Verification of Distributed Object-Based Systems*

Fernando L. Dotti[2], Luciana Foss[1], Leila Ribeiro[1], and Osmar M. dos Santos[2]

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

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

Abstract. Distributed systems for open environments, like the Internet, are becoming more frequent and important. However, it is difficult to assure that such systems have the required functional properties. In this paper we use a visual formal specification language, called Object-Based Graph Grammars (OBGG), to specify asynchronous distributed systems. After discussing the main concepts of OBGG, we propose an approach for the verification of OBGG specifications using model checking. This approach consists on the translation of OBGG specifications into PROMELA (PROcess/PROtocol MEta LAnguage), which is the input language of the SPIN model checker. The approach we use for verification allows one to write properties based on the OBGG specification instead of on the generated PROMELA model.

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 open systems, like: massive geographical distribution; high dynamics (appearance of new nodes and services); no global control; faults; lack of security; and high heterogeneity. It is therefore 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.

We have developed a formal specification language [7], called Object-Based Graph Grammars (OBGG), suitable for the specification of asynchronous distributed systems. Currently, models defined in this formal specification language can be analyzed through simulation [4] [5]. Moreover, starting from a defined model we can generate code for execution in a real environment, following a straightforward mapping from an OBGG specification to the Java programming language [5]. In order to deal with open environments, we have worked on an approach to consider classical failure models still during the specification phase,

* This work is partially supported by HP Brasil - PUCRS agreement CASCO (24° TA.), ForMOS (FAPERGS/CNPq), PLATUS (CNPq), IQ-Mobile II (CNPq/CNR) and DACHIA (FAPERGS/IB-BMBF) Research Projects.

allowing one to reason about a given model in the presence of a selected failure [8]. Investigations about the complexity of verifying properties of OBGG specifications considering message passing as the main operation were done in [13]. By using the methods and tools mentioned above we have defined a framework to assist the development of distributed systems. The innovative aspect of this framework is the use of the same formal specification language (OBGG) as the underlying unifying formalism [6].

In this paper we focus on the verification of object-based distributed systems. More specifically, we add to our framework the possibility of model checking distributed systems written according to the OBGG formalism. To achieve that, we propose a mapping from OBGG specifications to PROMELA (PROcess/PROtocol MEta LAnguage, which is the input language of the SPIN model checker), showing the semantic compatibility of the original OBGG specification with the PROMELA generated model. After that, we show how to specify properties using LTL (Linear Temporal Logic) over the OBGG specification. An important aspect is that the user does not need to know the generated PROMELA model to specify properties.

This paper is organized as follows: Section 2 presents the formal specification language OBGG together with an example (modeling the dining philosophers problem); Section 3 shortly introduces PROMELA; in Section 4 the translation from OBGG to a PROMELA model is presented together with a discussion of its semantic compatibility; in Section 5 we present our approach for the verification of OBGG models using SPIN and then conclude in Section 6.

2 The Specification Language OBGG

Graphs are a very natural means to explain complex situations on an intuitive level. Graph rules may complementary be used to capture the dynamical aspects of systems. The resulting notion of graph grammars generalizes Chomsky grammar from strings to graphs [9,15]. The basic concepts behind the graph grammars specification formalism are:

- states are represented by *graphs*;
- possible state changes are modeled by *rules*, where the left- and right-hand sides are graphs; each rule may *delete, preserve and create* vertices and edges;
- a rule have *read access* to items that are preserved by this rule, and *write access* to items that are deleted/changed by this rule;
- for a rule to be *enabled*, a match must be found, that is, an image of the left-hand side of a rule must be found in the current state;
- an enabled rule may be *applied*, and this is done by removing from the current graph the elements that are deleted by the rule and inserting the ones created by this rule;
- two (or more) enabled rules are in *conflict* if their matches need write access to common items;
- many rules may be applied in *parallel*, as long as they do not have write access to the same items of the state (even the same rule may be applied in parallel with itself, using different matches).

Here we will use graph grammars as a specification formalism for concurrent systems. The construction of such systems will be done componentwise [14]: each component (called entity) is specified as a graph grammar; then, a model of the whole system is constructed by composing instances of the specified components (this model is itself a graph grammar). Instead of using general graph grammars for the specification of the components, we will use Object-Based Graph Grammars (OBGG) [7]. This choice has two advantages: on the practical side, the specifications are done in an object-based style that is quite familiar to most of the users, and therefore are easy to construct, understand and consequently use as a basis for implementation; on the theoretical side, the restrictions guarantee that the semantics is compositional, reduce the complexity of matching (allowing an efficient implementation of the simulation tool), as well as ease the analysis of the grammar. Basically, we impose restrictions on the kinds of graphs that are used and on the kind of behaviors that rules may specify.

Each graph in an OBGG is composed of instances of the vertices and edges shown in Figure 1. These vertices represent entities and elements of abstract data types. Elements of abstract data types are allowed as attributes of entities and/or parameters of messages (in the visual representation, such attributes are drawn inside the entity). Messages are modeled as (hyper)arcs which have one entity as target and as sources the message parameters (that may be references to other entities or values).

