Science of Computer Programming 77 (2012) 214–246

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Correct transformation: From object-based graph grammars to PROMELA

Leila Ribeiro[a,*], Osmar Marchi dos Santos[b], Fernando Luís Dotti[c], Luciana Foss[d]

[a] *Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil*
[b] *Department of Electronics and Computing, Universidade Federal de Santa Maria, Santa Maria, Brazil*
[c] *Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil*
[d] *Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, Brazil*

ARTICLE INFO

Article history:
Received 20 June 2010
Received in revised form 17 March 2011
Accepted 28 March 2011
Available online 7 April 2011

Keywords:
Graph grammars
Model transformation
PROMELA
Correctness

ABSTRACT

Model transformation is an approach that, among other advantages, enables the reuse of existing analysis and implementation techniques, languages and tools. The area of formal verification makes wide use of model transformation because the cost of constructing efficient model checkers is extremely high. There are various examples of translations from specification and programming languages to the input languages of prominent model checking tools, like SPIN. However, this approach provides a safe analysis method only if there is a guarantee that the transformation process preserves the semantics of the original specification/program, that is, that the transformation is correct. Depending on the source and/or target languages, this notion of correctness is not easy to achieve. In this paper, we tackle this problem in the context of Object-Based Graph Grammars (OBGG). OBGG is a formal language suitable for the specification of distributed systems, with a variety of tools and techniques centered around the transformation of OBGG models. We describe in details the model transformation from OBGG models to PROMELA, the input language of the SPIN model checker. Amongst the contributions of this paper are: (a) the correctness proof of the transformation from OBGG models to PROMELA; (b) a generalization of this process in steps that may be used as a guide to prove the correctness of transformations from different specification/programming languages to PROMELA.

1. Introduction

Modern software systems are becoming ubiquitous in the sense that more and more people depend on them to carry out from simple to safety-critical daily activities. Such systems have increased complexity and specialization levels and the ability to propose abstractions to formally specify and reason about specific classes of systems in a suitable way is gaining in importance. In this context, the capacity to carry out transformations between models is highly desired such that existing concepts and tools can be reused to integrate, analyze and implement software systems.

In this paper, we report on results and experiences in this direction. We have proposed a formal specification language tailored to a specific class of systems, namely asynchronous distributed systems, and adopted a transformation approach whenever possible to take advantage of several existing analysis and implementation techniques, languages, and tools.

We have adopted Graph Grammars (GGs) [1] as a foundation and proposed a formal specification method, which we called Object-Based Graph Grammars (OBGGs) [2]. Graphs are a very natural means to explain complex situations on an

* Corresponding author. Tel.: +55 51 33087308.
E-mail addresses: leila@inf.ufrgs.br (L. Ribeiro), osmar@inf.ufsm.br (O.M. dos Santos), fernando.dotti@pucrs.br (F.L. Dotti), luciana.foss@ufpel.edu.br (L. Foss).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.03.010

intuitive level. Graph rules may complementarily be used to capture the dynamical aspects of systems. The resulting notion of Graph Grammar [3,1,4] generalizes Chomsky grammars from strings to graphs. Due to their declarative nature, GGs are well-suited for concurrent systems' specification and have already been used for the specification of distributed systems [5]. The basic idea is to model the states of a system as graphs and describe the possible state changes as rules (where the left- and right-hand sides are graphs). The behavior of the system is then described via applications of these rules to graphs describing the actual states of a system. Rules operate locally on the state-graph, and therefore it is possible that many rules are applied at the same time. Although formal, GGs are considered intuitive and easy to learn. Thus, this formalism is a good candidate to be integrated and used by the wide community which currently adopts semi-formal, often visual, methods like UML.

Object-based graph-grammars [2] follow the object paradigm, well-known by most users. The language itself is a restricted form of Graph Grammars and captures the main abstractions to represent reactive and distributed systems considering the asynchronous computation model: communication takes place through message passing; state changes are local and concurrent; and time for processing as well as message delivery is unbounded, characterizing the asynchronous computation model [6].

The functional analysis of OBGG models is supported by model checking [7,8] using a transformation to PROMELA. An approach for the analysis of partial OBGG models is presented in [9]. The quantitative analysis of OBGG models is possible using several techniques. If delay distribution probabilities are assigned to messages, OBGG models can be translated to discrete event simulation models. Both a simulation kernel and a library to define simulation entities out of OBGG objects were proposed [10,11]. Analytical Markovian models can be generated out of OBGGs through a transformation [12] to Stochastic Automata Networks [13]. If we define temporal assumptions on message delays, a transformation from OBGGs to Timed Automata [14] allows for analysis of deadlines to receive and process messages. Besides these analysis methods, there is also the possibility of generating code for execution in a real environment, via a transformation to the Java programming language [11]. Modeling high-performance applications in OBGGs is possible via a mapping to cluster environments [15] (C++ code using MPI (Message Passing Interface) [16]). Moreover, [17] and [18] introduce the representation of classical fault models for distributed systems in OBGG models, allowing one to reason about a distributed system in the presence of such faults. According to classical ideas in the literature of fault-tolerant distributed systems [19], the representation of faulty behavior takes place through a model transformation step.

By using the methods and tools mentioned above, a framework to assist the development of concurrent and distributed systems was defined [20]. A tool to assist the modeling and reasoning of OBGG systems has been developed [21]. Various models have been defined and analyzed using OBGGs: mobile code applications [2], a pull-based failure detector [17], active networks [22], distributed election in a ring [23], dining philosophers [7], and readers and writers [8], among others.

Our approach is strongly based on transformations from OBGG to several target environments and languages. Of special interest in this paper is the discussion about how systems described with OBGGs can be verified using the model checker SPIN [24]. OBGG models can be translated to PROMELA (the input language of SPIN). Here we focus on proving the semantic compatibility of the generated PROMELA model, described by the transformation, with respect to the original OBGG model. This is of paramount importance in a model transformation based approach, which we proposed to follow, and comprises a non-trivial task that, to be accomplished, unfolds in several details and arguments. The proved correct transformation from OBGG to PROMELA plays an important role in our setting since several other transformations mentioned above are based on this one. Whilst this does not eliminate the need to show their semantic compatibility, the proof presented in this paper can be used as a basis to construct other proofs.

The issue of proving correctness (in the sense of behavior preservation) when transforming one model into another one is quite old in computer science [25–30]. This was one of the triggers for the development of a rich theory of formal semantics and verification, as well as many software development methods. With the growing complexity and size of computing systems, the plethora of heterogeneous platforms currently available, and the increasing number of languages to describe various aspects in different stages of software development, the area of Model Driven Engineering (MDE) [31] has gained strength. MDE aims to increase the productivity of software development by focusing on the definition of models. The basic concepts are that every language used in the development process must be formalized as a meta-model and then translations between different languages can be defined by transformations of the corresponding meta-models. Model transformation is thus used to relate information between models.This relation may occur within the same abstraction level (different views of the same system, optimizations, migration, etc.) or between different levels (refinements, code generation, refactoring, etc.). Just as in the case of translating specifications to programming languages studied decades ago, the issue of proving the correctness of model transformations is fundamental in MDE, assuring that the transformation does not change the behavior of the system. Unfortunately, this problem is impossible to solve at a general level, since it strongly depends on the definition of the concrete transformation and usually involves a lot of details about the models being used and their semantics. Although specific, we believe that the proposed semantic compatibility proof for the transformation from OBGG to PROMELA presented in this paper contributes in a broader scope:

- The proposed translation[1] is based on classes, objects and messages, that are traditional concepts, which appear in the definition of both specification and programming languages following the object-based paradigm. Moreover, we consider

[1] From now on, the terms translation and transformation are used interchangeably in this paper.

the behavior of object-based systems in a scenario where objects execute concurrently and communicate exclusively via message passing. Since we show how to model such behavior in terms of PROMELA, it is possible to use our translation as a basis to define other translations from different object-based languages to PROMELA.

- SPIN is a popular state-of-the-art model checker, and its input language PROMELA is used as a target for model checking several different languages. In Section 2 we survey some transformations for object-based/oriented languages and note that formal proofs of the semantic compatibility of the transformations are not presented. Although informal argumentation is important, the process of constructing a proof typically reveals many subtleties and leads to a deeper understanding of the considered models. We believe that transformations from other languages equipped with transition system semantics (a representative class) to PROMELA can potentially benefit from the strategies and proofs presented here. It is worth mentioning that the formal semantics for PROMELA used in this paper (see Section 5) is a revised version of existing ones, and it can be used as the basis for correctness proofs from other input languages.

This paper is structured as follows. First, we discuss related work (Section 2). In Section 3 we present the Object-based Graph Grammars specification language. The approach adopted for verifying OBGG models is then described in Section 4. In Section 5 the PROMELA language is presented. A summary describing general guiding steps for proving the correctness of a transformation (semantic compatibility) from a language with formal syntax and semantics to PROMELA is shown in Section 6. Using these steps, in Sections 7 and 8 we define in detail the transformation of OBGG models to PROMELA and present a proof for the semantic compatibility of this transformation. Finally, Section 9 presents closing remarks.

2. Related work

Discussion on related work is divided in two different research areas: (i) Approaches to model checking Graph Grammars; and (ii) Contributions aiming at the verification of object-based/oriented systems using transformations to PROMELA.

There are several contributions addressing the verification of GGs. Some of these contributions propose approaches to the verification of infinite-state systems using unfolding techniques [32] or theorem proving [33] and others concentrate on finite-state systems using model checking, like the one proposed in this paper. Amongst approaches for GG model checking, we can cite CheckVML and GROOVE, described and compared in [34]. The main difference from those contributions to our approach is that both CheckVML and GROOVE address graph grammars in general. OBGG imposes important restrictions that fit well in the context of object-based systems, reducing the problem of finding matches for rules in a state graph. Another difference is that both works in [34] focus on reachability properties. In [35] a comparison between GROOVE and SPIN for the verification of graph-based systems is presented. The same example was specified using GROOVE and PROMELA and results were analyzed. The main result was that, using a suitable encoding of graphs, the SPIN model checker was able to verify the properties in a much more efficient way. No explicit transformation from GROOVE to PROMELA was presented, and no proof of behavioral equivalence between the two specifications was carried out. Our approach is complementary to these ones, since we focus on checking properties of derivations, that is, of sequences of events (rule applications).

The surveyed contributions, whose aim is the verification of object-based and object oriented systems, consider the transformation of the corresponding language to PROMELA. This approach is becoming a common practice, since many times it is easier (and more efficient) to reuse than build a specific verification tool. Moreover, we can classify these contributions by the kind of language being either visual or non-visual.

There are various approaches for the verification of visual languages [36–39]. The one proposed in [36] defines a visual and object-oriented language (called v-PROMELA) that can be mapped to the SPIN model checker. Property specifications can be defined over v-PROMELA models, instead of translated PROMELA models, but there is no approach to visualize counter-examples in terms of v-PROMELA. [37] proposes a tool, called vUML, that maps UML models to PROMELA. Using this tool, it is possible to verify UML models with respect to deadlocks, livelocks, invalid states, etc. The counter-examples for verifications using SPIN are presented with UML sequence diagrams. Another approach to translate UML diagrams to PROMELA can be found in [38]. Here a special notation is used because the main aim is to verify distributed systems that are specified over the CORBA middleware. During transformation, the behavior of the middleware service is included automatically. In none of these transformations was an explicit proof of behavioral preservation shown. They rather present informal discussions to argue about the correctness of the approach. In [39] Graph Grammars are used as a basis to define the semantics of visual languages (type graphs are viewed as meta-models, graphs are corresponding models and rules specify the behavior of the models). A formal proof of operational equivalence relating the Graph Grammar and the corresponding ASM (Abstract State Machine) is presented [40]. Differently, our aim is to provide a correct transformation to PROMELA, enabling the use of the SPIN model checker to verify OBGGs. Our proof is more complex, in the sense that PROMELA is much more different from graph transformations than ASMs (that also rely on rule based transformations, like GGs).

When focusing on the verification of non-visual object-based systems, there are also many approaches, including [41–45]. [41,42] aims at the verification of restricted Java programs via the transformation to the input language of the SPIN model checker. The authors in [43] extend PROMELA to consider the actors concurrency model, in order to model check object-based distributed systems. This is also the case of the language Rebeca [45], that can use the SPIN tool for verification. An UML profile for concurrent and distributed systems that can be converted to Rebeca was proposed in [46]. However, in none of these works was a formal proof of correctness of transformation given.

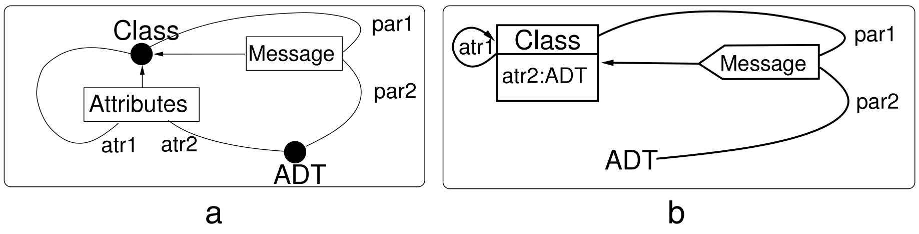

Fig. 1. (a) Object-Based Graph Scheme. (b) Graphical Representation of Object-Based Graphs.

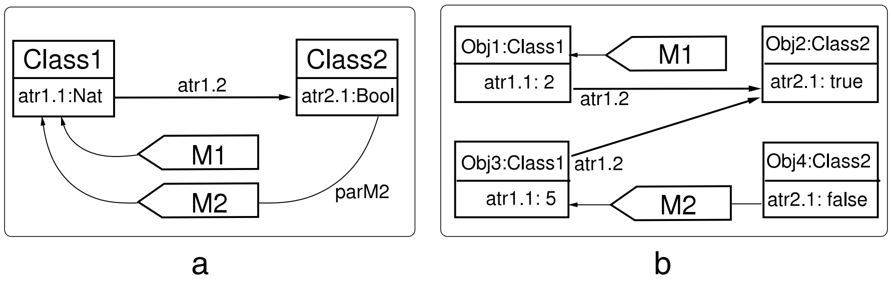

Fig. 2. (a) Class Graph. (b) ObjectGraph

3. Object-Based Graph Grammars

In this section, we first present an informal introduction to object-based graph grammars. Then, in Section 3.2, we exemplify its use with the dining philosophers problem such that the main notions can be clarified. In Section 3.3, the formal definitions of OBGGs are introduced.

3.1. Informal description

The specification of an object-based system is done via an (object-based) graph grammar. Objects encapsulate attributes and communicate through message passing. Graph rules specify object behavior in reaction to messages. The graphs used in this work are called *object-based graphs* and were introduced in [2]. Each graph in an object-based graph grammar may be composed by instances of the vertices and edges shown in Fig. 1(a). The vertices represent objects and Abstract Data Types (ADTs), whereas messages and attributes of objects are modeled as hyperedges (edges with one destination and many source vertices). We defined a distinguished graphical representation for these graphs to increase the readability of the specifications. This representation is shown in Fig. 1(b). Elements of ADTs are allowed as attributes of classes and/or parameters of messages. Note that the graph in Figure Fig. 1 defines only a scheme of which kinds of vertices and edges may occur in a specification, and does not oblige classes or messages to have attributes.

Fig. 2(a) shows an example of a graph that can be used as a description of classes and messages. There are two classes: Class1, that has attribute atr1.1 of type natural number, and atr1.2, that is a reference to an object of type Class2; and Class2, that has only one attribute atr2.1 of type boolean. Objects of Class1 may receive messages M1 (without parameters) and M2 (with a reference to an object of type Class2). A concrete instance of these class types is a set of objects together with corresponding messages, for example, the one illustrated in Fig. 2(b), with two instances of each of the existing classes.

A rule expresses the reaction of a class to the receipt of a message and consists of:

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

Formally, we use typed attributed hypergraphs and the rule is a (partial) graph homomorphism with application conditions — formal definitions are presented in Section 3.3. Now we can define an *object-based system*, which is composed of:

- *a Type Graph or Class Graph*: a graph containing information about all attributes of all classes involved in the model (an attribute may be either an ADT or a reference to another object) and messages sent/received by each class. This graph can be seen as an instantiation of the object-based graph scheme described above;
- *a set of Rules*: the rules specify how the instantiated objects of a class will behave when receiving messages. For the same kind of message, we may have many rules specifying the intended behavior. Depending on the conditions imposed by these rules (on the values of attributes and/or parameters of the message), they may be mutually exclusive or not. In the latter case, one of them will be chosen non-deterministically to be executed. Note that the behavior of an object when receiving a message is not specified as a series of steps that shall be executed, but rather as an atomic change of the values of the object attributes together with the creation of new messages to other (or the same) objects. That is, there is no control structure to govern the application of the rules that specify the behavior of a class. Our approach is data driven. This has the advantage that unnecessary sequentializations of computation steps are avoided because the developer only has to care about the causal dependencies between events;
- *an Initial Graph*: this graph specifies the instantiated objects from the classes of the model, the initial values of attributes of these objects, as well as messages that must be sent to these objects when they are created. The messages in this graph can be seen as triggers for the execution of the objects.

