that are sometimes not easy to avoid, may be latent in the software and compromise the systems functioning.

During the development of parallel applications, as of any concurrent application, one must take care of the correctness of the system as well as of its performance

[1] Projects IQ-Mobile (CNPq-CNR) and DACHIA (FAPERGS/IB-BMBF).
[2] Author partially sponsored by HP-Brasil/PUCRS agreement.
[3] Email: fpasini@inf.pucrs.br
[4] Email: fldotti@inf.pucrs.br

doi:10.1016/j.entcs.2007.03.018

(which is more commonly the focus of attention of parallel application builders). Assuring the correctness of the model of a parallel application, besides increasing the reliability in the results, can be important also economically since it reduces the computational time of a cluster (that it is an expensive resource) spent with tests.

Clusters are high performance parallel machines with an architecture based on a set of independent workstations, interconnected by a low latency communication network, used as execution platform for parallel applications. Each station has a local memory, to which only the local processor(s) has access. Most parallel application builders for distributed memory computers use MPI (*Message Passing Interface*) [15]. MPI is a standard message passing interface for applications using distributed memory computers, *e.g.* clusters. The MPI library is portable, and currently allows one to write programs in FORTRAN, C and C_{++} languages. It does not offer support for fault tolerance and assumes the existence of error free communications. According to this programming paradigm, all parallelism is explicit, *i.e*, the programmer is responsible for identifying the parallel parts of the application and explicitly mapping them to communicating processes or threads.

Several attempts to the development of tools for parallel programming use abstraction to reduce the level of complexity in the programming, helping the users to develop programs quickly and avoiding many programming errors. Most of these tools focus on the generation of code representing the various concurrent processes and their communication, and the user fills the generated code with sequential code fragments [12]. Some of these environments/tools offer support to analyze the application under development through simulations or performance tests.

Object-Based Graph Grammars (*OBGG*) [5] is a formal visual language suited to the specification of asynchronous distributed systems based on message passing. Parallelism is implicit, through the declaration of rules that define possible state changes, and non-determinism is supported. The few abstractions provided and the object-based style makes it relatively easy to learn. Due to the few constructions and its formal semantics, OBGG models can be model-checked via a translation to PROMELA [4] [11], the input language of the SPIN model-checker [8].

In this paper we propose the use of OBGG for the construction of parallel applications models and a translation step to generate C/MPI implementations from OBGG models. One of the arguments is to allow the specification of parallel applications using a language with implicit parallelism, increasing the level of abstraction offered to the designer. Another argument is to bring the possibility of model-checking the parallel applications under development as an OBGG model, which can then be directly translated to a C/MPI implementation. These ideas are illustrated with a simple parallel application which is defined in OBGG, model-checked for some properties, has its C/MPI code generated through the proposed translation, and is executed on a cluster, generating performance graphics which are discussed.

This document is organized as follows: Section 2 discusses aspects related to parallel programming and the generation of parallel programs; Section 3 presents the OBGG language as well as an example using this language. The properties checked

for this example are discussed in Section 3 as well. In Section 4 the translation from OBGG to C/MPI is presented, followed by performance results obtained for the example application. Finally, conclusion and future works are commented in Section 5.

2 Parallel Programming

Parallel programming differs in many aspects from the programming of a sequential system, the model which majority of developers has its first contact. The possible intercalation between the states of parallel processes are difficult to predict, making it hard to identify the problem sources with traditional debugging techniques used for sequential algorithms. Also, considering the differences between execution platforms and communication systems available, programming errors are very common.

Three main kinds of errors in parallel programs were identified [9]:

(i) Semantic errors: These errors are caused by a programmer's misconceptions about the parallel computing model being used and how to apply it to the problem at hand. For example, errors often arise from misunderstanding the semantics of memory sharing between processes and the semantics of the parallel programming language used;

(ii) Implementation errors: These errors occur because of the complexity added by parallel programs such as process launch, communication and synchronization. Typical errors include incorrectly packing/unpacking a message or creating a deadlock scenario;

(iii) Performance errors: These errors are caused by a lack of intuition or experience concerning the costs of parallelism and concurrency. For example, this class of errors includes executing fine grained program segments in parallel, or poor synchronization choices that restrict concurrency.

The first two types of errors usually result in programs that execute incorrectly, or yet make it impossible the generation of an executable. The third source often leads to programs that run correctly but have poor performance gains when compared to sequential solutions. The use of automatic code generation tools makes possible to prevent the first two problems, once it frees the user of the necessity to work with specific language aspects or communication libraries. The third error can be minimized through the specialization of the code generator, in order to guide the user to a more adequate model during the development.

Standards for parallel programming exist about two decades in many forms such as *design patterns* [14] [17], *skeletons* and *templates* [13], for example. *Skeletons* and *templates* provide a structure with parallel code, where the user introduces the application dependent code. *Design patterns* are basically descriptive structures for code fragment of common use.

The tool *Frameworks* [13] is one of the first works to use a graphical language to model parallel applications. The generated system consists in modules that communicate through RPC (Remote Procedure Call), being C the language used.

The offered constructors are based on data flowcharts, with symbols representing data division or grouping, scheduling and synchronization in the model. The vertices used in the model have a variable semantics, and edges represent communication channels.

Hence (Heterogeneous Network Computing Environment) [1] is a graphical environment for parallel programming that uses PVM [16] as communication platform. Vertices represent user provided sequential routines and/or data flow control between vertices. Unlabeled edges represent communication channels.

The tool *Code (Computation-Oriented Display Environment)* [2] provides code generation using MPI and PVM. As well as in the *HeNCE* tool, there are data processing and control vertices. Vertices can have more than one communication channel, and the edge label defines which channel is used for message passing.