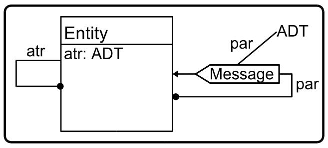


Fig. 1: Object-Based Type Graph

For each entity, a graph containing information about all its attributes, relationships to other entities, and messages sent/received by this entity is built. This graph, called *type graph*, is an instantiation of the object-based type graph described above. All rules that describe the behavior of this entity may only refer to items defined in this type graph. Instances of entities are called *objects*.

A rule describes the reaction of objects to the receipt of a message. A rule of an OBGG must delete exactly one message (trigger of the rule), may create new messages to all objects involved in the rule, as well as change the values of attributes of the object to which the rule belongs. A rule shall not delete or create attributes, only change their values. At the right-side of a rule, new objects may appear (instances of entities can be dynamically created). Besides, a rule may have a condition, which is an equation over the attributes of its left- and right-hand sides. A rule can only be applied if this condition is true.

An OBGG consists of a type graph, an initial graph and a set of rules. The type graph is actually the description of the (graphical) types that will be used in this grammar (it specifies the kinds of objects, messages, attributes and parameters that are possible – like the structural part of a class description).

The initial graph specifies the start state of the system. Within the specification of an entity, this state may be specified abstractly (for example, using variables instead of values, when desired), and will only become concrete when we build a model containing instances of all entities (objects) involved in this system. As described above, the rules specify how the instances of an entity will react to the messages they receive.

According to the graph grammars formalism, the computations of a graph grammar are based on applications of rules to graphs. Rules may be applied sequentially or in parallel. Each state of a computation of an OBGG is a graph that contains instances of entities (with concrete values for their attributes) and messages to be treated. In each execution state, several rules (of the same or different entities) may be enabled, and are therefore candidates for execution. 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, we need to choose a set of rules that is consistent, i. e., a set in which no two or more rules have write access to (delete) the same resources. Due to the restrictions imposed in OBGG, write-access conflicts can only occur among rules of the same entity. When such a conflict occurs, one of the rules is (non-deterministically) chosen to be applied. This semantics is implemented in the simulation tool PLATUS [4,5].

We can describe this semantics as a labeled transition system (LTS) in which the states are the reachable graphs, and each transition represents a rule application, having as label the name of the rule that was applied. Actually, there are many possible choices for the labels of transitions, ranging from very simple ones, like just the name of the rule, to more complex ones, containing also the identity of the object and message involved in the rule application, and even names and values of attributes changed by this computation step. As these labels are the events that can be observed in the semantics, if we have richer labels, we will be able to describe more complex properties over the system represented by this transition system. However, the number of types of events (labels) has a direct impact in the size of the state space of the (translated PROMELA) system, and the risk of state explosion during verification is quite high if we have many different labels. The choice of just having rule names as labels is a trade off between expressiveness for describing properties and the limitations of verification tools.

2.1 The Dining Philosophers Problem

In this Section we model the dining philosophers problem using OBGG. The type graph, and rules for the objects that compose the specification are presented in Fig. 2. Using the same type graph and set of rules, we present two different models for this problem, given by two different initial graphs, describing a symmetric and an asymmetric solution. We use these solutions in Section 5 in order to illustrate our approach for the verification of properties.

Traditionally the dining philosophers 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 its left and right forks. After eating a philosopher release both left and right forks and starts thinking again.