3.2. The dining philosophers problem

We take the classical dining philosophers problem [47] to exemplify the use of OBGGs. In a table there are **N** philosophers in a circle, with a common fork between every two philosophers (with **N** forks in total). Philosophers spend some time thinking and, from time to time, a philosopher gets hungry. In order to eat a philosopher must grab both his left and right forks. After eating a philosopher releases both left and right forks, and starts thinking again. Neighboring philosophers compete on the common fork.

The dining philosophers is a classical problem employed to discuss concurrency and synchronization. Traditional models for the dining philosophers are based on shared memory. As already mentioned, here we employ message passing as communication mechanism and consider distributed systems, where no global state is available. Since message passing decouples synchronous actions in a shared memory model, intermediate states, where communication takes place, naturally appear. Our model for the dining philosophers problem defines objects for representing forks (the Fork class) and philosophers (the Phil class).[2] Forks are acquired and released through message passing among philosophers and forks. This resembles a distributed system composed by several processes, without a known global state, which is the class of applications we address.

The type graph for classes Fork and Phil is presented in Fig. 3(a). The definitions of the rules of these classes are also shown, respectively, in Fig. 3 (b) and (c). An instance of a Fork class is composed of the attributes id (an identification number) and use (determines if the fork is currently in use by a philosopher, true value, or not, false value). A Fork object can receive two types of messages, Acq and Rel, used by a philosopher to acquire (grab the fork) or release the fork, respectively.

An instance of a Phil class is composed of the attributes: id, an identification number; leftfirst, that defines the order in which the philosopher tries to acquire the forks; leftfork and rightfork, which are references to Fork objects, representing the left and right forks, respectively; phase, representing the current phase of the philosophers behavior, which may be thinking, hungry (and thus competing for forks), or eating. Once defined at the initial graph (see Fig. 4), the leftfirst attribute is never changed. Its sole purpose is to define the order in which the philosopher acquires the forks: left fork then right fork (attribute is set to true), right fork then left fork (attribute is set to false).

With respect to the phase attribute, a philosopher may have the following phases: thinking (phase = 1), hungry holding zero forks (phase = 2) or one fork (phase = 3), or eating (phase = 4). In the rules defined for the Phil class (Fig. 3 (c)), we need to access the current value of the phase attribute. Therefore, we instantiate at the Left-Hand Side (LHS) of the rule a variable, called n, which contains a snapshot of the initial value of phase at the start of the rule application. Variable n is then used to define a guard for the rule application (shown below the arrow from the LHS to the Right-Hand Side (RHS) of the rule) and, when needed, to specify the new value for the phase attribute (at the RHS of the rule).

The behavior of a Phil object is defined as follows. Initially, the philosopher is thinking (phase = 1). The philosopher stops thinking and starts competing for forks when a message GetFork is processed (phase = 2). Depending on the value of leftfirst, the philosopher asks for the fork at the left (rule FFLeft — First Fork Left) or at the right (rule FFRight — First Fork Right) side. The philosopher then waits for an answer to acquire the first (required) fork. When this answer arrives (message HasFork), the philosopher proceeds to ask for the second (required) fork (phase = 3), again depending on the value of leftfirst, with rules SFRight (Second Fork Right) and SFLeft (Second Fork Left). When the philosopher receives confirmation of acquiring

[2] One could optionally model access to a fork with a protocol for distributed mutual exclusion, such as [48], among every two philosophers. This would eliminate the need for fork objects but certainly overcomplicate philosophers for our illustrative objectives here.

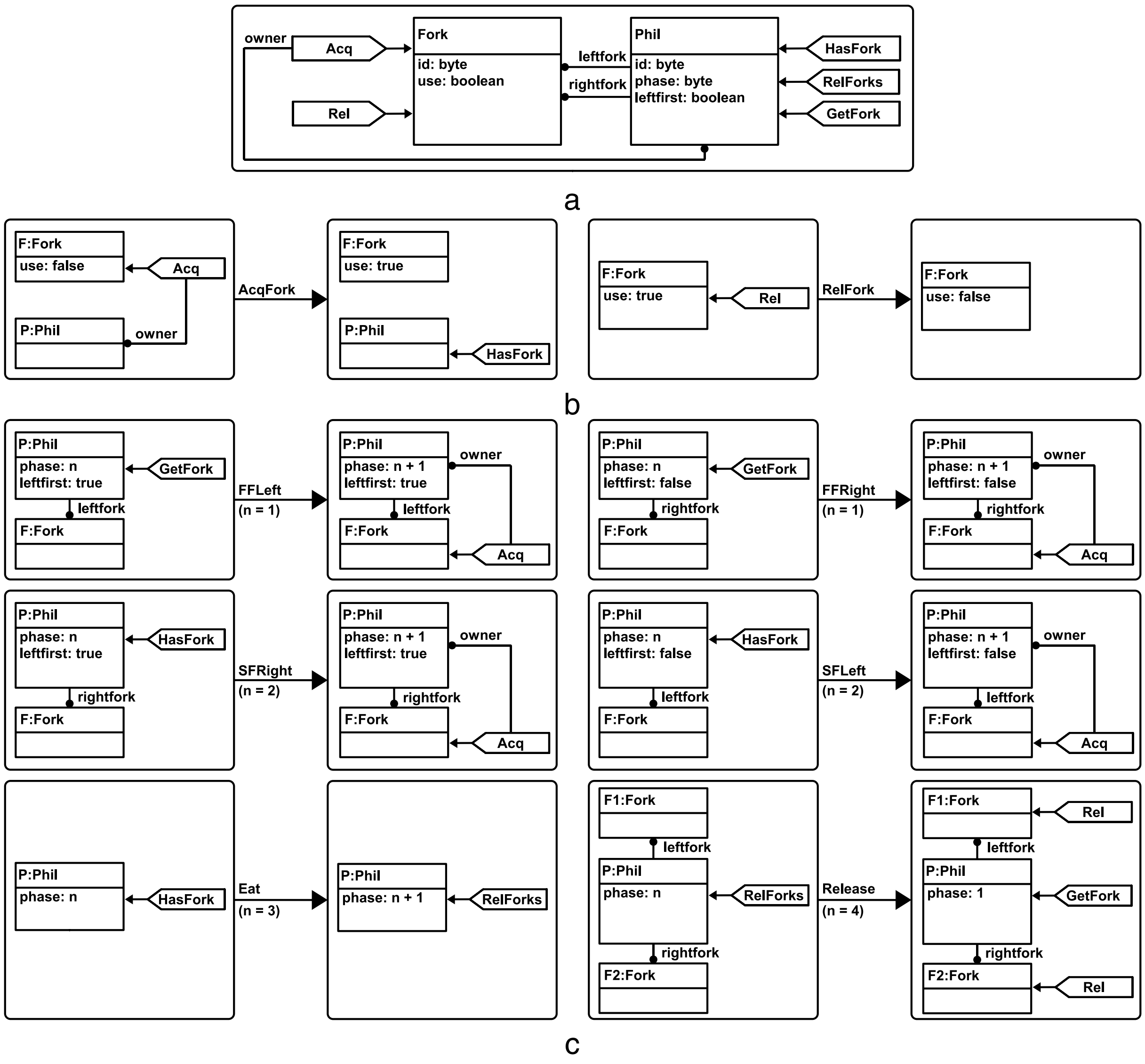

Fig. 3. Type graph for *Fork* and *Phil* classes (a), rules for *Fork* class (b) and rules for *Phil* class (c).

the second fork (message HasFork in rule Eat), the philosopher stops competing for forks and starts eating (phase = 4). Moreover, it sends a message RelForks to itself, so that it can eventually stop eating. When the philosopher processes the RelForks message, it stops eating and starts thinking again (phase = 1). This culminates on the release of the acquired forks with messages Rel to each fork, and the message GetFork being sent to itself, starting a new cycle.

Now we can define an initial graph as a composition of instances of philosophers, forks and corresponding messages. We may study the behavior of the model when all philosophers try to grab the left fork first — see Fig. 4(a). This is called a symmetric configuration, since all philosophers have the same behavior. Fig. 4(b) shows an asymmetric configuration of the dining philosophers problem, where the object Phil2 has the leftfirst attribute set to false (different from the other philosophers), indicating that it will try to acquire the right fork first.

In terms of correctness, a dining philosophers solution should have the following properties guaranteed to hold:

1. *Deadlock freedom*: it is always possible for a philosopher to start eating;
2. *Mutual exclusion*: no two neighboring philosophers should eat at the same time.

There is a third property called *non-starvation* that might also be required, assuring that each philosopher may always eat again in the future. Guaranteeing non-starvation typically requires priorities or ordering mechanisms for messages, and to keep the model easy, we do not consider this aspect in this paper.

The two discussed properties (deadlock freedom and mutual exclusion) are used to demonstrate the verification of properties for OBGG and are further discussed in Section 4.

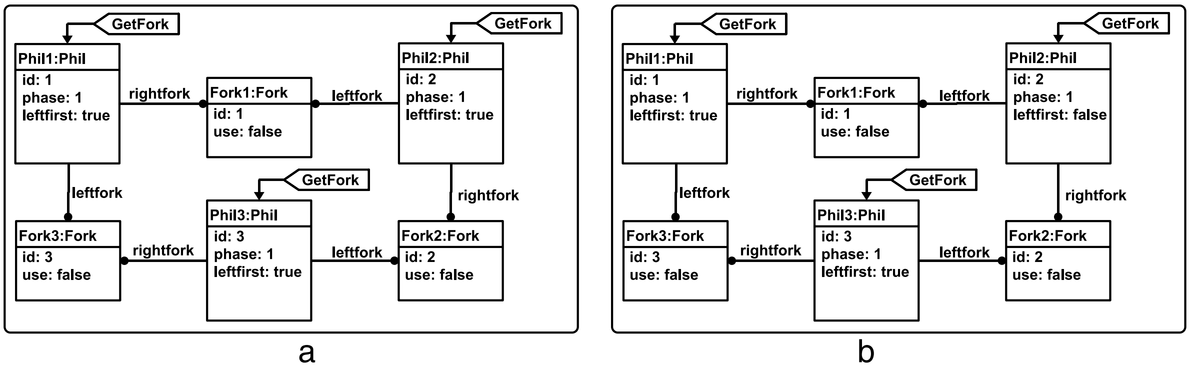

Fig. 4. Initial graph for symmetric (a) and asymmetric (b) configurations.

3.3. Formal definitions

Now we present the formal definitions of OBGG used in this work, which are based on the definition of OBGG found in [2]. It is assumed that the reader is familiar with basic notions of algebraic specification. Though, we introduce them informally as necessary.

Our approach uses the concepts of (typed and attributed) *hyper*graphs, i.e., graphs where edges can be connected to any (finite) number of vertices. Graphically, an edge is depicted as a box (whose shape may vary), and the connections to the vertices are drawn as thin lines, called *tentacles*. The tentacles of an edge are labeled by natural numbers. The main characteristics of object-based graphs (OB-graphs) are:

- Each OB-graph models a set of objects, in which the internal state of an object (its set of attributes) consists of references to other objects and/or values of pre-defined data types;
- The set of vertices is partitioned into two, modeling object identities and data values;
- The set of (hyper)edges is partitioned into two, modeling messages and attributes of an object. Each edge has one target (the object that receives the message or the object to which the attribute belongs) and may have many sources (parameters of the message or attributes of the object).

Following the Single-PushOut (SPO) approach to graph grammars [49], the definition below is based on a category of graphs and *partial* morphisms.

Definition 1 (*Weak Commutativity*). Given two partial functions $f, f' : A \to B$, we say that f is **less defined** than f' (and we write $f \le f'$) if $dom(f) \subseteq dom(f')$ and $f(x) = f'(x)$ for all $x \in dom(f)$. Given two partial functions $f : A \to B$ and $f' : A' \to B'$, and two total functions $a : A \mapsto A'$ and $b : B \mapsto B'$, we say that the resulting diagram **commutes weakly** if $f' \circ a \le b \circ f$.

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "a"'] \arrow[dr, phantom, "\le"] & B \arrow[d, "b"] \\ A' \arrow[r, "f'"'] & B' \end{tikzcd}$$

Now, we introduce OB-graphs. As discussed before, each hyperedge has one target vertex, and may have many source vertices. Source vertices are identified by different numbers of the tentacles, that is, a hyperedge is associated to a *list of vertices*. Each hyperedge has as label a list of names corresponding to each of its tentacles. For the data part, we assume a fixed specification *Spec*. Each OB-graph has an algebra with respect to this specification, whose carrier sets contain the possible values that can be used as attributes of objects or parameters of messages.

Algebraic specification basics: A *signature* $SIG = (S, OP)$ consists of a set S of sorts and a set OP of constant and operation symbols. Given a set of variables X (of sorts in S), the *set of terms* over SIG is denoted by $T_{OP}(X)$ (this is defined inductively by stating that all variables and constants are terms, and that all admissible applications of operation symbols in OP to existing terms are also terms). An *equation* is a pair of terms $(t1, t2)$, and is usually denoted by $t1 = t2$. A *specification* is a pair $SPEC = (SIG, Eqns)$ consisting of a signature and a set of equations over this signature. An *algebra* for specification $SPEC$, or $SPEC$-algebra, consists of one set for each sort symbol of SIG, called *carrier set*, and one function for each operation symbol of SIG such that all equations in $Eqns$ are satisfied (satisfaction of one equation is checked by substituting all variables in the equation by values of corresponding carrier sets and verifying whether the equality holds, for all possible substitutions). Given two $SPEC$-algebras, a homomorphism between them is a set of functions mapping corresponding carrier sets that are compatible with all functions of the algebras. The set obtained by the disjoint union of all carrier sets of algebra A is denoted by $\mathcal{U}(A)$.

Definition 2 (*Object-Based Graph, Object-Based Graph Morphism*). Given an algebraic specification *SPEC*, an **object-based graph** (OB-graph) G is a tuple $G = (V_G, E_G, s^G, t^G, Lab_G, lab^G, Alg_G, a^G)$, where V_G is the set of *vertices* and is partitioned into sets $objV_G$ and $valV_G$ (of objects and data values, respectively), E_G is a set of *(hyper)edges* and is partitioned into sets $msgE_G$ and $atrE_G$ (of messages and attributes, respectively), a total *source* function $s^G : E_G \to V_G^*$, assigning a list of vertices to each edge, a total *target* function $t^G : E_G \to objV_G$ assigning an object vertex to each edge, a set of labels Lab_G, a total *label* function $lab^G : E_G \to Lab_G^*$, assigning a list of labels to each edge, an algebra Alg_G over *SPEC*, and an attribution function $a^G : valV_G \to \mathcal{U}(Alg_G)$, assigning to each value-vertex a value from a carrier set of Alg_G such that the list of names associated to each edge (via lab^G) has exactly the same length as the number of source vertices associated to the same edge (via s^G).