DPnDP (Design Patterns and Distributed Processes) [14] uses parametrizable design patterns as an extensible library for parallel programming. An application is built using one or more design patterns, and they can be combined with other code fragments using also low level communication primitives. New design patterns can be incrementally added in the system, making it extensible.

The tool CO_2P_3S (*Correct Object-Oriented Pattern-based Parallel Programming System*) [17] generates Java code using design patterns. The modeling language used express the topology of the application. The tool has resources for monitoring the processing and communication between the processes, allowing the user to modify the model and search for a better load balance. Also, a pattern editor called $MetaCO_2P_3S$ is associated to this environment.

The graphical modeling languages used in these tools are elaborated aiming the user necessities, providing mechanisms for description of the desired system, but in many cases (such as in [13] and [17]) a formal definition of the language semantics is not presented.

3 Object Based Graph Grammars

Graph Grammars [7] offer a natural way to express complex situations, where the system under analysis can be described as a graph and the dynamic aspects can be captured by rules of a grammar. A graph grammar is composed by:

(i) *type graphs*, that represents the vertices and edges allowed in the system;
(ii) an *initial graph*, that represents the initial state of system and
(iii) a *set of rules* that describes the changes that can occur in a system.

In [5] a restricted form of graph grammars called OBGG is proposed. OBGG is an object-based language suited for specification of asynchronous message passing systems. In OBGG, a system consists of autonomous entities called *objects*. Objects have an internal state and communicate through asynchronous message passing. In OBGG system states are described as a graph where the objects and the messages are shaped as vertices. The object's attributes are arcs that leave the object and connect to other objects as the values of predefined data types, which also are

shaped as vertices. Figure 1 presents the model of an OBGG object. Graphically, an object has the notation of a rectangle containing its name (*Obj_N* in Figure 1), a class identifier (*i*) and the set of attributes. Instances of one same class held the same class identifier. Attributes of predefined data types are listed inside the rectangle (such as *atr_A*), and attributes that refer to other objects (*e.g. atr_B*) have the notation of edges that bind to other objects, or to a class identifier in case of rules and type graphs.

Attributes could be primitive data types (such as *integer*, *byte*, *float*) or Abstract Data Types (ADT). User defined ADT can be used in OBGG models under some restrictions: **(i)** the correctness of the user defined data types must be assured by the user; **(ii)** the operations on user defined data types must be atomic, *i.e*, they finish during the rule application. For instance, they can not leave any thread created or pending processing such as receiving a response [5].

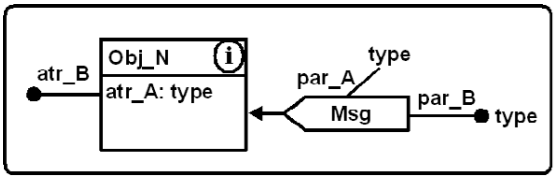

Figure 1. Model of an OBGG Object.

The object´s behavior corresponds to actions executed when receiving messages. Such actions can change the internal state of the object and/or cause the sending of messages to other objects and/or to itself.

Graphically, a message has the form of a polygon as message *Msg* in Figure 1. Messages can have only one object as destination and can have as parameters values of predefined types (*e.g. par_A*) or other objects (*e.g. par_B*). The destination of a message is given by an arrow and the message parameter(s) are given by lines that bind these parameters to the message´s polygon.

An object-based system may be comprised by objects belonging to different classes. In OBGG each class is defined using a type graph. The type graph of a class (see Figure 2) determines the class attributes and the messages (and message's parameters) that any instance of that class may receive and send.

The initial graph describes all objects, messages and attribute values that must exist in the desired initial model situation. All objects with their attributes and messages are instances of classes defined in type graphs. The objects instantiation can be done statically, from the initial graph, or dynamically, through the execution of rules that create new objects. As example of an initial graph see Figure 3.

Example 3.1 To illustrate the use of this formalism we adopt an OBGG model for a master-slave system implementing a Monte Carlo like method to calculate the value of π. The method consists of:

(i) Inscribe a circle in a square of side s (the radius of the circle is $s/2$);
(ii) Generate n random points inside the square;

[5] More concretely for this work, they may not call the MPI communication interface for any operation.

(iii) Be k the number of points inside the square that also are inside the circle;
(iv) Calculate $\pi \approx 4 \times k \div n$.

Observe that the more points generated, better the approximation. A good random algorithm generates a homogeneous distribution of points inside a domain. If a bad random generator were used, the value obtained would not be close enough to π. So, depending on the set of randomized points (*i.e.* depending on the random number generator used), there are possible model computations where the final value is not a good approximation of π. We will exemplify the model for one master and three slaves.

Figure 2 presents the type graph for classes *Master* and *Slave*. Messages that an object can receive are presented pointing to the object, and messages that the object generates are represented in right side of each type graph, pointing the object class identifier to which they are sent.

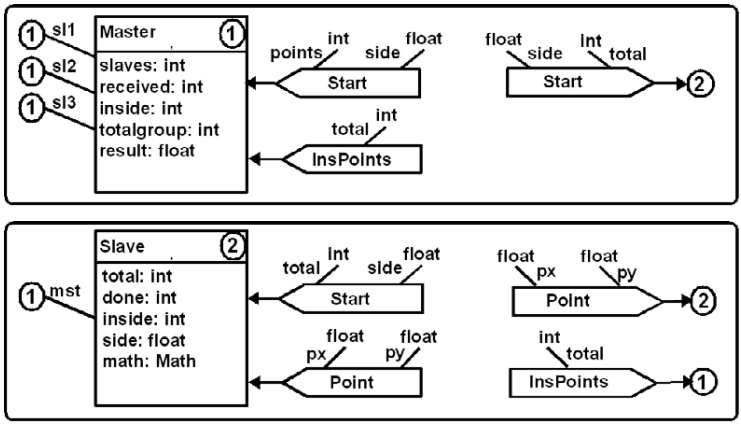

Figure 2. Type Graph for Master and Slave objects.