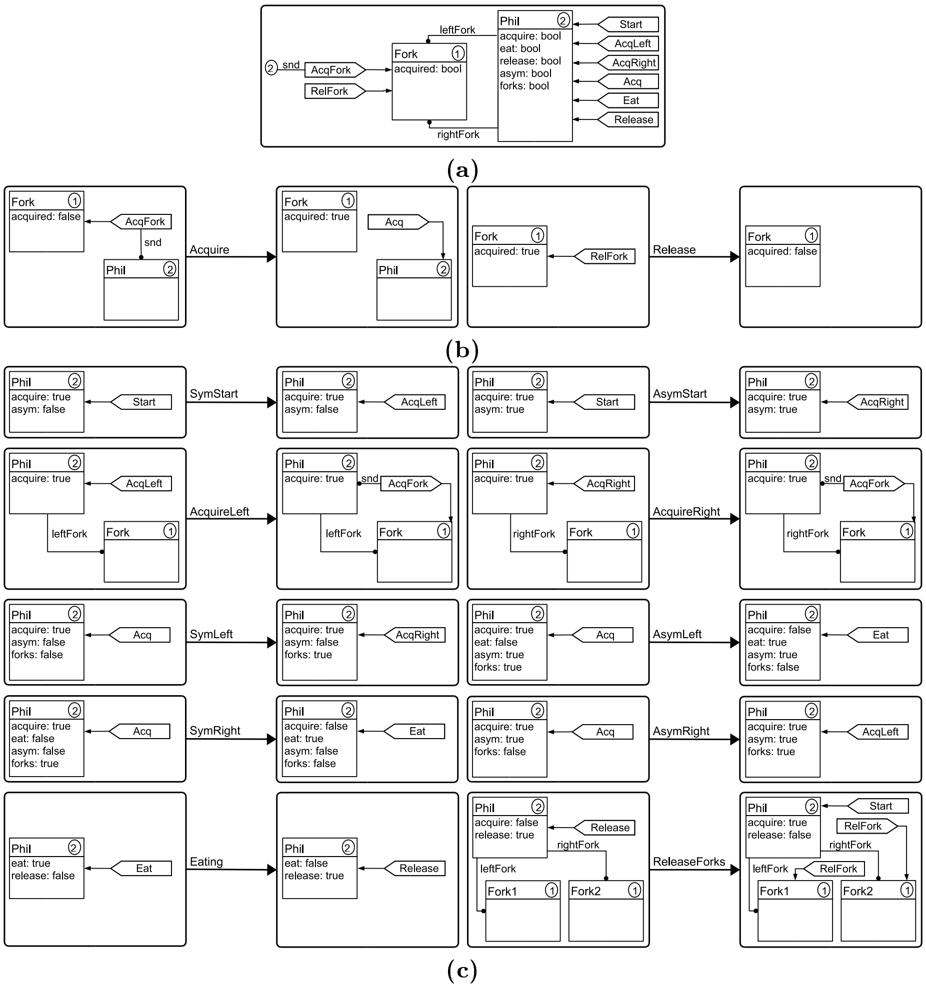


Fig. 2: Type Graph (a) and Rules of *Fork* (b) and *Phil* (c) Objects

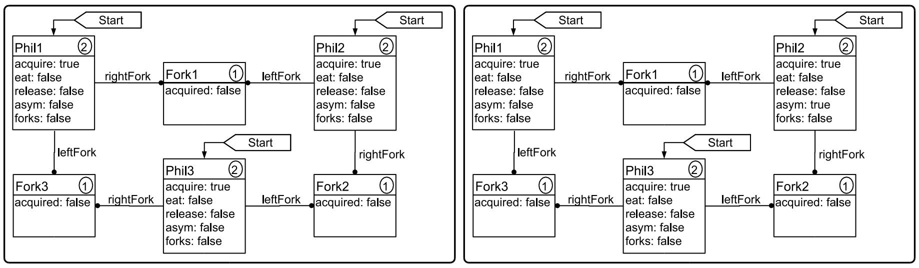
In OBGG we model the problem with two entities: *Fork* and *Phil*. The messages that objects of these entities can receive and their attributes are described in Fig. 2 (a)[3]. The *Fork* entity represents the forks and is composed of a boolean attribute (*acquired*) that determines if the fork is currently in use by a philosopher (*acquired* true) or not (*acquired* false). The *Phil* entity represents

[3] The numbers inside the circles are used to indicate the type of each entity. They are defined in the type graph and are used as a type information for the instances that appear in the rules and state graphs.

the philosophers and is composed of 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 have two references to *Fork* objects that are its left (*leftFork*) and right (*rightFork*) forks.

The rules for the *Fork* objects are shown in Fig. 2 (b), and the rules for the *Phil* objects are presented in Fig. 2 (c). The behavior of the specification is as follows. A philosopher starts 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 (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 release his forks and starts all over again.

In Fig. 3 (a) we show an initial graph for a symmetric solution of the problem. This solution is symmetric because all the philosopher has his *asym* attribute set to false, meaning that all of them will try to acquire the left fork first. Fig. 3 (b) illustrates 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.



(a) **(b)**

Fig. 3: Initial Graph for Symmetric (a) and Asymmetric (b) Solutions

3 Process/Protocol MEta LAnguage

PROMELA [1] is a process based language, being used by the SPIN model checker [10] for the specification of models. In SPIN, from a PROMELA specification it is possible to define properties using LTL (Linear Temporal Logic) formulas, and verify if the formulas are true for a given specification.

The language has a *C-like* syntax and constructs for receiving and sending messages similar to the ones found in the specification language CSP (Communication Sequential Processes). Processes in PROMELA can be created statically or dynamically (*proctype* keyword). There is a special process, called *init*, used to initialize a specification. Processes can exchange information through message channels (*chan* keyword) or global variables (variables declared outside the scope of the processes). The message channels can be synchronous (the buffer of the message channel has 0 messages) or asynchronous (the buffer of the message channel can have N messages, being $N > 0$). Message channels are typed, in the sense that one has to explicitly declare the types of variables a channel might receive. Besides, PROMELA offers several functions used to check, for example, if a channel is not full (*nfull(channel)*), how many messages a channel has in its buffer (*len(channel)*), and others [1].

In PROMELA, non-determinism, i.e. the existence of more than one possible execution path, is modeled in condition (*if ... fi*) or repetition (*do ... od*) structures. The entries of condition and repetition structures are composed of guarded commands. Once the condition of a guarded command is not satisfied, the entry is blocked, possibly blocking the process that contains it. This blocking occurs until the condition is satisfied. In condition and repetition structures, non-determinism occurs when several entries have their conditions satisfied. In this case, one of the possible paths is chosen in a non-deterministic way. It is possible to define atomic structures (*atomic* { ... }) for a specification, i.e., a sequence of statements that must be executed without interleaving with the execution of statements of other processes. However, if there are guarded commands inside an atomic structure and they are not satisfied, the structure will lose its atomicity characteristic and will interleave its statements with other processes. We can define enumeration types in PROMELA (*mtype* keyword). The language provides a *goto* statement that enables a developer to jump into the body of a process. Finally, one can insert assertions in a PROMELA specification. An assertion statement evaluates an expression (*assert(expression)*) to true or false, each time the statement is executed. If the expression evaluates to false, an error is generated and the verification procedure stops.