A **(partial) OB-graph morphism** $g : G \to H$ is a tuple (g_V, g_E, g_L, g_A), where the first two components are partial functions $g_V = g_{oV} \cup g_{vV}$ with $g_{oV} : objV_G \to objV_H$ and $g_{vV} : valV_G \to valV_H$ and $g_E = g_{msgE} \cup g_{atrE}$ with $g_{mE} : msgE_G \to msgE_H$ and $g_{atrE} : atrE_G \to atrE_H$; the third component is a total function $g_L : Lab_G \to Lab_H$; and the last component is a total algebra homomorphism $g_A : Alg_G \to Alg_H$ which are weakly homomorphic, i.e. $g_V^* \circ s^G \leq s^H \circ g_E$, $g_V \circ t^G \leq t^H \circ g_E$, $g_L \circ lab^G \leq lab^H \circ g_E$ and $\mathcal{U}(g_A) \circ a^G \leq a^H \circ g_V$ (if an edge is mapped, its corresponding source, target and label must be mapped accordingly, and the same applies to attributes of mapped vertices). A morphism is called total if all components are total. The category of OB-graphs and partial OB-graph morphisms is denoted by **OBGra** (identities and composition are defined componentwise).

$$\begin{tikzcd}
E_G \arrow[r, "g_E"] \arrow[d, "s^G"'] \arrow[dr, phantom, "\leq"] & E_H \arrow[d, "s^H"] \\
V_G^* \arrow[r, "g_V^*"'] & V_H^*
\end{tikzcd}
\quad
\begin{tikzcd}
E_G \arrow[r, "g_E"] \arrow[d, "t^G"'] \arrow[dr, phantom, "\leq"] & E_H \arrow[d, "t^H"] \\
V_G \arrow[r, "g_V"'] & V_H
\end{tikzcd}
\quad
\begin{tikzcd}
E_G \arrow[r, "g_E"] \arrow[d, "lab^G"'] \arrow[dr, phantom, "\leq"] & E_H \arrow[d, "lab^H"] \\
Lab_G^* \arrow[r, "g_L^*"'] & Lab_H^*
\end{tikzcd}
\quad
\begin{tikzcd}
valV_G \arrow[r, "g_{vV}"] \arrow[d, "a^G"'] \arrow[dr, phantom, "\leq"] & valV_H \arrow[d, "a^H"] \\
\mathcal{U}(Alg_G) \arrow[r, "\mathcal{U}(g_A)"'] & \mathcal{U}(Alg_H)
\end{tikzcd}$$

The label set Lab_G is supposed to contain the names of object attributes and message parameters that are used in a specification. The labeling function lab^G assigns to each attribute edge a list of names of attributes, whereas the source function (s^G) assigns a list of vertices to each edge, that can be either object or value vertices. The restriction that both lists of labels and vertices related to the same edge have the same length assures that each attribute of an object has a name (label) and a corresponding value (object or data value). For example, the graph in Fig. 2(a) is given by the following tuple $C = (V_C, E_C, s^C, t^C, Lab_C, lab^C, Alg_C, a^C)$ where

$V_C = \{Class1, Class2\} \cup \{Nat, Bool\}$

$E_C = \{M1, M2\} \cup \{atr1, atr2\}$

$s^C = \{M1 \mapsto \langle\rangle, M2 \mapsto \langle Class2\rangle, atr1 \mapsto \langle Nat, Class2\rangle, atr2 \mapsto \langle Bool\rangle\}$

$t^C = \{M1 \mapsto Class1, M2 \mapsto Class1, atr1 \mapsto Class1, atr2 \mapsto Class2\}$

$Lab_C = \{atr1.1, atr1.2, atr2.1, parM2\}$

$lab^C = \{M1 \mapsto \langle\rangle, M2 \mapsto \langle parM2\rangle, atr1 \mapsto \langle atr1.1, atr1.2\rangle, atr2 \mapsto \langle atr2.1\rangle\}$

Alg_C contains two carrier sets $\{Nat\}$ and $\{Bool\}$

$a^C = \{Nat \mapsto Nat, Bool \mapsto Bool\}$

To distinguish different kinds of vertices and edges, we use the notion of *typed graphs* [50]: every graph is equipped with a morphism *type* to a fixed graph of types, called *class graph* here. The only requirement for a graph of types is that its attribute algebra is final, that is, an algebra in which each carrier set contains only one element. In practice, we use the name of the corresponding sort as the only element in a carrier set interpreting it.

Definition 3 (*OB-Graphs over C*). Given a specification *SPEC*, a graph C is called a **class graph** over *SPEC* if its attribute algebra is a final *SPEC*-algebra. An **OB-graph over** C is a pair $OG^C = (OG, type^{OG})$ where OG is an OB-graph called **instance graph** and $type^{OG} : OG \to C$ is a total OB-graph morphism, called the **typing morphism**.

A morphism between OB-graphs over C OG_1^C and OG_2^C is an OB-graph morphism $f : OG_1 \to OG_2$ such that $type^{OG_1} \geq type^{OG_2} \circ f$. If there is an isomorphism $f : OG_1 \to OG_2$, we say OG_1 and OG_2 are **equivalent**, denoted by $OG_1 \equiv_G OG_2$. The category of OB-graphs over C is denoted by **OBgraph(C)**.

Rules define how objects react when receiving messages. Each rule expresses how one particular message is treated (many rules may be necessary to describe all possible reactions to one message). By defining a rule as a morphism we assure a structural compatibility between left- and right-hand sides of rules. Since rules specify patterns of behavior of a model, it is natural that variables and expressions (terms) are used for the data part of the graph. Moreover, there may be equations involving variables associated to the rule. Formally, we use terms as attribute values, that is, we use the term algebra over the signature of the specification as attribute algebra (in the definition below, we equivalently use the term algebra over a specification without equations). In such an algebra, each carrier set consists of all terms that can be constructed using the operations defined for the corresponding sort, functions just represent the syntactical construction of terms (for example, consider a term t and an algebra operation op^A (corresponding to an operator op in the signature), in this case we would have $op^A(t) = op(t)$). Consequently, all terms are considered to represent different values in a term algebra, since they are syntactically different. The satisfaction of the equations is dealt with in the match construction — in the application of a rule.

Definition 4 (*Rule*). A rule is a pair (r, Eq) where Eq is a set of equations over the specification $Spec$ and $r : L \to R$ is an OB-graph morphism s.t.

1. L and R are finite;
2. There is exactly one message hyperedge in L and it is deleted: $\exists! e \in msgE_L$, called *trigger(r)* and $trigger(r) \notin dom(r_E)$;
3. Only attributes of the target of the message may appear in the Left-Hand Side (LHS): $(atrE_L = \emptyset) \vee ((\exists! e \in atrE_L) \wedge t^L(e) = t^L(trigger(r)))$;
4. Attributes are preserved: $\forall e \in atrE_L. \exists e' \in atrE_R. r_V(t^L(e)) = t^R(e')$;
5. Objects may not be deleted: $\forall o \in objV_L. o \in dom(r_V)$;
6. The algebra of r is a term algebra $T_{Spec'}(X)$ over the specification $Spec$ using a set of variables X;
7. Attributes appearing in the LHS may only be variables of X: $\forall v \in valV_L. a^L(v) \in X$;
8. r_L and r_A are identities.

$Rules(C)$ is used to denote the set of all rules over a class graph C.

Most of the restrictions imposed on rules basically guarantee that the approach follows the object-based paradigm. For instance, one message is deleted at each time (2), all attributes of each object are always present (4), and an object can only change its own attributes (3). Some restrictions are necessary to enable/ease translations and implementations of the rules. More specifically, these Left- and Right-Hand Sides of rules are finite (1) and the attribute terms of Left-Hand Sides are restricted to variables (6, 7 and 8). The restriction forbidding rules to delete (destroy) objects (5) is used to avoid problems, such as having dangling references to already destroyed objects. Nevertheless, it is still possible to use our approach and encode the destruction of objects in the model, for example by using an attribute to mark whether or not an object is active. This requires that two sets of rules must be defined. The first set specifies a "normal behavior" for active objects and the second set defines an "extended behavior" for dealing with non-active (destroyed) objects (e.g., similar to performing a garbage collection procedure).

To define an OBGG, we first define a class graph, modeling the types of objects, messages and attributes that may be present in the model. Then we define the behavior of the model using rules, and the possible initial states (graphs containing instances of the types in the class graph).

Definition 5 (*Object-Based Graph Grammar (OBGG/OBGG Model)*). An **object-based graph grammar** of **OBGG model** is a tuple $OBGG = (Spec, X, C, IG, N, n)$ where $Spec$ is an algebraic specification, X is a set of variables, C is a class graph over $Spec$, IG is an OB-graph over C, called the **start graph**, N is a set of rule names and $n : N \to Rules(C)$ assigns a rule to each rule name.

The behavior of an OBGG is obtained by applying the rules successively from a start graph. Each rule application deletes one message (the trigger of the rule) and may change the value of internal attributes, create new messages and/or objects. Formally, the effect of a rule application is obtained by a pushout in the corresponding category (typed object-based graphs). Considering the restrictions imposed on rules, the pushout construction just deletes the message edge specified in the left-hand side of the rule and adds the new vertices and message edges to the current state graph. Moreover, if attributes are changed, the corresponding attribute edge is deleted and re-created pointing to the new attribute value.

To define a match for a rule, we have to relate, additionally to the graph elements, the variables of the left-hand side of the rule to the actual values of attributes in the graph in which the rule shall be applied. Moreover, the match construction must ensure that all equations of the specification as well as the rule equations are satisfied by the chosen assignment of variables to values. This is achieved by first lifting the rule to a corresponding one, having a quotient term algebra as attribute algebra. This is a standard construction in algebraic specification. Then, the actual match includes an algebra homomorphism from this quotient term algebra to the actual algebra used in the graph, to which the rule is being applied. The existence of this homomorphism guarantees that all necessary equations are satisfied.

Definition 6 (*Derivation Step, Derivation*). Let $OBGG = (Spec, X, C, IG, N, n)$ be an OBGG, with $Spec = (SIG, Eqns)$, $(r : L \to R, Eq) \in n(N)$ be a rule, and G_1 be an OB-graph over C. A **match** for r in G_1 is a total morphism $m : \overline{L} \to G_1$ in **OBgraph**(**C**), where $\overline{L}$ is essentially the graph L, but having as algebra the quotient term algebra of the specification $Spec' = (SIG, Eqns \cup Eq)$ using variables in X. A **derivation step** $G_1 \overset{r,m}{\Longrightarrow} G_2$ using rule r and match m is a pushout of $\overline{r}$ and m in the category **OBgraph**(**C**). The class of all derivation steps of an $OBGG$ is denoted by $DerStep_{OBGG}$.

$$\begin{tikzcd} \overline{L} \arrow[r, "\overline{r}"] \arrow[d, "m"'] \arrow[dr, phantom, "(PO)"] & \overline{R} \arrow[d, "m'"] \\ G_1 \arrow[r, "r'"'] & G_2 \end{tikzcd}$$

A **derivation sequence** δ of $OBGG$, denoted by $\delta : G_0 \Longrightarrow^* G_n$, is a sequence of derivation steps $G_i \overset{r_i,m_i}{\Longrightarrow} G_{i+1}, i \in \{0, \ldots, n-1\}, n \in \mathbb{N}$, where $G_0 = IG$ and $r_i \in Rules$ for all $i \in \{0, \ldots, n-1\}$. If a derivation sequence is of length $n \neq 0$, IN^δ and OUT^δ denote the **input graph** of the first step of δ and the **output graph** of the last derivation step of δ, i.e. $IN^\delta = G_0$ and $OUT^\delta = G_n$. If δ is the empty derivation sequence, then $OUT^\delta = IG$. The **rule-application sequence** corresponding to a derivation, denoted by $rapplic(\delta)$, is the list of all rule names that were applied in δ, in order of occurrence. The **semantics of an** $OBGG$ is the class of all derivation sequences, denoted by $SemOBGG$.

The semantics described above includes all possible sequential computations that can be generated by a graph grammar, providing an interleaving semantics for this specification method. True concurrent semantic models, like processes [50] or unfolding [51], are usually based on classes of equivalent sequential computations. The true concurrency semantic models for a graph grammar describe the same set of computations as the interleaving model, but in a more compact way. In this paper we consider the class of derivation sequences because this semantic model is closer to the one typically used in PROMELA.

4. Verification of OBGG using SPIN

Models specified using OBGG can be transformed to (semantically equivalent) PROMELA models (see details in Section 7) and verified using the model checker SPIN [24]. One important aspect of this transformation based method is that it keeps the same level of abstraction for the user whilst specifying and model-checking the system. The properties (to be verified), as well as the generated counter-examples (for properties that have a false result during verification), are expressed in terms of OBGG abstractions and not in terms of the generated PROMELA model (or code). This section describes a methodology for the specification and verification of properties of OBGG models. We use the dining philosophers problem described in the previous section to illustrate the methodology. In the next section an approach for specifying properties over OBGG models is presented. Section 4.2 complements the methodology for verifying OBGG models introducing an approach for generating graphical counter-examples, out of PROMELA traces, for specified properties that have a false result in the verification process.

4.1. Property specification

Two complementary approaches are commonly used for specifying properties of models, one based on events and another on states [52]. In order to use these approaches within the context of OBGG, the applications of rules can be seen as events, whereas the graphs, obtained from the application of rules, can be viewed as states.

4.1.1. Properties about events

Whilst analyzing a graph grammar, it is highly desired to state properties about possible rule applications of that grammar. To accomplish this, the notion of events is used. In this case, a rule application is an event of the model and properties relating different events in time are stated in Linear Temporal Logic (LTL) [53]. Using events also allows one to deal with dynamic creation of objects. Since new objects are instances of existing classes in the model, the set of possible events (rules that can be applied) remains the same.

SPIN is a state-based model checker. Therefore, events and properties over events have to be encoded into suitable states and properties over states. In [54] the notion of events for LTL properties was presented in two parts: (i) the definition of a desired value for a global variable; (ii) the LTL formula describing the change of a global variable to the value defined in (i). Step (ii) marks the occurrence of a state change that is considered an event of interest that has to be investigated. In the case of graph grammars, rule applications are the events of interest. Our approach is to define an additional global variable that is modified whenever a rule application is accomplished (variable *event_RuleName*). Whenever a rule is applied, the name of the rule is assigned to this variable — the structure of this name follows *rule_NameOfTheClass_NameOfTheRule*. In the PROMELA code, the global variable *event_RuleName* (see Section 7) is defined having as type an enumeration of names of the OBGG rules that compose the model. Events for every rule application are defined by LTL formulas using this added variable during the model's transformation, where the change of its value is modeled using the *next* temporal operator.[3]

The use of the *next* temporal operator may result in properties that are not closed under stuttering.[4] This may prevent the use of partial order reduction techniques in SPIN. To allow the use of existing techniques, for state space reduction, we adopt an approach to obtain closed under stuttering formulas constructively, by using property specification patterns [55,52,54]. Since expressing properties using temporal logic is usually a complex task, this approach has the additional benefit of guiding the user during property specification. We follow the notation proposed in [54], where the occurrence of an event is expressed using the symbol $\uparrow$ before the name of the event. To illustrate the specification of properties defined for OBGGs, we now state formally Property 1 discussed in the previous section for the dining philosophers problem.

Property 1 (*Deadlock Freedom for the Dining Philosophers Problem*)**.** *To show deadlock freedom, we can analyze if it is always possible that rule Eat is applied (see Fig. 3 (c)). Event $\uparrow$Eat denotes the application of rule Eat. The LTL formula for this property becomes: ($\Box \Diamond \uparrow$Eat) — specifying that event $\uparrow$Eat can always happen in the future.*

In order to describe this property in terms of the transformed PROMELA model, which can be analyzed using SPIN, we have to follow the two steps (i) *and* (ii) *presented earlier. More concretely, for this example, we have:*

[3] LTL notation used: $\Box$ — always; $\Diamond$ — eventually; X — next; U — (strong) until.

[4] Intuitively, closure under stuttering requires a property to be insensitive to successive repetition of any of the states in a sequence.

(i) *the desired value (application of rule Eat) is assigned to the global variable event_RuleName. In SPIN's syntax:*

$$\#define\ phil\_eat\ (event\_RuleName == rule\_Phil\_Eat)$$

(ii) *the LTL formula describing (↑Eat) (the occurrence of an Eat event) is defined as the change of value of the global variable event_RuleName:*

$$(\neg\, phil\_eat \wedge X\ \ phil\_eat)$$

During the verification process, the deadlock freedom property was not satisfied by the symmetric configuration (where deadlock is possible) and was satisfied by the asymmetric configuration of the problem. The obtained counter-example from SPIN corresponds to a file, where lines of the PROMELA model have variables substituted by values of the current state of the model. This counter-example is used in Section 4.2, in order to illustrate the approach for generating graphical counter-examples that are meaningful to OBGG users.

4.1.2. Properties about states

Besides specifying properties about events, it would be desirable to express properties considering the internal state of objects in the model. In order to accomplish that, we extend the notion of events (previously described) to consider internal attributes of objects. Therefore, instead of only storing the name of the applied rule, we allow any internal attribute to be stored. More specifically, for each attribute that is used for verification purposes, we introduce a global variable with name structured as *event_NameOfTheClass_NameOfTheAttribute* in the PROMELA model. Whenever a rule is applied to an object of that class, the value(s) of the attribute(s) is(are) assigned to the respective global variable(s) (if any) after the rule application. This way, events become more specific, they may describe changes of attributes of objects of the model. Now we show the formalization of Property 2 presented in the previous section.