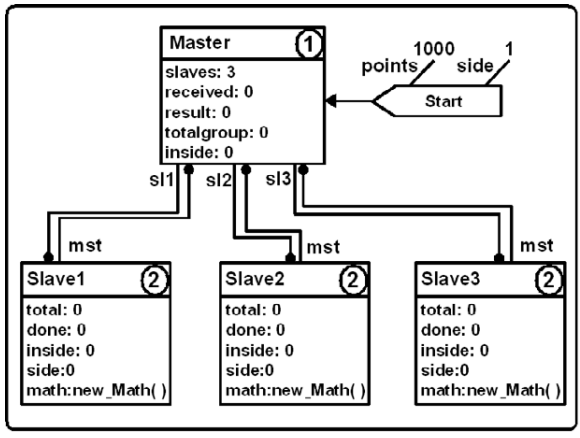

Figure 3. An Initial Graph.

In the type graph, references to other objects also link to class identifiers, as it is the case of the references $sl1$, $sl2$ and $sl3$ in object *Master* and *mst* in the object

Slave.

In OBGG, the computational system state – that involves the attribute's values, existing messages and processes in execution – is represented through a state graph, and the system functioning is described by a set of transformation rules applied on the current state of the system. Rules define the behavior of an object class. In a graph grammar, a rule $r : L \xrightarrow{C} R$ specifies one state change in the system that occurs as follows:

- All items that are on the left side of the rule L must be present in the current state of a system to make possible the rule application;
- C, if exists, is an equation over the attributes of its left side, and the rule can only be applied if this condition is true;
- All items that are mapped from L to R in the mapping r are preserved;
- All items that are not mapped from L to R are extinguished of the current state;
- All items that are in R and are not in L are added to the graph.

In OBGG, rules of a class present in the left-hand side one message being received. Each rule specifies the reaction of any object from that class to the reception of such message. In the right-hand side of the rule, this message will be consumed, attributes of the object can change values and new messages can be generated. All actions described in one same rule occur in an atomic step. The identification of a possible rule application is called *match*. Each rule describes the treatment of only one message. An object can send a message to another object only if exist at least one reference to the destination in the right-hand side of the rule. This reference can be an attribute of the object or a parameter of the incoming message.

OBGG rules allow to represent the concurrence and the non-determinism. The concurrence is represented by the possibility of applying more than one rule in one same situation, when these rules are not conflicting. Two rules are conflicting if they consume or modify the same items. Concurrence can occur involving different objects (external competition) or one same object (internal competition – when the same object can apply more than one rule in the same state of the graph). The non-determinism is represented in the choice of the rule for application. In case that more than one rule can be applied in a situation, one of these is non-deterministically selected to execute.

Example 3.2 Figure 4 and 5 respectively present the rules for classes *Master* and *Slave* previously defined. The notation $ObjectName\_RuleName$ will be used as reference for the presented rules.

Master_Start is the first rule applied in the system. It consumes the message *Start* and sends for each *Slave* a *Start* message, indicating in n how many points the *Slaves* must randomly generate, and in s the size of the square to be used. The rule *Master_Receive* presents the behavior of *Master* when receiving messages $InsPoints$, accumulating the total number of points inside the circle calculated by each *Slave*. Receiving the last message from a slave (rule *Master_Done*), $Master$ finally calculates the approximated value of π. The differentiation between the choice

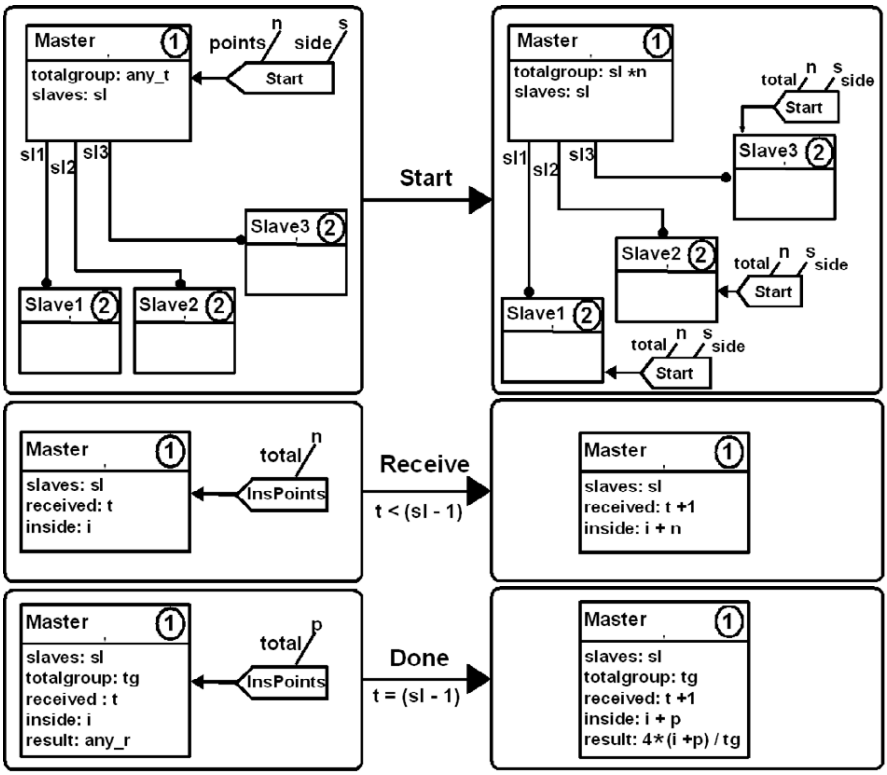

Figure 4. Rules for Master Object.

of *Master_Receive* and *Master_Done* is made by each rule's associated guard, that involves a comparison among the total number of participants (value *sl* for attribute *slaves*) and the number of received messages (value *t* for *received*).