We can describe the operational semantics of PROMELA by a labeled transition system (LTS). This semantics can be found in [16]. A state of a PROMELA program consists of a fragment of this program (its still unexecuted code), a function that relates each global variable (or channel) name with its value and a function that relates each channel identifier with its current value (length of channel buffer and values stored in it). Moreover, the state stores information about definitions of processes, about active processes and about which processes are currently executing atomic blocks. Each *proctype* statement defines a process. A process definition consists of the process body and a parameter function that defines the name and the type of its parameters. A process instantiation (active process) stores the following information: the body of the process and its current unexecuted code fragment, the values of all local variables (or channels) and a continuation stack that stores program fragments (used for do statements).

Each transition of a PROMELA LTS has as label some statement of the language. These transitions are defined by the SOS-rules of [16], which describe the behavior of each statement of PROMELA. The initial state for every LTS is $(\pi, G_\perp, C_\perp, pdef_\perp, act_\perp, \perp)$, where π is a program body; $G_\perp$, $C_\perp$, $pdef_\perp$ and $act_\perp$ are the mappings of global variables, channels, definitions and instantiations of process, respectively (all these functions are undefined for all values of their domains); and $\perp$ is a process identifier that is executing atomically.

4 Translation of OBGG into PROMELA

The translation of OBGG into PROMELA defines how the abstractions of OBGG are mapped to PROMELA, in a way that the semantics of OBGG is preserved. In Section 4.1 we present how the abstractions of object, message, rule and initial graph present in OBGG are translated to PROMELA, and in Section 4.2 we discuss how semantic compatibility is achieved. We do not show neither the concrete syntax of this mapping nor the proofs of compatibility due to space constraints.

4.1 Syntactical Mapping of OBGG into PROMELA

Objects and Messages Objects in OBGG are translated into processes in PROMELA (we call such processes *object process(es)*). Attributes of an object are mapped to variables, passed as arguments in the definition of an *object process*. For verification purposes, attributes of OBGG objects are restricted to the types supported in PROMELA. A reference to an OBGG object is mapped to a PROMELA channel. Messages in OBGG are translated into messages in PROMELA. The receipt of messages is done through an asynchronous channel, called *object process channel*, that is also passed as an argument in the definition of an *object process*. The *object process channel* is typed according to: the name of a message (an *mtype* in PROMELA, composed of the name of all messages in the system type graph), and the parameters of all messages that an object can receive. The parameters of all messages that an object can receive become variables declared inside an *object process*. The dynamic creation of objects corresponds to the dynamic creation of processes and their associated channels in PROMELA.

Concurrency among objects is naturally preserved by the concurrency between *object processes*. Nevertheless, in OBGG it is possible to have intra-object concurrency. That is, when an object has several non-conflicting messages to be processed we may have the parallel reception of these messages. We face two main problems when translating this feature to PROMELA.

The first problem is due to the way messages are received in OBGG, i.e. messages are not stored in a specific place, they are simply connected to the object in the system state graph. Since we translate OBGG messages to messages sent through PROMELA channels, we require the user to set a buffer size for the *object process channel* of *object processes*. The problem occurs when a small size is set, introducing possible points of synchrony in the model (that do not exist in the original OBGG model because an object may receive an unbounded number of messages at each moment). We handled this problem by inserting assertions that, just before sending a message in the translated model, evaluate an expression to determine if the destination channel is not full. Thus, when verifying a model with a small buffer size, an error is generated when the *object process channel* is full, requiring the user to increase the buffer size.

The second problem is related to the non-deterministic reception of messages in OBGG. Because PROMELA channels work in a first-in first-out manner, we have to introduce a structure that non-deterministically receive messages to be processed. This is done by creating an internal buffer in every *object process* that is responsable for receiving a message in a non-deterministic way.

Thus, we define the generic behavior of an *object process* as follows: **(i)** wait for new messages in the *object process channel*; **(ii)** once new messages are received, send them to an internal buffer of the *object process*; **(iii)** non-deterministically choose a message from the internal buffer and try to apply a rule to process that message (see Rules below); **(iv. a)** if a message is processed and the *object process channel* is empty, return to **(iii)**; **(iv. b)** if no message is processed or the *object process channel* is not empty, return to **(i)**.

Rules For an OBGG object there may exist several rules that are capable of handling the same message type. When receiving a message, one of the enabled rules that can handle the message is chosen in a non-deterministic way. We use a condition structure inside the *object process* to implement such abstraction. This condition structure has in its entries the necessary conditions to trigger the rules of the object (the match). Thus, an object with N rules will have N entries in this structure.

Initial Graph The OBGG initial graph is composed of the instances of objects and the (initial) messages of the model. In our translation, the initial graph becomes an *init* process in PROMELA. This *init* process has three stages: **(i)** create the *object process(es) channel(s)* for objects that appear in the initial graph; **(ii)** execute the *object process(es)* defined in the initial graph, passing as arguments the values of its attributes and its *object process(es) channel(s)*; **(iii)** send defined (initial) messages using the *object process(es) channel(s)*.