Property 2 (*Mutual Exclusion for the Dining Philosophers Problem*)**.** *In order to show that mutual exclusion is preserved, it is necessary to show that whilst a philosopher starts eating (rule* Eat*) and releases the forks (rule* Release*) no neighboring philosopher starts eating (rule* Eat *again). As described in [54], this property is captured by the property pattern* (i) *"absence of an event" in a scope* (ii) *"between two other events". In terms of the dining philosophers problem, we specify the events* ↑*Eat_Phil*1 *and* ↑*Eat_Phil*3 *(the neighboring philosophers) for* (i) *and the events* ↑*Eat_Phil*2 *and* ↑*Release_Phil*2 *for* (ii) *(the philosopher entering and leaving the mutual exclusion). This culminates in the following temporal logic formula that has to be verified:*

$$\Box(((\uparrow Eat\_Phil2) \wedge \Diamond(\uparrow Release\_Phil2)) \rightarrow\ (\neg(\uparrow Eat\_Phil1 \vee\ \uparrow Eat\_Phil3)\ U\ (\uparrow Release\_Phil2)))$$

Intuitively, the left-hand side of the implication selects executions where events ↑*Eat_Phil*2 *and* ↑*Release_Phil*2 *take place (in this order). For these executions, the right-hand side requires that, until the event* ↑*Release_Phil*2 *takes place, the events* ↑*Eat_Phil*1 *or* ↑*Eat_Phil*3 *do not take place.*

When specifying these events, it is necessary to know the value of the internal attribute id of the philosophers, to distinguish between philosophers 1, 2 *and* 3*. Following the discussion presented previously, a global variable representing the id value is introduced in the PROMELA model. More concretely, we show how the event* ↑*Eat_Phil*2 *is defined in the PROMELA model*[5]*:*

(i) *The desired values for the added global variables are specified. In SPIN's syntax:*
#define phil_eat2 (event_RuleName == rule_Phil_Eat ∧ event_Phil_id == 2)
(ii) *The LTL formula describing (*↑*Eat_Phil*2*) the change of value of the global variables:*
$(\neg\, phil\_eat2 \wedge X\ \ phil\_eat2)$

The verification for this formula in both symmetric and asymmetric configurations of the dining philosophers delivered a true result, that is, both models satisfy the mutual exclusion property.

4.2. Generation of counter-examples

One widely accepted graphical form to view the execution of distributed systems has its basis on the exchange of messages between processes. This approach consists in defining a time-line for each process in the model and by showing the messages (via labeled arcs) being sent/received by processes.

Since OBGG has its focus on the specification of distributed systems, the use of a graphical representation similar to the one described has the advantage of being intuitive for users that work with the abstraction of message passing. However, showing only that the exchange of messages does not capture another important abstraction of OBGG, the application of rules. In order to consider the application of rules in a graphical execution view of OBGG models, we add information about rule applications to the time-line of each object in the model. This information contains the name of applied rule(s) and is added whenever a rule is executed by an object.

Fig. 5 shows an example of this approach, using the counter-example obtained from the verification of Property 1 (deadlock freedom — discussed in Sect. 4.1.1) for the symmetric solution of the dining philosophers problem. For each object

[5] The events ↑*Eat_Phil*1, ↑*Eat_Phil*3, and ↑*Release_Phil*2 are defined analogously.

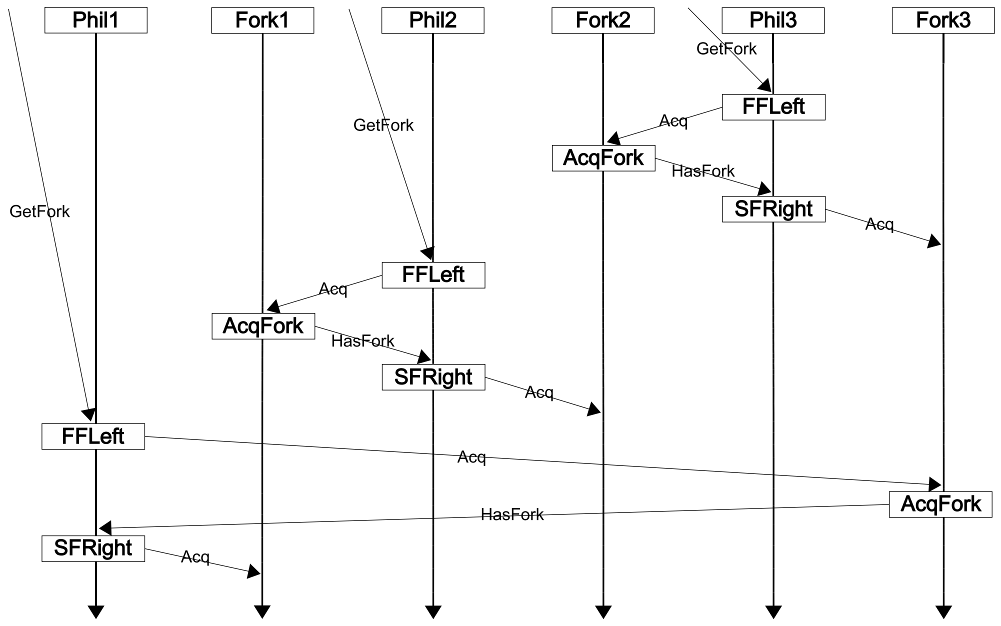

Fig. 5. Graphical view of the counter-example obtained from the verification.

of the model, a time-line is defined, being identified by a box at the top with the name of the identity of this object. Messages are shown as labeled arcs. Rules executed by objects are presented in boxes along the object's timeline. This execution shows a situation where the objects Phil1, Phil2, and Phil3 acquire the left fork. After acquiring the left fork, all objects Phil1, Phil2, and Phil3 try to acquire the right fork, entering in a deadlock situation. By filtering the counter-example file obtained from SPIN [8], this graphical counter-example is generated automatically in our verification tool [56].

5. PROMELA

PROMELA [57] is the input language of the SPIN model checker [24]. This section introduces the PROMELA subset used in our work. Fig. 6 describes the syntax for the considered PROMELA fragment. The main restriction of this subset, with respect to the full PROMELA language, is that we consider neither `goto` statements nor labels for commands, as these are not needed in our transformation from OBGG to PROMELA. As usual, each non-terminal symbol describes a set of words (all that can be derived from it using the rules of the grammar). We call a program written in PROMELA a **PROMELA model**.

The language has a *C-like* syntax and constructs for receiving and sending messages similar to Communicating Sequential Processes (CSP) [58]. The basic PROMELA types are `bit`, `bool`, `byte`, `short`, and `int`. PROMELA also supports the declaration of arrays of basic types and a type message channel. Processes in PROMELA can be created statically or dynamically (`proctype` keyword). There is a special purpose process, called `init`, used to initialize a model. Processes can exchange information through message channels (`chan` keyword) or global variables (variables declared outside the scope of processes). Message channels can be synchronous (the buffer of the message channel has size 0 — cannot hold messages) or asynchronous (the buffer of the message channel has size N ($N > 0$) — can hold N messages). Message channels are typed. It is required to declare explicitly the types of variables a channel might receive. Moreover, PROMELA offers several pre-defined functions, for example, to check if a channel is not full (`nfull(channel)`), how many messages an asynchronous channel has in its buffer (`len(channel)`), and others [57].

Non-determinism is modeled using 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 entries is chosen non-deterministically to execute. It is also possible to define atomic structures (`atomic { ... }`) in a model, i.e., a sequence of statements that must be executed by one process without interleaving with the execution of other processes. However, if the process blocks for any reason during the execution of an atomic block (for example, if all guards in a guarded command are not enabled), the process is suspended and the block loses its atomicity, interleaving its execution with other processes. The language supports the definition of enumeration types (`mtype` keyword). It is also possible to insert assertions in a PROMELA model (`assert(Expr)`). When simulating or verifying a model, the SPIN model checker evaluates the expression (Expr) defined in the assertion to true or false, each time the statement is executed. If Expr evaluates to false, an error is generated at run time and the simulation or verification being performed stops its execution.

Program	::=	Unit | Unit Program
Unit	::=	`proctype` Name`( )` Body | `proctype` Name`(`DeclLst`)`Body | `init` Body | OneDecl | MType
DeclLst	::=	TYPE Name | TYPE Name`;` DeclLst
Body	::=	`{`Seq`}`
Seq	::=	Step | Step`;`Seq
Step	::=	OneDecl | Stmt
Stmt	::=	SStmt | CStmt | BStmt
SStmt	::=	VarRef`:=`Expr | Expr | VarRef`?`MArgs | VarRef`!`MArgs | `run` Name`()` | `run` Name`(`ArgLst`)`
CStmt	::=	`if` Options `fi` | do Options od | `break`
Options	::=	`::` Seq | `::` Seq Options
ArgLst	::=	Expr | Expr`,` ArgLst
MArgs	::=	ArgLst | Expr`(`ArgLst`)`
BStmt	::=	`atomic{`Seq`}`
OneDecl	::=	BType IVarLst | `chan` VarDecl = ChInit | `chan` VarDecl
BType	::=	`bool` | `byte` | `int` | `short`
Type	::=	BType | `chan` | `mtype`
IVarLst	::=	IVar | IVar`,` IVarLst
IVar	::=	VarDecl | VarDecl = Expr
VarDecl	::=	Name | Name`[`Const`]`
ChInit	::=	`[`Const`] of {`TypeLst`}`
TypeLst	::=	Type | Type`,` TypeLst
MType	::=	`mtype{`NameLst`}`
Expr	::=	Const | Name | `(`Expr`)` | Expr BinOp Expr | UnOp Expr
NameLst	::=	Name | Name`,`NameLst
BinOp	::=	`+` | `-` | `*` | `/` | `%` | `&` | `|` | `<` | `>` | `==` | `!=`
UnOp	::=	`~` | `-` | `!`

Fig. 6. Fragment of PROMELA syntax.

5.1. Formal semantics

The semantics of PROMELA used here is a revised version from the semantics presented in [59]. Since we do not use `goto` statements and labels for commands, we simplified the basic structure defining local states. We also revised the definition of several rules, for example the rules regarding atomic blocks were simplified. An atomic block can only start executing if its first statement can be executed. In PROMELA, it is possible to stop the execution of an atomic block (and let other process run) if the process that is executing atomically reaches a state in which it cannot proceed. However, while executing the code generated from an OBGG, an atomic block, once started, can always terminate, and therefore the provided semantics is suitable.

Notation. *Let $\boldsymbol{A}$ be a set (an alphabet), a word (or string) w over $\boldsymbol{A}$ is a finite sequence of elements $a_i \in \boldsymbol{A}$ written $w = a_1 \cdot a_2 \cdots \cdot a_n$, where the length of w is given by $|w| = n$. The empty word is denoted by ε. The set of all words over $\boldsymbol{A}$ is written $\boldsymbol{A}^*$. Two words v, w can be concatenated via the operator “ ”. For a function $f : M \to M', f' := f[a \mapsto b]$ is equal to f except that $f'(a) = b$. Given a set M, the set $M_\perp$ denotes $M \cup \{\perp\}$. Given a function $f : A \to B_\perp, f_\perp$ denotes the undefined function, i.e. a function where for all $a \in A, f_\perp(a) = \perp$.*

The semantic model of PROMELA is described as a labeled transition system (LTS) defined by a set of structural operational semantics (SOS) rules. The types used to define the states of this LTS are defined in Section 5.1.1. The definition of the states and transitions is split into two main parts: behavior of single processes (Section 5.1.2) and behavior of the complete system (Section 5.1.3).

5.1.1. Data types and values

The basic types allowed in PROMELA are the basic types `bool`, `byte`, `short` and `int`. There is also a basic type called `chan` that is used to declare channels (channel identifiers are denoted by natural numbers). Arrays of basic types can also be defined by using the `arr` constructor. Channels are lists of messages. Each channel in PROMELA is typed, such that only messages of specific types are allowed in each channel. The type of a message is a tuple of basic and array types. In the following we use the notation (for the corresponding formal definitions, see Appendix A):

Type	denotes the set of all basic and array types;
Val	denotes the set of all values of basic and array types;
Asgn	denotes the set of all assignment functions $f : \mathsf{Name} \to \boldsymbol{Type} \times \boldsymbol{Val}$ of pairs of type and current value to variable names;
MsgType	is the set of all (finite) tuples of types in ***Type*** defining all possible types of channels;
Msg	is the set of all (finite) tuples of values in ***Val*** describing all possible messages;
Channel	is the set of channels, given by triples $(T, msgs, max)$ containing a channel type $T \in \boldsymbol{MsgType}$, a list of messages $msgs \in \boldsymbol{Msg}^*$ of this channel type and the bound $max \in \mathbb{N}_\perp$ of this channel (the length of the list of messages must be less or equal to max).

5.1.2. (Local) process behavior

The (local) state of a process consists of the following: the (yet unexecuted) program body, values of local variables, values of global variables, channels and a continuation stack (this stack is used to define the semantics of do statements).

Definition 7 (*Local State*)**.** The local state of a PROMELA program is a tuple $(\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$ where

$\pi : \mathsf{Seq} \cup \{error\}$ is the (yet) unexecuted process code[6];
$\mathcal{L} : \mathsf{Name} \to (\boldsymbol{Type} \times \boldsymbol{Val})$ is a local variable assignment;
$\mathcal{G} : \mathsf{Name} \to (\boldsymbol{Type} \times \boldsymbol{Val})$ is a global variable assignment;
$\mathcal{C} : \mathbb{N} \to \boldsymbol{Channel}$ is a channel definition;
$\gamma : \mathsf{Seq}^*$ is a continuation stack.

The semantics of process execution can be defined by an LTS. The states are local states defined above, and each transition is labeled with the PROMELA declaration or statement that executed, giving raise to the corresponding local state change. Transitions can also be labeled with expressions since expressions can be used as statements according to the syntax of PROMELA. In the event that an atomic block starts, the corresponding transition is labeled just by `atomic`. The label τ is used whenever a do statement starts executing (it is unraveled to an `if`). Violation labels are used to mark transitions that resulted in execution errors (this is the case, for example, when an expression used as a statement evaluates to zero). The SOS rules that define these transitions can be found in Appendix A.2.

Definition 8 (*Local Process LTS*)**.** Given global declarations of variables and channels $\mathcal{G}$ and $\mathcal{C}$, respectively, and a PROMELA process $p =$ "`proctype` *pname* `(...){`π`}`", the corresponding transition system is $LTS_p = (S_L, S0_L, L_L, \to_L)$, where

$S_L = \mathsf{Seq} \times \boldsymbol{Asgn} \times \boldsymbol{Asgn} \times \boldsymbol{Channel} \times \mathsf{Seq}^*$ is the set of local states;
$S0_L = (\pi, \mathcal{L}_\perp, \mathcal{G}, \mathcal{C}, \varepsilon) \in S_L$ is the initial state of the process;
$L_L = \mathsf{Step} \cup \{\mathtt{atomic}, \mathtt{end\_atomicviolation}, \tau\}$;
$\to_L$ is defined by the SOS rules in Figs. A.9–A.11 in Appendix A.2.

5.1.3. Global behavior

To build the global state of a PROMELA model, we need additional structures to keep track of process definitions and instantiations, as well as to control the atomic behavior of the processes.

Process Definition: Each `proctype` declaration defines a process type. The *process type* consists of the process body $\pi \in \mathsf{Seq}$ and a parameter function $p : \{1, \ldots, n\} \to \mathsf{Name} \times \boldsymbol{Type}$, which represents the Name and the type of the i-th parameter. Let $\mathcal{P}$ be the set of all parameter functions, that is, $\mathcal{P} := \{\{1, \ldots, n\} \to \mathsf{Name} \times \boldsymbol{Type} \mid n \in \mathbb{N}\}$. A *process definition* is $pdef : \mathsf{Name} \to (\mathsf{Seq} \times \mathcal{P}) \cup \{\perp\}$, mapping each process Name to its process type (or bottom if the Name does not denote a process). The set of all process definitions is denoted by ***PDef***.

Process Instantiation: It is defined by a tuple $(\pi, \mathcal{L}, \gamma)$, where π is the current program fragment, $\mathcal{L}$ is a local assignment function and γ is a continuation stack. A *process instantiation function act* $: \mathbb{N} \to \mathsf{Seq} \times \boldsymbol{Asgn} \times \mathsf{Seq}^*$ maps each (created) process identifier to its process instantiation. The set of all process instantiation functions is denoted by ***PInst***.

Atomic Behavior: If there exists a process executing atomically, then a flag ***at*** is set to the index of the process, otherwise, ***at*** $=\perp$.

[6] The value *error* denotes execution errors. However, it is outside the scope of this paper to give details on how such errors are handled in PROMELA. It is enough for the reader to understand that they lead to abortion of execution.

Definition 9 (*Global State*). The global state of a PROMELA model is a tuple $(\pi, \mathcal{G}, \mathcal{C}, pdef, act, at)$ where

π : Unit is the (yet) unexecuted program code;
$\mathcal{G}$: $\textbf{\textit{Name}} \to \textbf{\textit{Type}} \times \textbf{\textit{Val}}$ is a global variable assignment;
$\mathcal{C} : \mathbb{N} \to \textbf{\textit{MsgType}} \times \textbf{\textit{Msg}} \times \mathbb{N}_\perp$ is a channel definition;
$pdef$: $\textbf{\textit{PDef}}$ is a function assigning each process name to its definition, which is composed by a process code and a mapping of each parameter index to its name and type;
act : $\textbf{\textit{PInst}}$ maps each process identifier (natural number) to its instantiation. It consists of the yet unexecuted process code, a local assignment function and a continuation stack;
$at : \mathbb{N}_\perp$ is the identifier of the process currently executing an atomic block, or the value $\perp$ if no process is executing atomically.