For the rule *Slave_Start*, in presence of a message *Start*, *Slave* sets in attribute *total* the number of pairs to be generated, and fix in *side* the square side value. Also, it sends to itself the first coordinate pair as *Point* message's parameter. After that, *Slave* continuously consumes *Point*, computes if the point belongs or not to the circumference (attribute *inside*) and updates the total of generated messages (attribute *done*). When the last message is processed, it executes *Slave_Done* sending to *Master* the total number of points inside the circle.

The functions, $new\_Math()$, $ins(s, x, y)$ and $rand(s)$ are defined externally to OBGG model, using the following *Math* Abstract Data Type.

Abstract Data Type *Math*:
Operations:
Math new_Math ();
Initializes the ADT.
int ins (**float** s, **float** x, **float** y);
The function $ins(s, x, y)$ determines if the point (x, y) belongs or not to the circumference with radius $s/2$ inscribed in a square of side s. This function returns 1 (one) case the point is contained inside of the circle or returns 0 (zero)

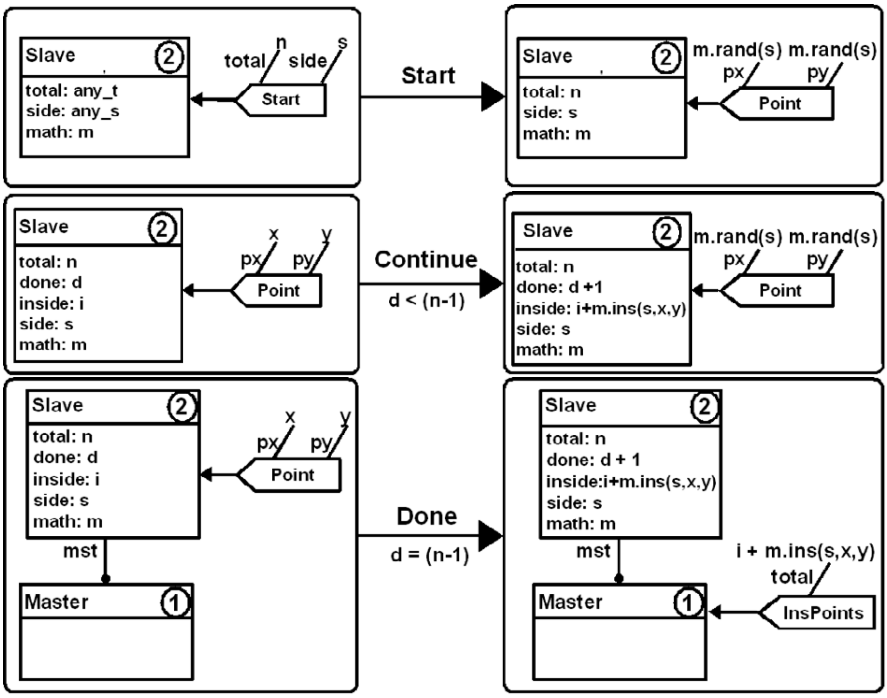

Figure 5. Rules for Slave Objects.

otherwise.

float rand (**float** limit);

Returns a number less or equal to *limit*.

int rand_eval (**float** s, **int** n);

Randomizes a set of n points inside a square of side s, returning the total of points inside a circle with radius $s/2$, inscribed in the square.

The corresponding implementation of the presented ADT must respect the restrictions previously mentioned. The function *rand_eval*() will be used in Section 4.7.

A serie of case studies have already been developed with OBGG such as mobile code applications [5]; the pull-based failure detector [6] and the readers and writers problem [11], among others. The few constructs make the language easy to learn and the object-based style eases reuse and extensibility.

3.1 Model Checking

To model-check an OBGG model, all the model elements (objects, messages, rules and parameters) are translated into PROMELA structures, allowing the use of SPIN [8] verifier. More details about the OBGG-PROMELA translation can be found in [4], including a formal proof of the semantics compatibility between original OBGG model and translated model. The SPIN model-checker supports the verification of properties described in Linear Temporal Logic (LTL). In [4] [11] it is proposed how to specify properties over OBGG models using LTL, taking into consideration

the definition of events as described in [10]. An OBGG development environment [3] allows one to edit OBGG models and translate the models into PROMELA in an automatic way. Moreover, the counter-examples generated by SPIN (when the model violates an LTL specification) can be translated to OBGG abstractions such that the counter-example can be interpreted using the OBGG model instead of the PROMELA generated one. This feature keeps the same level of abstraction for the developer.

In this section we discuss properties to be proven for the presented model. Due to the state space generated, the model will be verified using three *Slaves* and one *Master*, and every *Slave* will generate only four points. The following properties will be formalized in LTL and checked for the model:

(i) System termination: Eventually the model will generate a result, and after it, no more rules are applied;
(ii) Response: A *Slave* always answers a request;
(iii) Convergence: There exist (at least one) computations leading to an acceptable result.

In the following we use the SPIN syntax for LTL properties. Moreover, we use the notion of events as defined in [10]. An event is written $\uparrow Event$, and denotes the change of a state from an event not active to active, or ($!Event$ && $X\ Event$). Events are naturally used to denote the application of rules of OBGG models. [10] also introduces property patterns which can be used with the notion of events. The usage of such patterns is indicated for the properties discussed.