4.2 Semantic Compatibility

To assure that the translation preserves the OBGG semantics we have to prove (i) that every behavior in the OBGG-LTS can be found in the PROMELA-LTS of its translation and (ii) that no new behavior is added in the PROMELA-LTS. Due to a difference in the granularity of the LTSs, the latter proof can not be done, only a weaker version of it.

In order to carry out these proofs we translate the paths of the OBGG-LTS into paths of PROMELA-LTS **(i)** and vice-versa **(ii)**. For this, we must translate the states of the first into states of the second LTS, that is, we must find a correspondence between graphs and PROMELA states. We can always translate an OBGG state into a PROMELA state, but the opposite is not true. In the PROMELA-LTS of a program which results from the translation of an OBGG specification there are several states that do not correspond to any states of the OBGG-LTS. This is due to the fact that the treatment of messages in OBGG occurs atomically (in only one step), while in PROMELA this treatment occurs in several steps. Thus, in the PROMELA-LTS, there are states that represent the partial treatment of messages and these states do not have any corresponding state in OBGG-LTS. A PROMELA state that corresponds to some OBGG state will be called well-formed state. In a well-formed state, every instantiated process

($act(i) = (\pi_1, \pi_2,\ L, \varepsilon)$) must: not be more active ($\pi_1 = \varepsilon$); or not be executing ($\pi_1 = \pi_2$); or be ready to execute a labeled statement in the next step. The fist situation is the case of init process. The latter situation occurs when processes are waiting for messages or looking for enabled messages in the buffer. For a PROMELA-LTS state to correspond to a graph (OBGG-LTS state), it must have one active process (but not executing or with a labeled statement) for each object in the graph and one element in the channel or the buffer of an *object process* for each message in graph.

The initial state of an OBGG-LTS is the initial graph of the OBGG, which contains all objects and messages of the initial configuration of the system. The PROMELA-LTS initial state does not correspond to the OBGG-LTS initial state, because there is no running process in the initial state of the PROMELA-LTS. The state corresponding to the initial graph is the output state of the first transition labeled end_atomic (indicating that the *init* process (0) has ended and all processes corresponding to objects and messages present in the initial state of the system are running). The PROMELA state corresponding to the initial graph of Fig. 3 (b) is the state $ST = (\varepsilon, G, C, pdef, act, \perp)$, whose definitions are shown in Fig. 4 (π's definitions are omitted). The function *act* defines the instantiated and active processes. In Fig. 4, we can see that there is one active process for each object in the initial graph $(1 - 6)$. The functions L_n define the values of object attributes. For example, in the initial graph (Fig. 3 (b)) the *asym* attribute of $Phil2$ is true and in PROMELA-LTS state this same value can be seen in $L_3(atr\_asym) = (\mathrm{BOOL}, 1)$. The messages and their parameters, present in the initial graph, are defined by function C, that defines the channel value of each object. All messages (and their parameters) sent to each object are in these channels.

The information described in the type graph of the OBGG can be found via the $pdef$ function, that associates to each type of object (process name) a process body modeling its behavior. We can also obtain the object's attributes, as well as their types through functions $f_{name\_process}$. The message types of system can be found in the *mtype* construct, that enumerates all types of messages and rule names of an entity. The message parameters and their types can also be obtained from the process body associated with each object by the function $pdef$. The first statements in the object process body are the declarations of parameters of all message types that the object can treat.

Besides translating the states we must translate the transitions of the LTS's. In the OBGG-LTS there is one transition for each rule application, but in the PROMELA-LTS there are several transitions that represent the same rule application. The behavior of an object process can result into two kinds of transition sequences, a *matching* sequence, corresponding to testing whether a rule can be applied (without applying the rule), and a *rule application* sequence testing and applying a rule. The matching sequence corresponds to situations when there is no match to apply a rule, and therefore it is only tested and not applied. As this transition sequence only makes tests, the part of the PROMELA state that corresponds to an OBGG state (graph) is not changed. Both sequences start with an "atomic" and end with an "end_atomic" transition. In the rule application sequence, in the final state of the "end_atomic" transition, the local variable $rule\_applied$ is $true$, whereas in the matching sequence the value of this variable is $false$.

$pdef(Fork) = (\pi_3, f_{Fork})$,
$f_{Fork}(1) = (opc\_Fork, \text{CHAN})$
$f_{Fork}(2) = (atr\_acquire, \text{BOOL})$
$pdef(Phil) = (\pi_5, f_{Phil})$,
$f_{Phil}(1) = (opc\_Phil, \text{CHAN})$
$f_{Phil}(2) = (atr\_acquire, \text{BOOL})$
$f_{Phil}(3) = (atr\_eat, \text{BOOL})$
$f_{Phil}(4) = (atr\_release, \text{BOOL})$
$f_{Phil}(5) = (atr\_asym, \text{BOOL})$
$f_{Phil}(6) = (atr\_forks, \text{BOOL})$
$f_{Phil}(7) = (atr\_Fork\_leftFork, \text{CHAN})$
$f_{Phil}(8) = (atr\_Fork\_rightFork, \text{CHAN})$