If $act(i) = (\pi, \mathcal{L}, \gamma)$, then $(\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$ is the local state of the process with identifier i.

The semantics of a PROMELA model (PROMELA program) is the following: first, the code of the main program is executed, generating the structures containing all global variables, channels, and a process definition and a table of active processes containing just one process, the `init` process; once all these definitions are processed, the `init` process may start running, behaving according to the local semantics definition. The `init` process may then create other processes that may also start executing. When one of the processes that is executing enters in an atomic block, the flag at is set to the identifier of this process, and from then on, only statements of this process may execute until the end of the atomic block (when the flag is reset). This semantics is described by a labeled transition system in which the labels correspond to definitions of processes (`proctype`), of the initial process (`init`), to the creation of new processes (`run`), to the beginning or end of atomic blocks (`atomic` and `end_atomic`) or to statements describing local execution of processes (defined in the local process behavior).

Definition 10 (*Semantics of PROMELA*). The semantics of a PROMELA model Π is given by an LTS denoted by $SemPROM(\Pi) = (S, S0, L, \to)$, and defined as follows

$S = \text{Unit} \times \textbf{\textit{Asgn}} \times \textbf{\textit{Channel}} \times \textbf{\textit{PDef}} \times \textbf{\textit{PInst}} \times \mathbb{N}_\perp$ is the set of global states;
$S0 = (\Pi, \mathcal{G}_\perp, \mathcal{C}_\perp, pdef_\perp, act_\perp, \perp) \in S$ is the initial state;
$L = \text{Unit} \cup L_L$;
$\to \subseteq \mathcal{S} \times L \times \mathcal{S}$ is the transition relation defined by the SOS-Rules depicted in Fig. A.12 in Appendix A.3.

Paths of a transition system are sequences of transitions.
Notation. *Given an LTS $TS = (S, I, L, \to)$, for each transition $t = (i, l, f) \in \to$ we denote its initial state i, its label l and its final state f by $IN(t)$, $LAB(t)$ and $OUT(t)$, respectively. The set of paths of TS, denoted by Path(TS), is defined by: $Path(TS) = \{\sigma \in (S \times L \times S)^* \mid \sigma = \varepsilon \vee (IN(\sigma_1) = S_0 \wedge \forall j \in \{1..|\sigma| - 1\}.OUT(\sigma_j) = IN(\sigma_{j+1})\}$. If a path σ is finite of length $n \neq 0$, its output state OUT^σ is $OUT(|\sigma_n|)$.*

6. Correctness of transformations to PROMELA

The main aim of transforming a model in a given language to PROMELA is to perform model checking using the SPIN tool. Depending on the approach, the properties to be checked can be expressed naturally, in terms of the original model (as we do in our approach, see Section 4), or in terms of the transformed PROMELA model. Note that SPIN is a state-based model checker, and thus properties are checked over states of the transition system of the PROMELA model. As we already discussed in Section 4, it is possible to encode transitions or events into states in order to use a state-based model checker to reason about sequences of events. This notion of using sequences of events to express properties is fundamental to proving the correctness of the transformation. In the following we define a series of general steps that can be carried out to prove that the transformation of a model in a given language to PROMELA is correct. The notion of correctness adopted here is that the original model and its transformed PROMELA model give rise to the same sequences of events. The steps below are defined in a general way, and are instantiated in the next section for the case of transforming OBGG models to PROMELA.

Requirements. As a basis for the proof of correctness, we require:

- The (formal) syntax of a language $\mathcal{L}$;
- The (formal) semantics of $\mathcal{L}$, in terms of sequences of labeled events;
- The formal syntax and semantics for all PROMELA constructs that are used in the transformation.

The semantics of $\mathcal{L}$ shall be given in such a way that the atoms used to express the desired properties correspond to the labels of the events because these labels will become the global variables that SPIN will use to verify the properties.

Step 1 — Define the transformation from $\mathcal{L}$ to PROMELA. In this step, all constructs of $\mathcal{L}$ shall be transformed to PROMELA constructs, such that a model in $\mathcal{L}$ can be transformed into a PROMELA model. This transformation is based on the syntax of $\mathcal{L}$ and PROMELA. When defining this transformation, it is important (from the proof of correctness point-of-view) that the global variables of the PROMELA model correspond to the atoms used to express the desired properties — consequently, they also correspond to the labels of the transitions defining the semantics of $\mathcal{L}$.

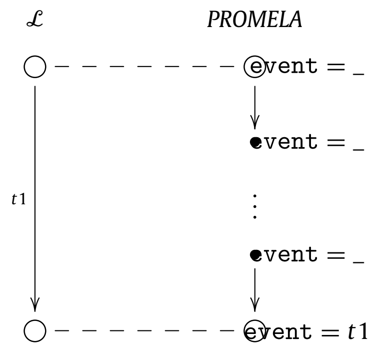

Fig. 7. Relating $\mathcal{L}$ events to PROMELA transitions (paths).

Step 2 — Define relations between states. Now we have to relate the states between the models in $\mathcal{L}$ and its PROMELA transformation. This relation shall be defined in both directions (from $\mathcal{L}$ to PROMELA and vice versa). However, since usually there is a change of granularity when transforming specification languages to PROMELA, it might be the case that not all PROMELA states can be transformed back to $\mathcal{L}$ states. In such cases, it is necessary to characterize which PROMELA states correspond to $\mathcal{L}$ states. We call such states *well-formed PROMELA states*.

Step 3 — Define relations between events and transitions. In order to show that the transformation preserves the behavior of the original model, the first step is to prove that whenever there is an event in the semantics of $\mathcal{L}$, there is a sequence of transitions of the corresponding PROMELA model starting and ending in the corresponding states. Fig. 7 illustrates this step. Here, we used a global variable called `event` that records the name of the last occurring event (*well-formed states* are depicted as white circles). Ideally, this whole sequence should occur in an atomic PROMELA block — to prevent interleaving with sequences implementing other events of $\mathcal{L}$. We must also characterize sequences implementing events of $\mathcal{L}$. Then, we have to show that whenever there is such a sequence in the PROMELA model, it could be possible to have the corresponding event in the original $\mathcal{L}$ model.

Step 4 — Prove correctness. The aim of the transformation is to prove properties about the original model. The proofs are performed over the sequences of transitions or paths of the corresponding PROMELA model, using global variables as atoms. Therefore, we have to guarantee that the PROMELA paths correspond exactly to the sequences of events of the original model. If this is true, then any formula using these atoms (labels of events) that are valid/invalid for the PROMELA model are also valid/invalid for the original $\mathcal{L}$ model. To obtain the correctness results, the proof must be done in both directions: (i) show that any sequence of events of the $\mathcal{L}$ model is also found in the transformed PROMELA model; and, (ii) show that any transition path that starts and ends in *well-formed states* corresponds to a sequence of events that also exists in the original $\mathcal{L}$ model. Note that this notion of correctness does not correspond to (some kind of) bisimulation between the (transition systems of the) models because it is usually not possible to relate all PROMELA states with the states of the more abstract model and this total relation is needed to define classical bisimulation relations, like strong or weak bisimulation [60].

7. Transformation of OBGG models to PROMELA models

The transformation of OBGG to PROMELA defines how the abstractions of OBGG are expressed in PROMELA. Initially, a description of the transformation is presented in Section 7.1, followed by the formal definition of the transformation in Section 7.2. This corresponds to Step 1 described in Section 6. The other steps are followed in Section 8.

7.1. Transformation overview

The general idea of the transformation is to map OBGG constructs to PROMELA constructs capable of preserving the semantics of OBGG. Below we discuss in a glance how OBGG constructs are mapped to PROMELA, explaining the 4 main components of the transformation which are then formalized in Section 7.2.

An OBGG model is composed of concurrent objects (**INIT**) that encapsulate state and react to incoming messages (**MSGS**) according to the rules (**RULES**) defined for the corresponding classes (**PROC**s). PROMELA offers the abstraction of processes that execute concurrently and use channels to exchange messages. Given this similarity, it is intuitive to map OBGG classes to PROMELA process definitions. The class state is then mapped to internal process variables. Each process instance (modeling an object) has a unique input channel associated to it, through which all messages to that process are sent. This channel represents the reference to the respective OBGG object. The body of such a process is a loop evaluating and possibly applying rules. The main components of the translation of an OBGG to PROMELA are:

INIT: An OBGG initial graph is mapped to a PROMELA init process: each object instance is mapped to the instantiation of a process and creation of its respective input channel; and each message in the initial graph is mapped to a message sent to the respective process input channel. Given this initial setting, each process starts reacting to incoming messages and the computation then unfolds.

INIT (Definition 12) describes the translation of an OBGG initial graph into a PROMELA init process. This is achieved by creating *object process channels*, passing these channels as parameters in the instantiation of *object processes*, and sending initial messages through the defined *object process channels* to activate those objects.

MSGS: Since in OBGG there is no notion of order among messages and PROMELA channels obey FIFO, each process (*object processes*) will transfer messages from its input channel (*object process channels*) to an internal structure that allows messages to be evaluated and consumed in a non-ordered way. This structure, called the *object process buffer*, is defined together with two operations for writing (WRITE) and non-deterministically reading (READ) messages to/from it. The component **MSGS** is described in Definition 13.

RULES: The left-hand side of an OBGG rule describes the trigger message (together with its parameters), as well as the possible internal states that enable application of this rule. The right-hand side specifies the modifications done upon application of this rule, which include: the consumption of the trigger message; change of state in the object; generation of new messages and creation of new object instances. This semantics can be captured properly using PROMELA guarded commands: a guard evaluates available messages and current object state to detect whether the rule may be applied; and the action specifies message consumption, the state change in the process variables, and possible generation of new messages (writing in channels) and new processes. The non-deterministic choice of rule to be applied in OBGG is mapped to a non-deterministic choice among the several guarded commands that represent rules. **RULES** (Definition 16) shows how rules of a given class are transformed to PROMELA code.

PROC: Definition 17 describes how OBGG classes are mapped to static definitions of PROMELA processes. Each class is mapped to one process definition. An OBGG object corresponds to a process instance, which we call *object process*. The body of such a process is a loop selecting a message to consume, evaluating and possibly applying rules. When a message is received through the *object process channel* it is moved to the *object process buffer* to be consumed in a non-ordered way as discussed in **MSGS**. When a message is selected, one of the possible enabled rules by that message is applied, as discussed in **RULES**.

Storing messages in PROMELA channels gives raise to a problem: PROMELA channels are bounded while there is no bound in the number of messages that can be waiting to be handled by an object in an OBGG. To handle this situation, the translation process has as input the maximum buffer size. This is a measure of the expected maximum number of simultaneous messages awaiting to be consumed by an object instance, and varies from model to model. With this value, the translation introduces assertions that check that neither the input channel of an object nor its internal buffer become full during verification. This is an auxiliary measure that guarantees that the translated model does introduce synchronizations which are not present in the OBGG. In case such an assertion becomes false, the analysis is stopped and the user is properly warned about the problem. The user may then redefine the buffer size or change the model.

Finally, since verification is done by analyzing traces of events, we defined a global variable in PROMELA, called *event_RuleName*, to record which event has just happened. The events that are the basis of the OBGG semantics are the derivation steps, and these are labeled with the names of the rules that were applied. Therefore, the variable *event_RuleName* records the name of the last applied rule.

7.2. Transformation definitions

The formal transformation of the OBGG constructs to PROMELA are now presented. The transformation is described using a top-down approach, starting with Definition 11. We use the following notation: (i) functions and variable names in italics are references to functions and components of the OBGG being transformed (defined in Section 3.3), (ii) PROMELA code is within quotes, (iii) blocks starting with $\forall e \in S$ denote translations that must be instantiated with all elements of S (the block is marked by a vertical bar), (iv) auxiliary functions used in the translation are underlined, and explained in Table B.1 in Appendix B. To increase readability, all auxiliary functions do not mention explicitly that they access the source OBGG, that is, we write $\text{R}\textsc{ules}_{class}$ instead of $\text{R}\textsc{ules}(GG, class)$. Although the translation function generates a PROMELA program, that is, a string, for the sake of clarity we omit functions to concatenate strings and to make a type conversion (for example, of integers into strings).

Definition 11 (*Transformation of an OBGG model to a PROMELA model*)**.** Let $GG = (Spec, X, C, IG, N, n)$ be an OBGG model. A transformation of GG to a PROMELA model is given by the function $Transl_{OBGG} : \mathbb{N} \times GG \rightarrow STRING$, where the first parameter is the length of buffers used in the transformed PROMELA model. $Transl_{OBGG}$ is defined as follows:

```
$Transl_{OBGG}(bsize, GG) = $ HEADER INIT OPROCESSES
HEADER = “#define BSIZE  ” $bsize$
             “mtype = {” $\underline{\text{LMSG}}$ “, ” $\underline{\text{LRULE}}$ “}; ”
             “mtype event_RuleName;”
OPROCESSES =   $\forall class \in obj_{V_C}$
             | $\text{MSGS}_{class}$ $\text{RULES}_{class}$ $\text{PROC}_{class}$
```

where INIT, MSGS_{class}, RULES_{class} and PROC_{class} are presented, respectively, in Definitions 12, 13, 16 and 17.

According to Definition 11, the components of a transformed OBGG model are (line 1): Header; Init; and OProcesses. As discussed before, OBGG messages are transformed to PROMELA messages sent through asynchronous message channels that have a bounded size. Therefore, during the transformation process, it is necessary to give as parameter the input size of these channels, which becomes a constant size in PROMELA (line 2). The functions $\underline{\text{Lmsg}}$ and $\underline{\text{Lrules}}$ return, respectively, a list of all the messages and rules of the OBGG model *GG*. The format used for each message name is msg_*class_msgname* and for each rule is rule_*class_rulename*, where *class* is the class's name (which receives the message or owns the rule), *msgname* is the message's name, and *rulename* is the rule's name. As depicted in line 3, both the list of messages and rules are declared in a PROMELA mtype definition (as enumerations). These names are used in the model to: (a) represent the message sent between *object processes*; (b) explicitly define in the PROMELA model which rule has been applied, by setting the contents of the global variable event_RuleName (declared in line 4).

The Init definition corresponds to an init PROMELA process that represents the OBGG initial graph. The definition of OProcesses (line 5) contains the transformation of all classes and respective rules for an OBGG model. Each of such object processes is composed of the definitions for the non-deterministic selection of messages and writing over the object's message buffer (Msgs), a conditional structure describing the OBGG rules (Rules), and a PROMELA process type representing the transformed class (Proc).

7.2.1. Initial Behavior

Definition 12 presents the transformation of an initial graph to a PROMELA init process (lines 1 to 6). Basically, an init process is composed of two parts. First, the *object process channels* of the transformed objects that compose the initial graph are defined and the concurrent execution of *object processes* representing OBGG objects is started (line 3). Second, the transformation of OBGG initial messages is defined, where initial messages are sent through the *object process channels* (line 4). Auxiliary functions $\underline{\text{ParType}_{class}}$, $\underline{\text{Atrib}_{obj}}$ and $\underline{\text{MsgParVal}_{msg}}$ are used to generate lists of message parameter types (for all messages treated by *class*), of current attribute values (of object *obj*) and of parameter values (for message *msg*), respectively (see Appendix B for more details).

Definition 12 (*Transformation of the Initial Graph*).

```
1   Init =      "init{
2                 atomic{"
3                   $\forall obj \in objV_{IG}$
                      "chan" $obj$ "= [BSIZE] of {mtype" $\underline{\text{ParType}_{type_V^{IG}(obj)}}$ "};"
                      "run" $type_V^{IG}(obj)$ "(" $obj$ $\underline{\text{Atrib}_{obj}}$ ");"
4                   $\forall msg \in msgE_{IG}$ with
                    $obj = t^{IG}(msg),\ tmsg =$ "msg"$\_type_V^{IG}(obj)\_type_E^{IG}(msg),\ lpar = \underline{\text{MsgParVal}_{msg}}$
                      $obj$ "!" $tmsg$ "("$lpar$");"
5                 "}"
6               "}"
```

7.2.2. Non-deterministic selection of messages

Definition 13 specifies how messages to objects are handled in PROMELA. This definition is needed to mimic the non-deterministic behavior of message selection of OBGG. This behavior is encoded in a read and a write procedures, described in Definitions 14 and 15, respectively.