(i) System termination: *Master_Done* is the rule that calculates the final value for π. So, we have to show that this rule is applied and after its application no other rule is applied. To show that the rule is applied we have:

(1) $$<> (\uparrow Master\_Done)$$

Which results true. To show that after its application no other rule is applied in the model, we have:

(2) $$[]\ (\uparrow Master\_Done \rightarrow\ []\ !\uparrow Any\_Rule)$$

Where: $\uparrow Any\_Rule = (\uparrow Master\_Start\ ||\ \uparrow Master\_Receive\ ||\ \uparrow Master\_Done\ ||\ \uparrow Slave\_Start\ ||\ \uparrow Slave\_Continue\ ||\ \uparrow Slave\_Done)$.

This formula is a pattern of *Absence After an Event*, as presented in [10]. Here, absence of *Any_Event* after *Master_Done*, which results in true.

(ii) Response: To each application of rule *Slave_Start* there is always *Slave_Done* associated.

(3) $$<>\uparrow Slave\_Start \rightarrow <> (\uparrow Slave\_Start\ \&\&\ <>\uparrow Slave\_Done)$$

This formula is a pattern of *Existence (of Slave_Done) After an Event (Slave_Start)* [10] and results in true. Again, it is important to remember that this step assumes as correct the functions *ins*() and *rand*(). These functions where manually translated. In the PROMELA model, *rand*() always returns the same 0 value and *ins*() may return 1 or 0 non-deterministically. This abstraction was used to diminish the state space, and does not modify the meaning of the given system,

since the result of $ins()$ defines if the point belongs or not to the circumference.

(iii) Convergence: The convergence of the model can be shown with the result within an expected interval. However, since the model-checking process will consider all combinations of random points generated, there will be computations where all points fall outside the circle and therefore will not converge. On the other side, there will be computations that converge. It is possible to prove by model-checking that exist computations that converge. Using LTL, to prove that convergence is possible, we work trying to prove that the model will never converge. The result is a trace where the model converges. How often the method will converge is a matter of the quality of the random number generator employed in a real implementation. The LTL formula verified for this property is:

$$[] \; ! \; (\uparrow Master\_Done\_Pi) \tag{4}$$

Where: $\uparrow Master\_Done\_Pi = \uparrow Master\_Done \; \&\& \; (3,0 \leq result \leq 3,4)$

This formula is a pattern of *Globally Absence of an Event* [10] and, as expected, results in false. Concerning this aspect, a limitation of the verification tool is the lack of support of floating-point variables. For the sentence verification, the model was modified becoming $\pi \approx 1000 \times (4 \times k \div n)$ the operation used in rule $Master\_Done$, and testing the convergence of the result inside the interval $(3000, 3400)$. With a higher number of randomized points, this interval could be narrowed.

4 Translation OBGG - C/MPI

Table 1 presents a comparative between characteristics offered by GBBO and the ones found in an C/MPI environment.

	OBGG	**C/MPI**
Basic Unit	Object	Process
Unit Creation	Initial Graph	System Call
Object initialization	Dynamic, Static	Static
Comunication Method	Message Passing	Message Passing
Non-Determinism	Rules	Pseudo-Random Functions
Concurrence	Inside/Between Objects	Between Process/Threads

Table 1
Comparing OBGG and C/MPI.

An OBGG model is comprised by various concurrent objects interacting by message passing. It is therefore natural to map OBGG objects to processes and the message passing communication among OBGG objects to message passing among processes. The non-determinism offered by OBGG can be embedded in the way that incoming messages are treated by the process. Dynamic object creation can be

supported through dynamic process creation. The initial graph of an OBGG can be mapped to the initial (main) process of a parallel application.

In the following Sections we discuss in detail these aspects. Complementarily we discuss the problem of resource deallocation at the end of the execution as well as argue about the semantic preservation of the resulting implementation.

4.1 OBGG Process

A first step in the translation process is to map each object in a separate process (called *OBGG process*), making possible the parallel execution of rules by different processes in the system (parallelism between objects - external competition).

Another aspect to be considered is related to the number of messages supported by the model. An OBGG state graph holds (in thesis) an unbounded amount of messages. This characteristic is represented using a linked list (L_{in}) to store the messages already received and still not processed for each OBGG process. Therefore, the limit of the real implementation is the available memory.

Moreover, any of the messages can be consumed at any time by the object to which the message was sent. This determines the rule selection and execution mechanism. Once L_{in} stores all the input messages to the object, all *matches* message-rule at a given moment can be calculated. This is made as follows: for each input message M_i in L_{in}, consider all rules that consume that kind of message and are enabled to be applied (conditions hold), building a list R_i. Each pair (M_i, R_i), generated for each message, is stored in the list ($L_{matches}$) from which a pair (M_i, R) is randomly selected, where $R \in R_i$. The indicated message is removed from L_{in} and consumed, triggering the associated rule.

A rule application can modify the attribute values inside an OBGG process and/or generate one or more messages. In analogous way to what is made with the received messages, the generated messages are inserted in a list L_{out} after the application of the rule, leaving the sending operation for a posterior step. This choice was made to "free" the rule application of the send data overhead, thus allowing to pass immediately to processing the next message in L_{in}. The adoption of L_{out} brings also other advantages that will be argued in next sections.

From the above, the basic algorithm executing inside an OBGG process can be organized in eight main steps:

1. Receive a message M_i;
2. Include M_i in L_{in};
3. Evaluate L_{in} generating $L_{matches}$;
4. Randomly select a pair (M_i, R) from $L_{matches}$;
5. Exclude M_i from L_{in};
6. Apply rule R, consuming M_i and generating (or not) new messages;
7. Insert the generated messages in L_{out};
8. Send every message in L_{out}.

In the following sections, the elements adopted for the translation schema are presented in more details, making possible the transformation of this basic algorithm

in usable source code.

4.1.1 OBGG Process Code

According to the OBGG formalism, all operations of message passing - send or receive - are non-blocking. This means that no process sending/receiving messages will stop waiting for the communication to finish.