$pdef(init) = (\pi_7, f_0)$

$G(event\_RuleName) = (\text{MTYPE}, 0)$

$C(0) = (\text{MTYPE}, Phil\_Start \cdot \varepsilon, 3)$
$C(1) = (\text{MTYPE}, Phil\_Start \cdot \varepsilon, 3)$
$C(2) = (\text{MTYPE}, Phil\_Start \cdot \varepsilon, 3)$
$C(3) = (\text{MTYPE} \times \text{CHAN}, \varepsilon, 3)$
$C(4) = (\text{MTYPE} \times \text{CHAN}, \varepsilon, 3)$
$C(5) = (\text{MTYPE} \times \text{CHAN}, \varepsilon, 3)$

(a)

$act(0) = (\varepsilon, \pi_7, L_1, \varepsilon)$
$L_1(Phil1) = (\text{CHAN}, 0)$
$L_1(Phil2) = (\text{CHAN}, 1)$
$L_1(Phil3) = (\text{CHAN}, 2)$
$L_1(Fork1) = (\text{CHAN}, 3)$
$L_1(Fork2) = (\text{CHAN}, 4)$
$L_1(Fork3) = (\text{CHAN}, 5)$
$act(1) = (\pi_5, \pi_5, L_2, \varepsilon)$
$L_2(opc\_Phil) = (\text{CHAN}, 0)$
$L_2(atr\_acquire) = (\text{BOOL}, 1)$
$L_2(atr\_eat) = (\text{BOOL}, 0)$
$L_2(atr\_release) = (\text{BOOL}, 0)$
$L_2(atr\_asym) = (\text{BOOL}, 0)$
$L_2(atr\_forks) = (\text{BOOL}, 0)$
$L_2(atr\_Fork\_leftFork) = (\text{CHAN}, 5)$
$L_2(atr\_Fork\_rightFork) = (\text{CHAN}, 3)$
$act(2) = (\pi_5, \pi_5, L_3, \varepsilon)$
$L_3(opc\_Phil) = (\text{CHAN}, 1)$
$L_3(atr\_acquire) = (\text{BOOL}, 1)$
$L_3(atr\_eat) = (\text{BOOL}, 0)$
$L_3(atr\_release) = (\text{BOOL}, 0)$
$L_3(atr\_asym) = (\text{BOOL}, 1)$
$L_3(atr\_Forks) = (\text{BOOL}, 0)$
$L_3(atr\_Fork\_leftFork) = (\text{CHAN}, 3)$
$L_3(atr\_Fork\_rightFork) = (\text{CHAN}, 4)$
$act(3) = (\pi_5, \pi_5, L_4, \varepsilon)$
$L_4(opc\_Phil) = (\text{CHAN}, 2)$
$L_4(atr\_acquire) = (\text{BOOL}, 1)$
$L_4(atr\_eat) = (\text{BOOL}, 0)$
$L_4(atr\_release) = (\text{BOOL}, 0)$
$L_4(atr\_asym) = (\text{BOOL}, 0)$
$L_4(atr\_Forks) = (\text{BOOL}, 0)$
$L_4(atr\_Fork\_leftFork) = (\text{CHAN}, 4)$
$L_4(atr\_Fork\_rightFork) = (\text{CHAN}, 5)$
$act(4) = (\pi_3, \pi_3, L_5, \varepsilon)$
$L_5(opc\_Fork) = (\text{CHAN}, 3)$
$L_5(atr\_acquire) = (\text{BOOL}, 0)$
$act(5) = (\pi_3, \pi_3, L_6, \varepsilon)$
$L_6(opc\_Fork) = (\text{CHAN}, 4)$
$L_6(atr\_acquire) = (\text{BOOL}, 0)$
$act(6) = (\pi_3, \pi_3, L_7, \varepsilon)$
$L_7(opc\_Fork) = (\text{CHAN}, 5)$
$L_7(atr\_acquire) = (\text{BOOL}, 0)$

(b)

Fig. 4: PROMELA State for OBGG Specification: Definitions, Global Variables and Channels (a) and Active Processes (b)

Figure 5 shows examples of OBGG-LTSs and the PROMELA-LTS of their translations. In (a) and (c), we can apply $r1$ and $r2$ in parallel ($r1$ and $r2$ are independent) and, in (b) and (d), we can apply $r2$ only after $r1$ ($r2$ depends on $r1$). In (a) and (c), we can observe that the two rules can be applied in any order. The states π_4 and π_5 are equivalent, in this context, because they differ only in the value of the global variable $event\_RuleName$ and the local variable $rule\_applied$ of one *object process*. These variables do not interfere in the message or process states, and therefore the OBGG states corresponding to these two different PROMELA states are the same (isomorphic). In (d), the transitions represented by the dashed arrows correspond to idle transitions in the OBGG (because this corresponds to a matching sequence) and states π_5 and π_6 (and also π_3 and π_4) are equivalent, because the differences between them are not in message or object states.