Definition 13 (*Non-Deterministic Handling of Messages*).

```
1   $\text{Msgs}_{class}$ =      $\text{Read}_{class}$ $\text{Write}_{class}$
```

Each *object process* has an *object process buffer* "*opb_*" *class* with *bsize* positions (declared in line 6 of Definition 17). Read is an inline macro (called by "*ndr_*" *class*) that chooses non-deterministically (due to a conditional structure) a position of the *object process buffer* containing a message to be read (block 4). This message is stored in msg_Received and the corresponding buffer position is released. If no message is available in the buffer, no message is taken from the *object process buffer* (line 6). The function $\underline{\text{ParObj}_{class}}$ returns the list of all possible parameter names used in messages that this *class* can receive.

Definition 14 (*Non-Deterministic Read a Message From the Buffer of the Class Process*).

```
1   $\text{Read}_{class}$ =      "inline ndr_" $class$ "(){"
2                   "have_msg = true;"
3                   "if"
4                   $\forall i \in \{0..bsize\}$
                      ":: (busy[" $i$ "] == true) &&"
                        "(inspected[" $i$ "] == false);"
                        "opb_" $class$ "[" $i$ "]" '?msg_Received" $\underline{\text{ParObj}_{class}}$ ";"
                        "busy[" $i$ "] = false;"
5                  ":: else;"
6                    "have_msg = false"
7                  "fi;" "}"
```

Write is also an inline macro (with syntax "*mbw_*" *class*), which writes a message to the first available slot in the *object process buffer* (block 3). If the *object process buffer* reaches its maximum size during the execution of a transformed PROMELA model, an error is generated (line 5), aborting the analysis of the model in SPIN and requiring the buffer size to be increased.

Definition 15 (*Write a Message to the First Available Slot in the Buffer of the Class Process*).

```
1   $\textsc{Write}_{class}$ =   "inline mbw_" $class$ "(){"
2                     "if"
3                     $\forall i \in \{0..bsize\}$
                        | ":: (busy[" $i$ "] == false);"
                        |   "opb_" $class$ "[" $i$ "]" "!msg_Received" $\underline{\textsc{ParObj}}_{class}$ ";"
                        |   "busy[" $i$ "] = true;"
                        |   "inspected[" $i$ "] = true;"
4                     ":: else;"
5                       "false"
```

7.2.3. Rules

The formal definition of transformation of rules of a given OBGG class is shown in Definition 16. A transformed rule is defined as the inline macro "*rules_*" *class*. A rule process non-deterministically chooses the rule to be applied to treat the message stored in msg_Received. The macro contains first the declaration of variables appearing in the definition of the rules of the corresponding *class* of *GG* (generated by $\underline{\textsc{Decl}}_{class}$). Each OBGG rule becomes an entry (block 5) of a conditional PROMELA structure. In order to apply a rule, there must be a match for the rule (represented by the guard in the conditional structure): the name of the message in msg_Received must be the name of the message treated by this rule, and the other conditions concerning the parameter of the message and rule equations must be true (tested by the auxiliary function $\textsc{TestAtr}_{nr,class}$). When a rule is applied, the variables of the *object process* are updated ($\underline{\textsc{UpdateAtr}}_{nr,class}$), new *object processes* may be created ($\underline{\textsc{CreateObj}}_{nr,class}$), messages may be sent ($\underline{\textsc{SendMsg}}_{nr.class}$), and the global variable *event_RuleName* is set to the name of the applied rule. If the message taken from the *object process buffer* cannot be applied (line 6), the message is sent back to the *object process buffer* (line 7), and the local variable *rule_applied* is set to *false* (line 8).

Definition 16 (*Transformation of Rules Particular to an OBGG Class*).

```
1   $\textsc{Rules}_{class}$ =  "inline rules_" $class$ "(){"
2                  $\underline{\textsc{Decl}}_{class}$
3                  "rule_applied = true"
4                  "if"
5                  $\forall\, nr \in N$ with $n(nr) = (r : L \to R, Eq)$ and $msg \in E_L$
                     | ":: (msg_Received == msg_" $class$ "_" $type_E^L(msg)$ ")and" $\underline{\textsc{TestAtr}}_{nr,class}$
                     | $\underline{\textsc{UpdateAtr}}_{nr,class}$
                     | $\underline{\textsc{CreateObj}}_{nr,class}$
                     | $\underline{\textsc{SendMsg}}_{nr,class}$
                     | "event_RuleName = rule_" $class$ "_" $nr$
6                  ":: else;"
7                    "mbw_" $class$ "();"
8                    "rule_applied = false;"
9                  "fi;"
10               "}"
```

7.2.4. Class process behavior

Definition 17 presents the body of a transformed OBGG *class* (lines 1 to 31). The *class process* has the name of the OBGG class. The *object process channel* and the list of pairs type/name of attributes of objects of this *class* (given by function $\textsc{AtrObj}_{class}$) are passed as parameters in the definition of the *class process* (line 1). The list of parameters of messages that an object of this class can receive (given by auxiliary functions $\underline{\textsc{ParObj}}_{class}$ and $\underline{\textsc{TypeParObj}}_{class}$) are defined in line 2, and are used in conjunction with the variable msg_Received (line 3) to map OBGG messages to PROMELA. Lines 4 to 6 define arrays used to represent the *object process buffer*. Line 7 declares two auxiliary boolean variables. The generic behavior of a *class process* is defined in the loop of lines 8 to 30. Initially, the object waits for new messages to appear in the *object process channel* (line 10). Upon reception of messages, it stores the messages in the *object process buffer*, by calling the Write procedure (line 11). This process of moving messages from the *object process channel* to the corresponding *object process buffer* occurs atomically (lines 9 to 13). When there is at least one message to be treated (have_msg is true) and the *object process channel* is empty (i.e., all the messages to this object are in the *class process buffer*), the object tries to handle one message (lines 14 to 29). The sequence of steps to handle a message is also performed atomically. The object reads a message from the *object process buffer* (line 17) and enters in another loop (lines 19 to 28), searching for a rule that is triggered by this message. If no rule is found, another message is read from the *object process buffer* (line 22) and the behavior is repeated. This loop ends when: (a) one message of the *object process buffer* is consumed and a rule is applied (line 25); (b) none of the messages in the *object process buffer* can be consumed (line 27). After leaving the loop, the process goes back to its external loop (checks whether new messages have arrived).

Definition 17 (*Transformation of OBGG Class Behavior*).

```
1   $\textsc{Proc}_{class}$ =   "proctype" $class$ "(chan opc_" $class$ ";" $\underline{\textsc{AtrObj}_{class}}$ "){"
2         $\forall i \in \{0..|\underline{\textsc{ParObj}_{class}}|\}$
              | $\underline{\textsc{TypeParObj}_{class}}[i]$ " " $\underline{\textsc{ParObj}_{class}}[i]$ ";"
3         "mtype mgs_Received;"
4         "bool busy[BSIZE] = false;"
5         "bool inspected[BSIZE] = false;"
6         "chan opb_" $class$ "[BSIZE] = [1]of{mtype" $\underline{\textsc{ParType}_{class}}$ "};"
7         "bool rule_applied = false, have_msg;"
8         "do"
9         ":: atomic{"
10            "opc_" $class$ "?msg_Received" $\underline{\textsc{ParObj}_{class}}$ ";"
11            "mbw_" $class$ "();"
12            "have_msg = true;"
13          "}"
14        ":: atomic{"
15            "(len(opc_" $class$ ") == 0)&&((have_msg)||(rule_applied));"
16        $\forall i \in \{0..bsize\}$
              | "inspected[" $i$ "] = false;"
17            "ndr_" $class$ "();"
18            "rule_applied = false;"
19            "do
20            ":: (have_msg)&&(!rule_applied);"
21              "rules_" $class$ "();"
22              "ndr_" $class$ "();"
23            ":: (have_msg)&&(rule_applied);"
24              "mbw_" $class$ "();"
25              "break;"
26            ":: (!have_msg);"
27              "break;"
28            "od;"
29          "}"
30        "od;"
31      "}"
```

8. Correctness of transformation

As discussed in Sections 4 and 6, we use SPIN to prove properties over sequences of events of a graph grammar — sequences of rule applications. In the proposed transformation to PROMELA, these events are described by changes in the global variable `event_RuleName`. Therefore, in order to assure that properties of a transformed PROMELA model Π are valid for the original OBGG GG, we must guarantee that the sequences of events generated by both models (GG and Π) are the same, i.e., sequences of rule applications for GG derivations correspond to sequences of changes in the variable `event_RuleName` for Π transition sequences. This is the semantic compatibility that is proved in this section.

8.1. Relating OBGG and PROMELA States (Step 2 described in Section 6)

The comparison of derivations of an OBGG and PROMELA paths is based on relating states and transition/derivation labels. We need to define how OBGG states are transformed to PROMELA states and vice versa, and the same for labels of transitions/derivations. This means that we must find a correspondence between PROMELA states and graphs. We can always transform an OBGG state into a PROMELA state, but the opposite is not true. In the PROMELA-LTS of a model, which is the transformation of an OBGG model, there are several states that do not correspond to any states of the original OBGG-LTS. This is due to the fact that the treatment of messages in OBGG occurs atomically (in only one step), whilst in PROMELA this takes several steps. Thus, in the PROMELA-LTS, there are states that represent the partial treatment of messages. A PROMELA state that corresponds to an OBGG state is called a *well-formed state*.

Definition 18 (*Well-Formed State*)**.** Given an OBGG *GG* and its transformation to PROMELA $Transf_{OBGG}(bsize, GG) = \Pi$, a state $(\pi, \mathcal{G}, \mathcal{C}, pdef, act, at) \in SemProm(\Pi)$ (see Definition 9) is well-formed if

1. $\pi = \varepsilon$;
2. $\forall i \in dom(act).act(i) = (\pi_1, \mathcal{L}, \varepsilon) \wedge (\pi_1 = \varepsilon \vee \pi_1 = \texttt{do ::atomic } \{\pi_{msg}\} \texttt{ ::atomic } \{\pi_{app}\} \texttt{ od})$.

The set of all well-formed states of a transformation Π is called $\mathbf{WFState}_{\Pi}$.

For a PROMELA-LTS state to correspond to a graph (OBGG-LTS state), all global (process, channel and variable) declarations must have been processed (1); there must be one active process (waiting to run) for each object in the graph (2). The first process that runs in a PROMELA model is the `init` process. The `init` process in our transformation creates all processes corresponding to objects of the initial graph of *GG*, creates all messages in the respective buffers (associated to each object by the function $\mathcal{C}$) and terminates. Only when this initial process terminates it is possible to have a well-formed state, because the main model has finished ($\pi = \varepsilon$). Moreover, we require that each object process is ready to execute the main do statement in its code, meaning that all local variables have been created, and no rule is currently being executed.

$$\begin{array}{ll}
\mathcal{G}(\texttt{event\_RuleName}) = (\texttt{mtype}, 0) & \\
\mathcal{C}(0) = \mathcal{C}(1) = \mathcal{C}(2) = ((\texttt{mtype}), \varepsilon, 3) & \text{external channels for } \texttt{Phil} \text{ objects} \\
\mathcal{C}(3) = \mathcal{C}(4) = \mathcal{C}(5) = ((\texttt{mtype}, \texttt{chan}), \varepsilon, 3) & \text{external channels for } \texttt{Fork} \text{ object} \\
\mathcal{C}(6) = ((\texttt{mtype}), (\texttt{GetFork}, \perp, \perp), 1) & \text{internal channel for } \texttt{Phil1} \text{ object containing one message} \\
\mathcal{C}(7) = \mathcal{C}(8) = ((\texttt{mtype}), \varepsilon, 1) & \text{empty internal channels for } \texttt{Phil1} \text{ object} \\
\ldots & \\
\mathcal{C}(21) = \mathcal{C}(22) = \mathcal{C}(23) = ((\texttt{mtype}, \texttt{chan}), \varepsilon, 1) & \text{internal channels for } \texttt{Fork3} \text{ object}
\end{array}$$

$$\begin{array}{ll}
pdef(\texttt{Phil}) & = (\pi_{Phil}, par_{Phil}) \\
& par_{Phil}(1) = (\texttt{opc\_Phil}, \texttt{chan}) \\
& par_{Phil}(2) = (\texttt{atr\_id}, \texttt{byte}) \\
& par_{Phil}(3) = (\texttt{atr\_phase}, \texttt{byte}) \\
& par_{Phil}(4) = (\texttt{atr\_leftfirst}, \texttt{bool}) \\
& par_{Phil}(5) = (\texttt{atr\_leftfork}, \texttt{chan}) \\
& par_{Phil}(5) = (\texttt{atr\_rightfork}, \texttt{chan})
\end{array}
\qquad
\begin{array}{ll}
pdef(\texttt{Fork}) & = (\pi_{Fork}, par_{Fork}) \\
& par_{Fork}(1) = (\texttt{opc\_Fork}, \texttt{chan}) \\
& par_{Fork}(2) = (\texttt{atr\_id}, \texttt{byte}) \\
& par_{Fork}(3) = (\texttt{atr\_use}, \texttt{bool})
\end{array}$$

$$\texttt{init} \left\{ \begin{array}{l}
act(0) = (\varepsilon, \pi_0, \mathcal{L}_0, \varepsilon) \\
\;\mathcal{L}_0(\texttt{Phil1}) = (\texttt{chan}, 0) \\
\;\mathcal{L}_0(\texttt{Phil2}) = (\texttt{chan}, 1) \\
\;\mathcal{L}_0(\texttt{Phil3}) = (\texttt{chan}, 2) \\
\;\mathcal{L}_0(\texttt{Fork1}) = (\texttt{chan}, 3) \\
\;\mathcal{L}_0(\texttt{Fork2}) = (\texttt{chan}, 4) \\
\;\mathcal{L}_0(\texttt{Fork3}) = (\texttt{chan}, 5)
\end{array} \right.$$

$$\begin{array}{l}\texttt{Phil1} \\ \text{object}\end{array} \left\{ \begin{array}{l}
act(1) = (\pi_1', \pi_1, \mathcal{L}_1, \varepsilon) \\
\;\mathcal{L}_1(\texttt{opc\_Phil}) = (\texttt{chan}, 0) \\
\;\mathcal{L}_1(\texttt{atr\_id}) = (\texttt{byte}, 1) \\
\;\mathcal{L}_1(\texttt{atr\_phase}) = (\texttt{byte}, 1) \\
\;\mathcal{L}_1(\texttt{atr\_leftfirst}) = (\texttt{bool}, 1) \\
\;\mathcal{L}_1(\texttt{atr\_leftfork}) = (\texttt{chan}, 1) \\
\;\mathcal{L}_1(\texttt{atr\_rightfork}) = (\texttt{chan}, 1) \\
\;\mathcal{L}_1(\texttt{opb\_Phil}) = (\texttt{arr(3,chan)}, [6, 7, 8])
\end{array} \right.
\quad \ldots \quad
\begin{array}{l}\texttt{Fork3} \\ \text{object}\end{array} \left\{ \begin{array}{l}
act(6) = (\pi_6', \pi_6, \mathcal{L}_6, \varepsilon) \\
\;\mathcal{L}_6(\texttt{opc\_Fork}) = (\texttt{chan}, 5) \\
\;\mathcal{L}_6(\texttt{atr\_id}) = (\texttt{byte}, 3) \\
\;\mathcal{L}_6(\texttt{atr\_use}) = (\texttt{bool}, 0) \\
\;\mathcal{L}_6(\texttt{opb\_Fork}) = (\texttt{arr(3,chan)}, [21, 22, 23])
\end{array} \right.$$

Fig. 8. PROMELA State corresponding to the Initial OBGG Graph.

For example, a PROMELA state corresponding to the initial graph of Fig. 4 (b) is the state $ST = (\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, \perp)$, with definitions shown in Fig. 8 (definitions of model fragments π are omitted and only part of channels and processes corresponding to objects are shown). The function *act* defines all instantiated processes (active or terminated). In Fig. 8, we can see that there is one active process for each object in the initial graph, since the initial process has terminated (the first argument of $act(0)$ is ε). The functions $\mathcal{L}_n$ define the values of object attributes. For instance, in the initial graph of Fig. 4 (b) the `leftfirst` attribute of `Phil1` is `true` and in the PROMELA state this same value can be seen in $\mathcal{L}_1(\texttt{atr\_leftfirst}) = (\texttt{bool}, 1)$. The messages and their parameters, present in the initial graph, appear in the range of function $\mathcal{C}$. All messages (and their parameters) sent to each object appear in one of the internal channels of the corresponding object (there are as many internal channels as the maximum number of messages sent to each object at each time). The information described in the type graph of the OBGG is stored via the *pdef* function, which associates to each type of object (process name) a process body modeling its behavior. To make the example more readable, in the description of the global variable and the channels, we have used the type `mtype` and the names of rules/messages — although it should be `byte` and rules/messages numbers (since enumeration types are actually treated using numbers in PROMELA).