MPI offers blocking and non-blocking communication primitives, however the non-blocking call semantic is not trivial[6]. Thus, blocking primitives were adopted. In order to break the synchrony among sending and receiving processes, three distinct threads to receive, process and send messages are used in an OBGG process. These threads will be referred respectively as *Receiver*, *Evaluate* and *Sender*.

From the basic algorithm previously presented, *Receiver* is associated to the execution of steps 1 and 2, *Evaluate* performs steps 3 to 7 and *Sender* executes step 8. Such strategy makes possible the implementation of a more modular program, making easier the understanding, legibility and maintenance of the generated code.

Complementarily, the attributes of OBGG objects are mapped to local variables of the corresponding OBGG process. Basic data types available in the destination language (C) can be used in OBGG specifications. User defined data types that follow the restrictions already discussed can be used as well.

4.2 OBGG Messages

OBGG messages are translated to MPI messages using C data structures. These structures are composed by primitive data types such as *char*, *int* and *float*. MPI messages are constructed field-by-field, using packing primitives offered by the library. Such messages must be unpacked in the same order by the generated *Receiver* code.

Inside of the data structure of generated by the translation, the first field contains identification of the type of the corresponding message. The other fields (when existing) are related to each present parameter in the message. In this presented translation schema, the generated system uses only one type of data structure for all messages, which is a superset of all the allowed parameters of messages in the system.

Another important restriction adopted is the limitation of the message parameters to be based only in basic data types supported by the primitives of packing/unpacking. Thus, during the modeling of an OBGG system, the sending of complex structures of data (*i.e.* lists) are not allowed, unless the user provides operations to transform these structures into primitive data types before send, and also the reverse operation to be used in the correspondent rule after receive.

[6] When using non-blocking primitives, the user issues a send request and has to explicitly program the sending process to repeatedly test with the communication platform if the send operation has finished.

4.3 OBGG Rules

Once the matching algorithm identifies the rule to be executed, the OBGG process performs the following steps:

(i) Consume the message (removes of M_i passing it to a *buffer* inside the code of the rule);
(ii) Update the OBGG process internal attributes, if necessary;
(iii) Generate the messages, inserting them in a temporary list [7] L_{temp}. Messages addressed to the process that is applying the rule are inserted directly in L_{in}.
(iv) Append L_{temp} to the end of L_{out}.

4.4 Initial Graph

The initial graph defines all instances which are active from the beginning of the system as well as the value of every attribute associated to these instances, including the relation among objects. Another function of the initial graph is to define the initial messages to each object.

The definition of the object's attribute values is translated as a set of attributions, that are inserted in the code of each OBGG process immediately before the point where the threads *Evaluate*, *Receive* and *Sender* are started. These values are fixed before receive any message, so the mechanism of *matching* will not find the object in an inconsistent state. After the code generation and compilation, the obtained binary code is copied to cluster nodes, and the processes are initiated through a system call.

The generation and sending of all messages is done by a special process called *INIT*. The sending order to be followed is defined non-deterministically. After sent all messages, INIT acts as a coordinator among the remaining processes, as will be explained in the next Section.

The translation currently does not offer support to dynamic creation of processes, due to limitations in the MPI library.

4.5 System Termination

In OBGG the termination of objects is not represented explicitly. A system stops processing when there are no more matches possible. However, explicit process termination is necessary in real systems to, for example, deallocate the resources of a cluster. Therefore, in order to consider the system terminated and thus be able to deallocate the resources, we have to somehow detect that there are no more matches possible in the distributed processes representing the OBGG objects. When such a situation is detected, then the various processes are signalized and terminate. The identification of the distributed termination (no more matches) is coordinated by the INIT process, after having performed the initialization procedures.

[7] Avoiding to successively lock and unlock L_{out} to assure data consistency. The use of one temporary list makes L_{out} more available to *Sender*.

In a totally asynchronous setting, this problem is analogous to a voting protocol. However, in our translated model we can assume certain communication characteristics and avoid a high amount of messages. The distributed protocol for detecting the termination is described bellow. It uses three control messages: R, B and C. The behavior of each participant process and the behavior of the INIT (coordinator) is as follows.

The participants behavior:

- Thread *Evaluate* is responsible for searching the input buffer for messages and identifying matches for them. If this thread does not identify any match for all the messages in the buffer or the buffer is empty, then it blocks until a new message is received. Once *Evaluate* is blocked for a certain amount of time, a control message (B), signaling the blocked state, with destination INIT, is appended to L_{out} and will be eventually sent by *Sender*.
- After *Evaluate* blocks, *Sender* periodically confirms the OBGG process blocked state (with a C message) to INIT.
- When the OBGG process receives a message, *Evaluate* tests for matchings again and, if successful, immediately signals to INIT the reactivation (using a R message). If no matches are found, the OBGG process will not go to confirm its blocked state (C) before informing at least one time a blocked state again.

Behavior of the INIT (coordinator) process:

- In the view kept by INIT, each process can be in one of three states: Running (R), Blocked (B) our Confirmed Blocked (C).
- At the moment where the coordinator first detects one view where *all the processes* are in state C, then there exist a possible termination situation.
- The coordinator then awaits for the immediate confirmation of each participant (C). Only after all the processes have confirmed their state C for one second time, and in previous view all the processes also were in state C, then the termination is characterized.
- All other configurations of the view of the INIT process are non valid for termination.
- When INIT detects the above termination situation, it sends to all participants a control message $Killmesg$, informing that the processes can finish and deallocate the resources.

This protocol is based on the fact that all communication is synchronous, *i.e.* the order in which each other process receives the control messages from a given process is assured to be the sending order.