The messages, in the states of PROMELA-LTS, do not have identifiers, so we are not capable of distinguishing, in a state, two messages with same type

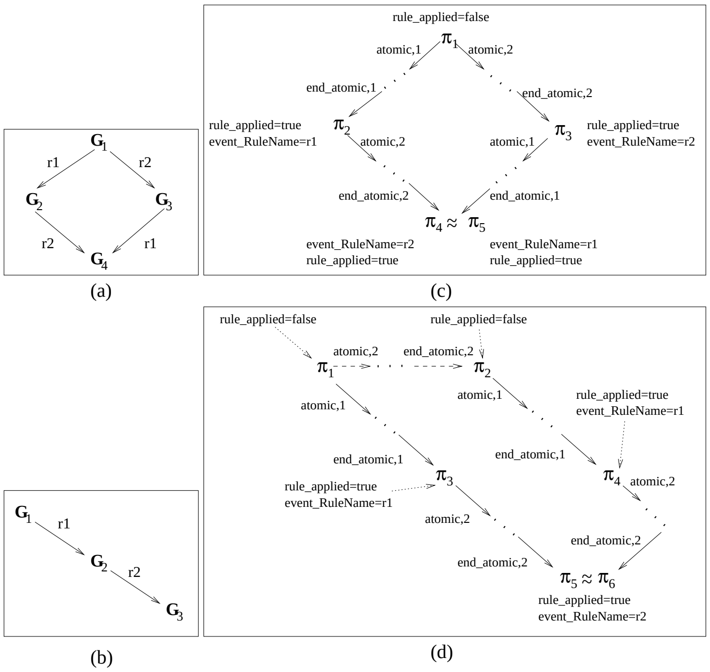


Fig. 5: OBGG-LTS's (a)(b) and PROMELA-LTS's (c)(d) of its Translation

and parameters. In an OBGG, however, messages have unique identifiers. This means that the OBGG-LTS have a richer representation concerning the causality relationships among messages than the PROMELA-LTS. For each OBGG path there is one corresponding PROMELA path, but for the same PROMELA path there may be many different corresponding OBGG paths, each one representing a different causal relation between messages that is compatible with the PROMELA path. What is important to notice is that all these OBGG paths are in the transition system of the original OBGG-LTS, and therefore we are not adding new behavior with the translation of OBGG into PROMELA.

5 Verification of OBGG Specifications

The model checker SPIN is a state-based verification tool. The property formulas, written using LTL, are specified over the state of the system. More specifically, the developer must have global variables in the PROMELA model to specify and verify properties over it.

For the verification of OBGG specifications we have noticed that it is more natural for the developer to express properties about the application of rules rather than based on the state of specific objects. Moreover, if we use the state of an object (its attributes) in a property specification we would have to make available the values of such attributes through global variables. While this approach works for specifications with a static number of objects, it is not feasible for specifications that have dynamic creation of objects because we would need to dynamically create new global variables, a feature not supported by the tool.

Switching from the state-based approach presented above to an event-based approach, besides being more natural from the OBGG point of view (a rule application is seen as an event), we gain a more structured form to handle the verification of specifications that have dynamic creation of objects.

In order to specify properties using rule applications as events we have somehow to mark that a rule application is an event for verification. In our approach using events, every PROMELA model generated from the translation has a global variable *event_RuleName*. To generate events for the application of rules, we include the name of the rules that will become events into the *mtype* definition of this variable. Thus, when interesting events occur, i.e., the application of rules that are relevant for verification, they are written (using the *atomic* structure in PROMELA) into the *event_RuleName* global variable.

It is then possible to write LTL formulas about the occurrence of rules as being events, and these formulas need to inspect only the global variable *event_RuleName*. An event is the change of value of this global variable. For instance, we can define an event *eat* as being the change of value of the variable *event_RuleName* from *not Eating* to *Eating* (where *Eating* is the rule applied). We need to use the *next* temporal operator (X) to mark the change of value, for instance (! $Eating$ && X $Eating$). The idea of specifying LTL formulas using events, and validating them with SPIN has been explored in [2].

As an example, we can define an LTL formula to specify that "it is always possible that some philosopher will eat", trying to prove that the specification is deadlock free. For that, we generate the events *Eating* (the philosopher is eating), *SymStart* (a philosopher is starting its execution), and $AsymStart$ (a philosopher is starting its execution). This property is specified by the formula ([] <> eat), where ([]) is the always temporal operator and (<>) is the eventually temporal operator. We were able to verify this formula for the symmetric and asymmetric solutions of the dining philosophers problem. As expected, for the symmetric solution the formula does not hold, but the formula does hold for the asymmetric solution, where it used 530 Mb of memory, generated 6.21266e+06 states, and took 9 minutes running in an Intel Xeon 2.2 GHz Processor with a limit of 1 Gb of memory under SPIN.

Another property verified over the asymmetric and symmetric models concerned mutual exclusion. In a setting with up to three philosophers it is sufficient to prove that "no two philosophers might be in their critical sections (eating) at the same time". To prove such property we use the events *Eating* (the philosopher enters in the critical section) and *ReleaseForks* (the philosopher leaves the critical section). The formula ([] (eat && <> rel) $\rightarrow$ X (! eat U rel)) specifies this property, where (U) is the strong until temporal operator. Like for the event *eat* defined above, we define the event *rel* as being the change of value of the *event_RuleName* variable from *not ReleaseForks* to *ReleaseForks* (where *ReleaseForks* is the rule applied), leading to (! $ReleaseForks$ && X $ReleaseForks$). When verified, the formula for mutual exclusion was true and used 420 Mb of memory, generated 2.67997e+06 states, and took 2 minutes running in the same computer and configuration of the previous formula.