As discussed in Section 7.2, for each transformed object, there is a corresponding PROMELA process (the *object process*). Before choosing a message to be treated, an object process has to read all messages in its *object process channel* (channel used to receive messages externally) and store them in the *object process buffer* (array of channels used to store the messages internally), where the messages are non-deterministically chosen. Thus, many PROMELA states may correspond to the same graph, since in the graph there is no distinction of message channels — all messages are always connected to the corresponding target objects. Now we make this correspondence precise, to be able to prove that the behavior is preserved by our transformation. To ease understanding, the following definition is stated in an intuitive way. The corresponding formal definition can be found in Appendix C (Definition 25).

Definition 19 (*Transformation from PROMELA Well-Formed States to OB-Graphs*). Given an OBGG *OG* over the class graph *C* and its transformation into a PROMELA model Π, the transformation of a well-formed state $S = (\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, \bot) \in \mathbf{WFState}_\Pi$ to an OB^C-graph H^C is given by the function $\mathcal{T}_{P\to G} : \mathbf{WFState}_\Pi \to \mathbf{OBGraph(C)}$ defined by $\mathcal{T}_{P\to G}(S) = (H, type^H)$, where

- $H = (V_H, E_H, s^H, t^H, Lab_H, lab^H, A_H, a^H)$
  - V_H is the set containing (i) all identifiers of processes representing objects (described as *class.c*, where *class* is the name of the process type and *c* is the number of the external channel associated to each object) and (ii) all possible PROMELA values;
  - E_H is the set containing (i) all messages that are in internal or external channels associated to object processes and (ii) one attribute edge for each object process of the PROMELA state;
  - Lab_H is the set of attribute and parameter names used in objects/messages;
  - A_H is the set of all PROMELA values;
  - functions s^H, t^H, lab^H and a^H are defined in a straightforward way, taking into account the targets of messages in channels and the local states of each object process;
- $type^H : H \to C$ maps all objects to corresponding classes, values to corresponding names of carrier sets, messages to corresponding message types.

Besides the fact that messages may be in different channels in a PROMELA state, there are other situations in which different PROMELA states may correspond to the same graph. Messages may have been created in different orders (leading to different orders in corresponding channels and buffers), although, viewed as a set of messages, they are the same. These states, when transformed into the OB^C-graphs, give rise to isomorphic graphs and are considered equivalent.

Definition 20 (*Equivalence of PROMELA States*). Given an OBGG *OG* over the class graph *C* and its transformation into a PROMELA model Π, two well-formed states $S1$ and $S2$ are **equivalent**, denoted by $S1 \equiv_P S2$, iff $\mathcal{T}_{P\to G}(S1) \equiv_G \mathcal{T}_{P\to G}(S2)$.

We can also transform an OB^C-graph in a PROMELA (WF-)state. Since the PROMELA state carries information about the last applied rule (an OB^C-graph does not contain it), we use a rule name as a parameter for this transformation (or zero, if no rule was applied). Moreover, we need the information about the maximum buffer size (*bsize*) to create the suitable PROMELA state. Again, the following definition is stated informally, the corresponding formal definition can be found in Appendix C (Definition 26).

Definition 21 (*Transformation from OB^C-Graphs to PROMELA Well-Formed States*). Given an OBGG *OG* over the class graph *C*, a rule name *nr* (a rule name or zero), a maximum buffer size *bsize* and the transformation of *OG* into a PROMELA model Π, the transformation of an OB^C-graph $(H, type^H)$ into a PROMELA WF-state $S = (\pi, \mathcal{G}, \mathcal{C}, pdef, act, at) \in \mathbf{WFState}_\Pi$ is given by the function $\mathcal{T}_{G\to P} : \mathbf{OBGraph(C)} \to \mathbf{WFState}_\Pi$ defined by $\mathcal{T}_{G\to P}(H, type^H) = S$, where

- $\pi = \varepsilon$;
- $\mathcal{G}(\texttt{event\_RuleName}) = (\texttt{mtype}, nr)$;
- $\mathcal{C}$ maps each channel identifier either to an external or to an internal channel associated to an object (each object has one external and *bsize* internal channels — the external channel has a buffer size of *bsize*, the internal ones have a buffer of one position). External channels are empty, and each message appearing in graph *OG* is put in one of the internal channels of the corresponding object;
- *pdef* associates a code (according to Definition 17) and a list of attributes to each class defined in *C*;
- *act* maps each object process identifier to its corresponding local state, where the code to be executed is `do ::atomic {`π_{msg}`} :: atomic{`π_{app}`} od` from Definition 17 (instantiated for the corresponding object) and the values of local variables are obtained from the corresponding attributes of objects in graph *OG*;
- $at = \bot$

The relation between the two transformations is given by the following proposition. Given an OB^C-graph G, transforming it into a PROMELA model and back we arrive in the same graph G (up to isomorphism).

Proposition 1. *Given an OBGG OG over the class graph C, a rule name nr (a rule name or zero), a maximum buffer size bsize and the transformation of OG into a PROMELA model Π, for any graph OB^C-graph G*

$$\mathcal{T}_{P\to G}(\mathcal{T}_{G\to P}(G)) \equiv_G G$$

Proof. The transformation $\mathcal{T}_{G\to P}(G)$ will produce a PROMELA well-formed state S (according to Definition 21), having external and internal channels (of size *bsize*) for each object of G and corresponding messages of G in these channels. When this state S is transformed using $\mathcal{T}_{P\to G}$ (according to Definition 19), for each PROMELA process corresponding to an object, a vertex with corresponding attribute edge is created, and for each message that is in an internal or external channels of S, a corresponding message edge is generated. The global variable storing the last applied rule and the buffer size are not mapped by $\mathcal{T}_{P\to G}$, since this information is not present in OB^C graphs. Therefore, we obtain an OB^C-graph that is isomorphic to G. $\square$

8.2. Relating derivation steps to paths (Step 3 described in Section 6)

Besides transforming the states we must transform derivations to paths. We start by defining a path (in fact, a class of paths) that generates a state corresponding to the initial graph of the grammar.

Proposition 2 (*Initial Path*)**.** *Given an OBGG OG with initial graph IG and its transformation into a PROMELA model Π. Let $\to^{init}$ be the path of SemPROM(Π) such that the last state of $\to^{init}$ is the only well-formed state in $\to^{init}$ and there is no transition labeled with τ in $\to^{init}$. Then $\mathcal{T}_{P\to G}(OUT^{\to^{init}}) \equiv_G IG$.*

Proof. When a PROMELA model starts, it must run its `init` process. Only when this process finishes the first component of the global state (the yet unexecuted code) can be ε (satisfying the first condition of Definition 18). As it can be seen in Definition 12, this process creates one process for each object of the initial graph *IG* (line 4) and generates elements in the buffer channels corresponding to all messages that are in *IG* (line 5). When this process finishes, each process corresponding to an object may run. These processes first execute a series of declarations and assignments, and then proceed to the statement `do ::atomic {`π_{msg}`} ::atomic {`π_{app}`} od`. If some process starts execution the do-loop, the first transition that is generated is labeled with τ (see DO-rule in the semantics of PROMELA). Since there are no such transitions in $\to^{init}$ and the final state is well-formed, all processes must have finished the initialization state and be ready to execute the corresponding do-loops. This state corresponds (up to isomorphism) to the initial graph of *OG*.

Our proof of semantical compatibility considers PROMELA paths starting with $\to^{init}$ sequences. However, there may be a sequence σ that has no well-formed state at all, but represents many rule applications. This is due to the fact that one (or more) object process may not finish the initialization phase (never reach the do-statement) — a PROMELA model may not be fair, and thus there may be processes that are postponed forever. This situation of not reaching a well-formed state may only happen in the initialization phase (when not every process has reached its do-loop), after this, it is always possible to reach again a well-formed state by executing a sequence of transitions that corresponds to rule applications or moving messages from external to internal channels. However, if some object did not reach its do-loop, it means that it is not ready to execute, and thus no rule can be applied to such objects. Therefore, all sequences of rule applications that can be observed in paths that do not start with $\to^{init}$ sequences can also be observed in a corresponding path starting with a $\to^{init}$ sequence (just some objects may be "inactive" in the former case). $\square$

Consider a derivation of an *OBGG*. Looking at the corresponding PROMELA code, it becomes obvious that this single step is implemented by a list of transitions. Due to the way the transformation model and well-formed states were defined, once we are in a well-formed state, the code to be executed is `do ::atomic {`π_{msg}`} ::atomic {`π_{app}`} od`. Thus, according to PROMELA's semantic rules, this model will execute atomically one of the blocks π_{msg} or π_{app} and start again. However, only in π_{app} it is possible to have a transition labeled by `event_RuleName := rule.obj.`r, and this transition occurs exactly after rule with name r has been applied. This means that a rule implementation sequence $\to^r$ will be a sequence of transitions having the following form (where only $\circ_1$ and $\circ_n$ are well-formed states)

$$\to^r = \circ_1 \xrightarrow{\tau} \bullet_2 \xrightarrow{\texttt{atomic}} \bullet_3 \cdots \xrightarrow{\texttt{event\_RuleName := rule.obj.r}} \cdots \xrightarrow{\texttt{end\_atomic}} \circ_n$$

This idea is formalized in the following definition.

Definition 22 (*Rule Implementation Sequence*)**.** Given an OBGG $GG = (Spec, X, C, IG, N, n)$ and its transformation Π into PROMELA. A finite subsequence $\sigma 1$ of a sequence $\sigma \in SemPROM(\Pi)$ is called a **rule implementation sequence**, denoted by $\to^r$, if

(i) $\sigma 1$ starts and finishes in well-formed PROMELA states;
(ii) there is exactly one transition labeled as `event_RuleName := rule.obj.`r.

If a sequence of PROMELA transitions σ that contains a well-formed state does not contain a rule implementation sequence, it means that either the executed statements belong to π_{msg} atomic sequences or π_{app} sequences that just check whether a rule is applicable, but without applying this rule (or parts of these sequences). In any case, all well-formed states of σ will give rise to isomorphic OB^C-graphs, since neither attributes of objects have changed, nor new messages have been created (because these effects can only be produced by the PROMELA code Rules (Definition 16) in the situation that a rule is being applied). Therefore, we say that these are non-observable sequences (because from the point of view of the underlying graph grammar, this sequence has no effect).

Definition 23 (*Non-Observable Sequence*)**.** Given an OBGG $GG = (Spec, X, C, IG, N, n)$ and its transformation Π into PROMELA. Then a sequence $\sigma \in SemPROM(\Pi)$ is called a **non-observable sequence**, denoted by $\rightarrow^{\tau}$, if

(i) σ contains at least one well-formed state;
(ii) no rule implementation sequence is a subsequence of σ.

Now we can relate a derivation step of an OBGG to a transition sequence of the corresponding PROMELA model.

Proposition 3. *Given an OBGG $GG = (Spec, X, C, IG, N, n)$, its transformation Π into PROMELA and an OB^C-graph OG. There is a derivation $OG \overset{r}{\Longrightarrow} OH$ if and only if there is a rule implementation sequence $PG \rightarrow^{r} PH$, with $\mathcal{T}_{G\rightarrow P}(OG) \equiv_P PG$ and $\mathcal{T}_{G\rightarrow P}(OH) \equiv_P PH$.*

Proof. 1. Assume there is a derivation $OG \overset{r}{\Longrightarrow} OH$. Then there is a match m and a pushout square

$$\begin{tikzcd}
\overline{L} \arrow[r, "\overline{r}"] \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & \overline{R} \arrow[d, "m'"] \\
OG \arrow[r, "r'"'] & OH
\end{tikzcd}$$

Let $PG = \mathcal{T}_{G\rightarrow P}(OG)$. The fact that m exists means that all objects and messages of the left-hand side of rule L are present in OG (because m is a total graph morphism). This is the condition that enables rule application. There must be at least one message in OG, since a rule application must always consume a message. By the transformation in Definition 21 of graphs into PROMELA states, OG is transformed to a state in which all messages are in the external channels of the object processes. Thus, there must be a non-observable sequence of transitions $PG \overset{\tau}{\rightarrow} \bullet_1$ that leads to a state $\bullet_1$ in which all messages are in the corresponding internal buffers (π_{msg} can always be performed if there are messages in the external channels, and, since in PG the internal buffers were empty, there will be enough space in the internal buffers for all messages in external channels). As $PG \equiv_P \bullet_1$, $OG \equiv_G \mathcal{T}_{P\rightarrow G}(\bullet_1)$. The same derivation step can be performed over $\mathcal{T}_{P\rightarrow G}(\bullet_1)$. Now, since there is a rule $r : L \rightarrow R$ that has a match in OG, there must be a message in the internal buffer of an object process of $\bullet_1$ that corresponds to the trigger of r. This rule is enabled in graph OG and, therefore, the corresponding conditions given by TestAttr$_r$ are satisfied (line 4 in Definition 16). In this case, the rules will be applied, generating a rule implementation sequence $\bullet_1 \overset{r}{\rightarrow} \bullet_2$. In Definition 16, attributes are changed, messages and objects are created according to the definition of rule r, and the effect of the pushout (1) is to change attributes, create new messages and objects, and delete the message that triggered the rule application. Thus, the effect of this rule application is completely captured by the PROMELA code in Definition 16. This means that $\mathcal{T}_{P\rightarrow G}(\bullet_2) \equiv_G OH$ and consequently $\mathcal{T}_{G\rightarrow P}(OH) \equiv_P \bullet_2$.

2. Assume there is a rule implementation sequence $PG \rightarrow^{r} PH$ and OBgraph OB, such that $\mathcal{T}_{G\rightarrow P}(OG) = PG$. By an argumentation analogous to the previous case, we can conclude that there must be a derivation step $\mathcal{T}_{P\rightarrow G}(PG) \overset{r}{\Longrightarrow} \mathcal{T}_{P\rightarrow G}(PH)$ (in this case we do not need to consider non-observable subsequences, since a rule implementation sequence does not have such subsequences). Moreover, since a rule implements exactly the pushout construction, we must have $\mathcal{T}_{G\rightarrow P}(OH) \equiv_P PH$. $\square$

8.3. Proving correctness of transformation (Step 4 described in Section 6)

We can extend this result to derivations and sequences of transitions, that is, paths. First, we define a way to relate derivations and paths.

Definition 24 (*Rule Name Sequence*)**.** Given an OBGG $GG = (Spec, X, C, IG, N, n)$, its transformation Π into PROMELA, and a path $\sigma \in SemPROM(\Pi)$. The **rule name sequence** of σ, denoted by $rname(\sigma)$, is defined as the list of rule names corresponding to rule implementation sequences of σ in order of appearance in σ.

Theorem 1. *Given an OBGG $GG = (Spec, X, C, IG, N, n)$ and its transformation Π into PROMELA. For any finite derivation δ of SemOBGG(GG) there is a corresponding PROMELA path $\sigma \in SemPROM(\Pi)$ such that the rule-name sequence of σ is the rule-application sequence of δ and $\mathcal{T}_{G\rightarrow P}(OUT^{\delta}) \equiv_P OUT^{\sigma}$.*

Proof. The proof is by induction on the size of the path δ. As buffer size for the PROMELA model, we consider the maximum number of messages that appeared in a graph derivation δ.

Basis: $|\delta| = 0$.
The PROMELA path $\rightarrow^{init}$ corresponding to the generation of the initial state can always occur. For this path, we have $rname(\rightarrow^{init}) = rapplic(\delta)$ and $\mathcal{T}_{G\rightarrow P}(OUT^{\delta}) \equiv_P OUT^{\rightarrow^{init}}$ (remember that the output state of an empty derivation is, by definition, the initial graph, and the sequence $\rightarrow^{init}$ generates a graph isomorphic to *IG* with zero as value of `event_RuleName`).

Induction Hyp. For derivation δ_n of length n, there is a PROMELA path σ_n such that $rname(\sigma_n) = rapplic(\delta_n)$ and $\mathcal{T}_{G\rightarrow P}(OUT^{\delta}_n) \equiv_P OUT^{\sigma}_n$.

Induction Step: Assume that for the n first steps of δ it was possible to generate the corresponding rule implementation sequence (induction hypothesis). Since δ_{n+1} is a derivation and δ_n is a prefix of it, OUT^n must be the graph from which the last derivation step of δ_{n+1} starts. Let r be the rule applied at the $i+1$th step of δ. If it is possible to apply rule r at graph OUT_n (generating the last step of δ_{n+1}), it must also be possible to generate the corresponding rule implementation sequence (according to Proposition 3). By concatenating this sequence with the sequence σ_n, we can generate a transition sequence σ_{n+1} such that $rname(\sigma_{n+1}) = rapplic(\delta_{n+1})$ and $\mathcal{T}_{G\rightarrow P}(OUT^{\delta}_{n+1}) \equiv_P OUT^{\sigma}_{n+1}$ (considering `event_RuleName`$= r$). □

Since any infinite derivation can be seen as a limit of a chain of finite derivations [51] ordered by inclusion, we can use the construction defined in the last theorem to construct corresponding infinite PROMELA paths, provided there is a bound in the number of messages that can be at the same time present in a (graph) state.