This protocol should assure that no false termination is detected, *i.e.* the coordinator will not have all processes twice in (C) in two subsequent views if at least one process is still in activity. To better understand the protocol, suppose all processes in (C), but one in state (R). Now suppose the running process sends a data message (M) to one process, then blocks (B) with INIT and after a while confirms

(C) the blocked state. In this moment, INIT has all processes in (C) and, for a false termination detection, INIT would need once again a confirmation from every process. Since the message (M) is assured to be delivered to the other process *before* the sending process sends (B) and (C) to INIT, the other process will not confirm (C) to INIT after this point, but send a (R) or (B) message, depending on the matching or not (respectively) of message (M). Therefore, the (false) termination will not be detected.

4.6 Considerations on the Translation

The generated code should behave according to the OBGG semantic. Assuring this behavior would require a formal proof that the semantic of the generated code is equivalent to the semantic of the respective OBGG model. Such a formal proof depend on the formal semantics of the programming language and the communication platform, which are not available. Therefore, in this section we argue that the main characteristics of an OBGG model are preserved in the generated source code. The main aspects discussed are:

(i) Concurrency: The concurrency between objects is represented through the parallel execution of the respective processes representing objects.

Considering intra-object concurrency, in OBGG non-conflicting rules may be applied in parallel. Our current mapping does not represent the possibility of simultaneous application of non-conflicting rules. This would impose too much overhead since the state of the attributes at the left-hand side would have to be copied for each active rule, and threads would have to be dynamically created. However, the implementation does represent all possible reachable states of the OBGG object. Since the algorithm that chooses a rule for execution does it non-deterministically, the generated system represents all potential sequences of rule applications and all possible states starting from the initial one. This is the same approach followed by the OBGG-PROMELA translation discussed in [4].

(ii) Non-determinism: As stated in the algorithm describing the steps performed by an object, the calculation of matches consider all messages in the input buffer, and, for every message, all possible rules (a message may be handled by more than one rule). From all the enabled rule applications a random choice is taken. This random choice among all enabled rules represents the non-deterministic behavior of OBGG models.

(iii) Non-ordered communication: with the same arguments as above (non-determinism), there is no order associated to the consumption of messages.

(iv) Asynchronous communication: the rule evaluation activity of one object does not stop for sending messages. Message transmission is performed on a separate thread.

(v) Encapsulation: the attributes representing an object become internal variables of a process. The only way to change those attributes is through the application of an object's rule.

(vi) Reactivity: A rule is enabled only if a particular message is received (and

specific conditions on attributes of the object and parameters of the message are fulfilled). Therefore, changes in the internal state of the object take place only if a message is processed.

4.7 Performance Measurement

In this section we present performance measurements for the parallel application generated from the model presented in Section 3. The parallel application was executed in the CPAD [8] cluster, which is comprised by 16 dual PentiumIII stations (550MHz, 256 MB-Ram) interconnected via a Fast-Ethernet network. For the model execution was used one cluster node per process.

The graphic of the Figure 6.a compares the execution times obtained for the generated application with a manually generated program (called *base program*), varying the number of *Slave* processes and taking the total execution time of the system. The base program has a master-slave logical topology and the communication between master and slave is analogous to the model considered in Section 3. Also, the function *rand_eval*() is used in base's *Slave* processes to generate and evaluate the points.

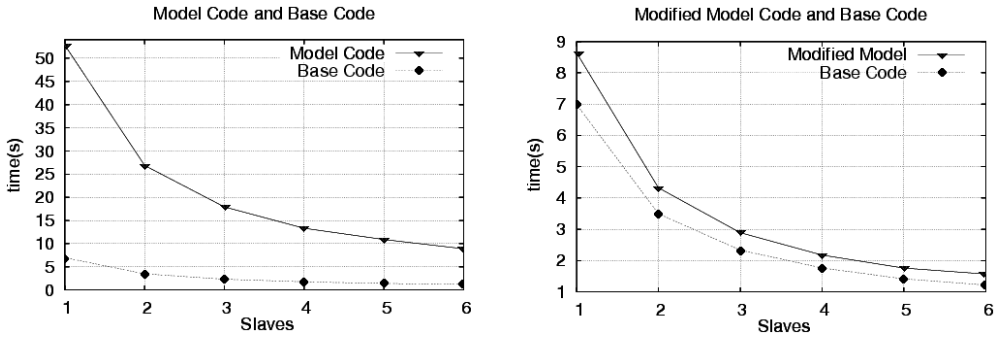

Figure 6. Performance for Original Pi Model, Base Algorithm and Modified Pi Model.

It can be observed that the execution times obtained by the generated code are around eight times the obtained from base program. This happens because in the OBGG model of *Slave*, in rule *Slave_Continue*, there is a message sent to the *Slave* itself which is used to model a sequential behavior. If we model this sequential behavior as part of the functionality of an abstract data type, avoiding message passing, then the performance of the generated application compared to the program base becomes as in Figure 6.b. Figure 7 presents the modified rule for *Slave* used in this execution. Note that now all the points generation and evaluation are made by the function *rand_eval*().

Although the performance of the generated application is lower than the base program, one should observe that the scalability of the model is the same as the base program; *i.e.* adding more processes does not lead to unexpected performance penalties.

[8] CPAD - Centro de Pesquisa em Alto Desempenho or High Performance Research Center - is a cooperation among PUCRS and HP-Brasil.

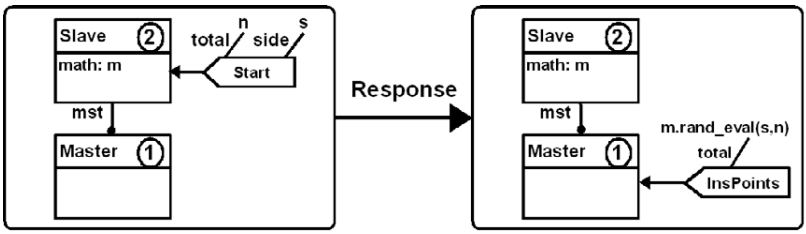

Figure 7. Modified Model for Slave.