It is important to note that this approach implies in the use of the *next* operator in SPIN. In order to use the partial order reduction algorithm available, the SPIN model checker requires that, when using the *next* operator, the given formula is closed under stuttering. [2] has proposed a group of useful formula patterns for events that are closed under stuttering and can be directly applied in our work.

An important feature of our approach based on events is that the developer may write formulas looking only to the OBGG specification. It is not necessary to know the structure of the translated PROMELA model.

6 Final Remarks

In this article we have defined a translation from OBGG specification to PROMELA, and provided a way to verify properties over OBGG specifications based on events. We have also discussed the semantic compatibility between an OBGG specification and its corresponding PROMELA model.

The translation and integration between formal languages in order to use model checking tools is becoming a common practice, since many times it is easier (and more efficient) to reuse than to build a specific verification tool. Nevertheless, such translations involve detailed comparisons, especially at the semantic level. We can find in the literature various works focused on the verification of object-based/oriented distributed systems. The work proposed in [11] defines a visual and object-oriented language that can be mapped to the model checker SPIN. In [3] PROMELA is extended considering the actors concurrency model. [12] proposes a tool that tries to make available the automatic verification of UML models, this approach consists in the mapping of UML models to PROMELA. In [17] an integration of the formal specification language Object-Z with ASM (*Abstract State Machine*) was introduced, creating the OZ-ASM notation. After a series of translations, it is possible to verify OZ-ASM specifications using the SMV tool. In contrast to some of these works, in this article we discussed the semantic compatibility of our translation. Moreover, in our approach it is possible to specify the properties to be verified at the same level of abstraction of the specified OBGG model, a feature that is not present in most of the approaches that use translations.

For the verification of some aspects of a given specification, only the name of the applied rule as event name may be too few information. For instance, if we wish to prove the problem of fairness for the asymmetric solution of the dining philosophers problem, we would need to specify formulas about the individual behavior of a philosopher, like “for each philosopher i it is always possible that philosopher i will eat”. To support such level of detail we need to add more information to the event names. However, in doing so, we will have exponentially more states in the LTS to be verified. We are currently working on an approach to tackle this problem.

References

1. Promela language reference. http://spinroot.com/spin/Man/promela.html, 2003.
2. M. Chechik and D. O. Păun. Events in property patterns. In *5th and 6th Int. SPIN Workshops*, volume 1680 of *LNCS*, pages 154–167, Germany, 1999. Springer.
3. Seung Mo Cho et al. Applying model checking to concurrent object-oriented software. In *4th International Symposium on Autonomous Decentralized Systems*, pages 380–383, Japan, 1999. IEEE Computer Society Press.
4. B. Copstein, M. C. Móra, and L. Ribeiro. An environment for formal modeling and simulation of control systems. In *33rd Annual Simulation Symposium*, pages 74–82, USA, 2000. IEEE Computer Society.
5. F. L. Dotti, L. M. Duarte, B. Copstein, and L. Ribeiro. Simulation of mobile applications. In *2002 Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 261–267, USA, 2002. The Society for Modeling and Simulation International.
6. F. L. Dotti, L. M. Duarte, F. A. Silva, and A. S. Andrade. A framework for supporting the development of correct mobile applications based on graph grammars. In *6th World Conference on Integrated Design & Process Technology*, pages 1–9, USA, 2002. Society for Design and Process Science.
7. F. L. Dotti and L. Ribeiro. Specification of mobile code systems using graph grammars. In *4th International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 177 of *IFIP Conference Proceedings*, pages 45–63, USA, 2000. Kluwer.
8. F. L. Dotti, O. M. Santos, and E. T. Rödel. On the use of formal specifications to analyze fault behaviors of distributed systems. In *1st Latin-American Symposium on Dependable Computing (accepted)*, LNCS, Germany, 2003. Springer.
9. H. Ehrig. Introduction to the algebraic theory of graph grammars. In *1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69, Germany, 1979. Springer.
10. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
11. S. Leue and G. Holzmann. v-Promela: a visual, object oriented language for SPIN. In *2nd International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 14–23, France, 1999. IEEE Computer Society Press.
12. Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. In *14th International Conference on Automated Software Engineering*, pages 255–258, USA, 1999. IEEE Computer Society Press.
13. A. B. Loreto, L. Ribeiro, and L. V. Toscani. Decidability and tractability of a problem in object-based graph grammars. In *17th IFIP World Computer Congress - Theoretical Computer Science*, volume 223 of *IFIP Conference Proceedings*, pages 396–408, Canada, 2002. Kluwer.
14. L. Ribeiro and B. Copstein. Compositional construction of simulation models using graph grammars. In *Application of Graph Transformations with Industrial Relevance (AGTIVE'99)*, volume 1779 of *LNCS*, pages 87–94, Germany, 2000. Springer.
15. G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation*, volume 1: Foundations, Singapore, 1997. World Scientific.
16. C. Weise. An incremental formal semantics for PROMELA. In *3rd SPIN Workshop*, The Netherlands, 1997.
17. Kirsten Winter and Roger Duke. Model checking object-Z using ASM. In *3rd International Conference on Integrated Formal Methods*, volume 2335 of *LNCS*, pages 165–184, Germany, 2002. Springer.