There are various reasons for the performance difference. While on the base program only one MPI call is needed to transmit a message (that consists only of an integer value), in the generated application there are many MPI calls involved: one for packing every parameter, one for sending, and finally one unpacking calls for every parameter. Also, due to the OBGG behavior, a (linked) list of messages has to be manipulated, and the matching is calculated over all messages in this input list each time a rule has to be chosen for application. Moreover, it has to be considered that this performance penalty is due to the more abstract modeling formalism which offers non-determinism and implicit parallelism.

5 Conclusions and Future Work

This work proposed the use of OBGG to model parallel applications as well as, from the OBGG model, generate the corresponding C code for MPI platforms. This translation from OBGG to C/MPI followed, in many aspects, the same ideas of an already existing translation from OBGG to PROMELA [4], which is also based on processes and communication channels.

As described in this paper, the translation step is straightforward and can be automated. An implementation of this translation is a current effort. The result should be integrated in an existing environment for the edition of OBGG models [3].

The performance of the translated model in Section 4.7 show the impact of offering the abstraction provided by OBGG. Some aspects of the translation are under investigation in order to allow a better performance. For instance, in the current translation only one MPI message type is defined for a whole model. In terms of message parameters, this MPI message is the superset of all OBGG messages, leading to extra processing and communication overhead. A next optimization is the generation of various MPI message types, one for each message type of the OBGG model.

A further topic to invest in optimization is in the internal algorithm of each object. More specifically, the *matching* calculation mechanism could profit by indexation techniques in order to identify candidate rules for a specific message in the input buffer.

Finally, more case studies should be performed in order to show the suitability of OBGG to parallel applications as well as to analyze the behavior of the generated applications and possibly enhance the translation here proposed.

References

[1] Baguelin, A., J. Dongarra, G. Giest, R. Manchek and V. Sunderam, *Graphical development tools for network-based concurrent computing*, in: *Supercomputing'91*, 1991, pp. 435–444.

[2] Browne, J. C., S. I. Hyder, J. Dongarra, K. Moore and P. Newton, *Visual programming and debugging for parallel computing*, IEEE Parallel Distrib. Technol. **3** (1995), pp. 75–83.

[3] Dotti, F. L., L. M. Duarte, L. Foss, L. Ribeiro, D. Russi, and O. M. Santos, *An environment for the development of concurrent object-based applications,* Electronic Notes in Theoretical Computer Science **127** (2004), pp 3–13.

[4] Dotti, F. L., L. Foss, L. Ribeiro and O. M. Santos, *Verification of object-based distributed systems*, in: *6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Lecture Notes in Computer Science (LNCS) **2884** (2003), pp. 261–275.

[5] Dotti, F. L. and L. Ribeiro, *Specification of mobile code systems using graph grammars*, in: *4th International Conference on Formal Methods for Open Object-Based Distributed Systems*, IFIP Conference Proceedings **177** (2000), pp. 45–63.

[6] Dotti, F. L., L. Ribeiro and O. M. Santos, *Specification and analysis of fault behaviours using graph grammars*, in: *2nd International Workshop on Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science (LNCS) **3062** (2003), pp. 120–133.

[7] Ehrig, H., *Introduction to the algebraic theory of graph grammars*, in: V. Claus, H. Ehrig and G. Rozenberg, editors, *1st Graph Grammar Workshop*, Lecture Notes in Computer Science (LNCS) **73** (1979), pp. 1–69.

[8] Holzmann, G. J., *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), pp. 279–295.

[9] Iglinski, P., N. Kazouris, S. MacDonald, D. Novillo, I. Parsons, J. Schaeffer, D. Szafron and D. Woloschuk, *Using a template-based parallel programming environment to eliminate errors*, in: *Proceedings of the High Performance Computing Symposium* (HPCS'96), University of Carleton, Ottawa (June 1996).

[10] Paun, D. O., and M. Chechik, *On closure under stuttering,* Formal Aspects of Computing **14** (2003), 342–368.

[11] Santos, O. M., F. L. Dotti and L. Ribeiro, *Verifying object-based graph grammars.*, Electronic Notes in Theoretical Computer Science (ENTCS) **109**, 2004, pp. 125–136.

[12] Schaeffer, J., D. Szafron, G. Lobe and I. Parsons, *The enterprise model for developing distributed applications*, IEEE Parallel and Distributed Technology **1** (1993), pp. 85–96.

[13] Singh, A., J. Schaeffer and M. Green, *A template-based approach to the generation of distributed applications using a network of workstations*, IEEE Transactions on Parallel and Distributed Systems **2** (1991), pp. 52–67.

[14] Siu, S., M. D. Simone, D. Goswami and A. Singh, *Design patterns for parallel programming*, in *International Conference on Parallel and Distributed Processing Techniques and Applications* (1996), pp. 230–240.

[15] Snir, M., S. W. Otto, D. W. Walker, J. Dongarra and S. Huss-Lederman, "MPI: The Complete Reference´´, MIT Press, Cambridge, MA, USA, 1995.

[16] Sunderam, V. S., *PVM: a framework for parallel distributed computing*, Concurrency, Practice and Experience **2** (1990), pp. 315–340.

[17] Tan, K., D. Szafron, J. Schaeffer, J. Anvik and S. MacDonald, *Using generative design patterns to generate parallel code for a distributed memory environment*, in: *PPoPP '03: 9nd ACM Symposium on Principles and Practice of Parallel Programming (SIGPLAN)* (2003), pp. 203–215.