Rule implementation sequences must be executed atomically and, therefore, their execution cannot be interleaved with other sequences of transitions. Moreover, all transition sequences that are not rule implementation sequences must be non-observable sequences (because only the rule application may change the (graph) state of a system). Thus, its seems natural to expect that exactly the same sequences of rule applications and rule implementation sequences can be observed

Theorem 2. *Given an OBGG $GG = (Spec, X, C, IG, N, n)$ and its transformation Π into PROMELA. Let $\sigma \in SemPROM(\Pi)$ such that σ is finite, $\rightarrow^{init}$ is the prefix of σ and OUT^{σ} is a well-formed state. There is a derivation δ of GG such that the rule-name sequence of σ is the rule-application sequence of δ and $\mathcal{T}_{P\rightarrow G}(OUT^{\sigma}) \equiv_G OUT^{\delta}$.*

Proof. We can construct a derivation δ such that $rapplic(\delta) = rnames(\sigma)$ and $\mathcal{T}_{P\rightarrow G}(OUT^{\sigma}) \equiv_G OUT^{\delta}$ by induction on the length of $rname(\sigma)$:

Basis Length of $rname(\sigma) = 0$.
In this case, $rname(\sigma) = \varepsilon$, and thus it must be a non-observable transition sequence. In this case, the corresponding derivation δ of *GG* would be the empty derivation, since no rule has been applied. Obviously, in such a situation we have $rapplic(\delta) = \varepsilon = rname(\sigma)$ and $\mathcal{T}_{P\rightarrow G}(OUT^{\sigma}) \equiv_G OUT^{\delta}$, since OUT^{σ} must be the PROMELA state corresponding to the initial graph (because $\rightarrow^{init}$ is a prefix of σ, no other rule has been applied, and OUT^{σ} is well-formed) and OUT^{δ} is the initial graph (by definition, the output graph of an empty derivation is the initial graph).

Induction Hyp.: There is a derivation δ_n with length n such that $rapplic(\delta_n) = rname(\sigma_n)$ and $\mathcal{T}_{P\rightarrow G}(OUT^{\sigma}_n) \equiv_G OUT^{\delta}_n$

Induction Step: Now we have to construct the derivation δ_{n+1} such that $rapplic(\delta_{n+1}) = rname(\sigma_{n+1})$ and $\mathcal{T}_{P\rightarrow G}(OUT^{\sigma}_{n+1}) \equiv_G OUT^{\delta}_{n+1}$. Let r be the name of the rule corresponding to the $i+1$th element of $rname(\sigma_{n+1})$. The transition sequence σ_{n+1} must have the form

$$\cdots OUT^{\sigma}_n \rightarrow^{\tau} \bullet_1 \rightarrow^{r} \bullet_2 \rightarrow^{\tau} OUT^{\sigma}_{n+1}$$

where $\rightarrow^{\tau}$ are non-observable transition sequences. Since these sequences are non-observable, we have (a) $OUT^{\sigma}_n \equiv_P \bullet_1$ and (b) $OUT^{\sigma}_{n+1} \equiv_P \bullet_2$. Using the induction hypothesis and (a) we conclude that $\mathcal{T}_{P\rightarrow G}(\bullet_1) \equiv_G OUT^{\delta}_n$. Then, using Proposition 3, the fact that there is a rule implementation sequence using rule r and (b) we can conclude that there is a derivation step $OUT^{\delta}_n \overset{r}{\Longrightarrow} OUT^{\delta}_{n+1}$ such that $\mathcal{T}_{P\rightarrow G}(\bullet_2) \equiv_G OUT^{\delta}_{n+1}$. Concatenating this step with sequence δ_n would lead to a sequence δ_{n+1} such that $rapplic(\delta_{n+1}) = rname(\sigma_{n+1})$ (because $rapplic(\delta_n) = rname(\sigma_n)$ and the $n+1$th element of both sequences is r) and $\mathcal{T}_{P\rightarrow G}(OUT^{\sigma}_{n+1}) \equiv_G OUT^{\delta}_{n+1}$. □

Infinite sequences of PROMELA transitions are obtained as limits of chains of finite sequences ordered by the prefix relation. If a sequence σ is infinite, $\rightarrow^{init}$ must be a prefix of it, since the initialization process always terminates. Thus σ has at least one well-formed state (corresponding to the initial graph). We can use the construction presented in the proof of Theorem 2 to build the corresponding derivation δ (that will also be the limit of the chain of its prefix derivations).

We can use these results to consider properties involving state variables. Indeed, we would have to define corresponding global variables in PROMELA and include these variables in the labels of transitions (of the PROMELA model) and derivation steps (of the OBGG). In this setting, an event would not be given just by a rule name, but would include more information (for example, the object that executed the event).

9. Conclusions

The use of model transformation for model checking purposes is widely advocated in the literature since it enables the use of already established model checking tools. One fundamental task that serves as basis to such approaches is to show that the transformed model, on which model checking will actually be performed by the tool, preserves the semantics of the original model. In this paper we have tackled this issue. We have described in detail the proof of semantic compatibility between a model in the Object-Based Graph Grammars (OBGG) language and its corresponding PROMELA model (the input language of the SPIN model checker). Both languages are used to model highly concurrent systems, and therefore proving that a transformation from OBGG to PROMELA preserving semantics is not a trivial task. This proof was based on the transition system semantics of both languages.

The OBGG formal specification language was tailored to the specification of asynchronous distributed systems, where we have focused on adopting a transformation-based approach whenever possible to take advantage of several existing analysis and implementation techniques, languages, and tools. The proved correct transformation from OBGG to PROMELA presented in this paper plays an important role in our setting because several other transformations are based on this one.

Although the semantic compatibility proof was specific to our case, we explained the steps (Section 6) we have followed to accomplish this proof. These steps include relevant hints on which aspects shall be of special care on building translations to the PROMELA language. They may serve as guide to construct analogous proofs for different input languages in order to prove the correctness of their transformations to PROMELA, validating thus these approaches.

Acknowledgements

This work was partially supported by CNPq, FAPERGS and projects NESS (10/0043-0) and PqG (1014867).

Appendix A. PROMELA semantics

The semantics of PROMELA used here is a revised version from the semantics presented in [59]. Since we do not use `goto` statements and labels for commands, we simplified the basic structure defining local states. We also revised the definition of several rules.

The semantic model of PROMELA is described as a labeled transition system (LTS) defined by a set of structural operational semantics (SOS) rules. The types used to define the states of this LTS are defined in Appendix A.1. The definition of the states and transitions is split into two main parts: behavior of single processes (Appendix A.2) and behavior of the complete system (Appendix A.3).

A.1. Data types and values

Data is modeled as follows:

Basic types, array and channel types: The basic PROMELA types bool, byte, short, and int are modeled by the sets $\textbf{\textit{Bool}} = \{0, 1, \bot\}$, $\textbf{\textit{Byte}} = \{0, \ldots, 255, \bot\}$, $\textbf{\textit{Short}} = \{-2^{15}, \ldots, 2^{15} - 1, \bot\}$, and $\textbf{\textit{Int}} = \{-2^{31}, \ldots, 2^{31} - 1, \bot\}$. The symbol $\bot$ represents an undefined value. Moreover, there is a type $\textbf{\textit{Undef}} = \{\bot\}$. A channel variable type is a natural number, defined by $\textbf{\textit{ChanId}} := \mathbb{N} \cup \{\bot\}$. This number is not the channel itself, but the channel identifier. The set of basic type names is specified by

$$\textbf{\textit{BASE}} = \{\texttt{bool}, \texttt{byte}, \texttt{short}, \texttt{int}, \texttt{chan}, \texttt{undef}\}$$

We use a function

$$eval : \textbf{\textit{BASE}} \to \{\textbf{\textit{Bool}}, \textbf{\textit{Byte}}, \textbf{\textit{Short}}, \textbf{\textit{Int}}, \textbf{\textit{ChanId}}, \textbf{\textit{Undef}}\}$$

to map each type name to the corresponding set of values. Arrays are mappings from an index set to a basic type, e.g., an array of booleans b[2] is modeled as a mapping $f : \{0, 1\} \to \textbf{\textit{Bool}}$. The set

$$\textbf{\textit{ARR}}_n = \{\texttt{arr}(n, T) \mid T \in \textbf{\textit{BASE}}\}$$

holds all types of arrays of length n, and $\textbf{\textit{ARR}} := \bigcup_{n\in\mathbb{N}} \textbf{\textit{ARR}}_n$ is the set of all array types. The set of arrays is

$$\textbf{\textit{VARR}} = \{\{0, \ldots, n-1\} \to eval(T) \mid \texttt{arr}(n, T) \in \textbf{\textit{ARR}}\}$$

and an element $f \in \textbf{\textit{VARR}}$ is denoted by $[v_0, \ldots, v_{n-1}]$, where $v_i = f(i)$, for all $i \in dom(f)$. The set of all types a variable can have and the set of all possible values are defined by

$$\textbf{\textit{Type}} := \textbf{\textit{BASE}} \cup \textbf{\textit{ARR}} \qquad \textbf{\textit{Val}} := \bigcup_{T\in\textbf{\textit{BASE}}} eval(T) \cup \textbf{\textit{VARR}}$$

The set Name contains all legal variable names. A function $f : \mathsf{Name} \to \boldsymbol{Type} \times \boldsymbol{Val}$ is used to map each variable to a pair (T, v), where T is the variable type and v its value. The set of all such assignment functions is denoted by ***Asgn***.

We do not describe the rules for expression evaluation here. The reader is referred to [59]. Given assignment functions $\mathcal{L}$ and $\mathcal{G}$, we use an evaluation function that, given an expression, provides the corresponding value (looking first in function $\mathcal{L}$ and then in $\mathcal{G}$, if some variable is not defined in $\mathcal{L}$):

$$\boldsymbol{ev}_{\mathcal{L},\mathcal{G}} : \mathsf{Expr} \to \boldsymbol{Val}$$

In PROMELA's syntax, it is possible to also define enumerations and constants. To keep the semantics simpler, we do not treat this case, since we can consider a pre-processing phase in which constants and enumeration types are substituted by their corresponding values.

Channels: A channel is described by a tuple composed of the type of its messages, its contents, and its capacity. A message type is a structure composed of basic PROMELA types, defined as

$$\boldsymbol{MsgType}_n := \{(Type_1, \dots, Type_n) \mid Type_i \in \boldsymbol{BASE}, 1 \le i \le n\}$$

and the set of all message types is $\boldsymbol{MsgType} := \bigcup_{n\in\mathbb{N}} \boldsymbol{MsgType}_n$. A message is thus a tuple of values according to some message type. The set of messages is

$$\boldsymbol{Msg} = \bigcup_{n\in\mathbb{N}} \{Set_1 \times \dots \times Set_n \mid (Type_1, \dots, Type_n) \in \boldsymbol{MsgType}, Set_i = eval(Type_i), 1 \le i \le n\}$$

Given a message m (an element of Msg) and a message type definition T, we say that message m is of type T, denoted by $type(m, T)$ if each element of the tuple m has the corresponding type in tuple T. The channel type is the set

$$\boldsymbol{ChanType} = \boldsymbol{MsgType} \times \boldsymbol{Msg}^* \times \mathbb{N}_\perp,$$

with $(T, msgs, max) \in \boldsymbol{ChanType}$ iff for each $msg \in msgs$ $type(msg, T)$ and $|msgs| \le max$. A channel definition

$$\mathcal{C} : \boldsymbol{ChanId} \to \boldsymbol{ChanType}$$

translates each channel identifier in a real channel. The set of all channel definitions is denoted by ***Channel***. Thus, for a given channel identifier c, $\mathcal{C}(c)$ gives the type of the entries of c, its contents and its capacity. If c is not initialized, then $\mathcal{C}(c) = (\mathtt{undef}, \perp, 0)$. If there is no channel c, then $\mathcal{C}(c) = \perp$. The semantics of expressions and type checking are not presented here. They can be found in [59].

A.2. (Local) Process Behavior — SOS Rules

The SOS rules that describe the local behavior of PROMELA processes are given in Figs. A.9–A.11. For the SOS rules, we assume that $\varepsilon; \pi = \pi = \pi; \varepsilon$, and that $\varepsilon \cdot \gamma = \gamma$ (that is, empty commands have no effect in composing statements and in the continuation stack.

The functions *compat* and *type* used in rules (*RCV*) and (*SND*) check for type compatibility and return variable types, respectively.

A.3. Global behavior — SOS rules

The global semantic rules describe the semantics for global declarations and execution. The global execution of model is initially described by the relation $\to_P$ (see Fig. A.12), which expresses that the global execution can only occur if there is an active process in the model that can be executed (if a local transition is possible, a $\to_P$ transition is possible having as label the label of the local transition indexed by the identifier of the executing process). Then, the complete semantics is given by the relation $\to$, that controls the interleaving of (local) executions taking into account atomic blocks.

Appendix B. Auxiliary functions

Table B.1 provides an informal overview of auxiliary functions used in the translation. In order to define name formats for the variables in the PROMELA model that do not conflict, we use certain names, derived from the components being translated. In particular, we use: *className* to denote an OBGG class name; *msgName* to describe an OBGG message name; *ruleName* to describe an OBGG rule name; *parName* to describe the parameter name for an OBGG message.

$$\frac{}{(\texttt{t}\ \ \texttt{x=e}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x=e}_L (\varepsilon, \mathcal{L}[x \mapsto (t, ev_{\mathcal{L},\mathcal{G}}[\![\texttt{e}]\!])], \mathcal{G}, \mathcal{C}, \gamma)}\ (VARD)$$

$$\frac{\text{let } n := ev_{\mathcal{L},\mathcal{G}}[\![\texttt{c}]\!],\ \text{let } I := \{0, \ldots, n-1\},\ \text{let} f : I \to t : \forall i \in I. f(i) = ev_{\mathcal{L},\mathcal{G}}[\![\texttt{e}]\!]}{(\texttt{t}\ \ \texttt{x[c] = e}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x[c]=e}_L (\varepsilon, \mathcal{L}[\texttt{x} \mapsto (I \to t, f)], \mathcal{G}, \mathcal{C}, \gamma)}\ (AARD)$$

$$\frac{(\texttt{t}\ \ \texttt{x=0}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x=0}_L (\varepsilon, \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}{(\texttt{t}\ \ \texttt{x}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x}_L (\varepsilon, \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}\ (UVARD)$$

$$\frac{(\texttt{t}\ \ \texttt{x[c]=0}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x[c]=0}_L (\varepsilon, \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}{(\texttt{t}\ \ \texttt{x[c]}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x[c]}_L (\varepsilon, \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}\ (UARRD)$$

$$\frac{\begin{array}{l}\forall i \in \{1, \ldots, n\}.\\ (\texttt{t}\ \ \texttt{ivi;...;t}\ \ \texttt{ivn}, \mathcal{L}_i, \mathcal{G}_i, \mathcal{C}_i, \gamma_i) \xrightarrow{t\ iv_i}_L (\texttt{t}\ \ \texttt{ivi+1;...;t}\ \ \texttt{ivn}, \mathcal{L}_{i+1}, \mathcal{G}_{i+1}, \mathcal{C}_{i+1}, \gamma_{i+1})\end{array}}{(\texttt{t}\ \ \texttt{iv1,...,ivn}, \mathcal{L}_1, \mathcal{G}_1, \mathcal{C}_1, \gamma_1) \xrightarrow{t\ iv_1,\ldots,iv_n}_L (\varepsilon, \mathcal{L}_{n+1}, \mathcal{G}_{n+1}, \mathcal{C}_{n+1}, \gamma_{n+1})}\ (MULD)$$

$$\frac{\text{let } \mathcal{L}' := \mathcal{L}[\texttt{x} \mapsto (CHAN, \perp)]}{(\texttt{chan}\ \ \texttt{x}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\text{chan } x}_L (\varepsilon, \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}\ (UCHAND)$$

$$\frac{\text{let } \mathcal{L}' := \mathcal{L}[\texttt{x} \mapsto (CHAN, |\mathcal{C}|)], \text{let } \mathcal{C}' := \mathcal{C} + (\texttt{t1} \times \ldots \times \texttt{tn}, \varepsilon, ev_{\mathcal{L},\mathcal{G}}[\![\texttt{c}]\!])}{(\texttt{chan}\ \ \texttt{x=[c]}\ \ \texttt{of \{t1, ...,tn\}}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\text{chan } x=[c] \text{ of } \{t1,\ldots,tn\}}_L (\varepsilon, \mathcal{L}', \mathcal{G}, \mathcal{C}', \gamma)}\ (CHAND)$$

Fig. A.9. SOS-Rules for Local Semantics: Declarations.

Table B.1
Informal description of auxiliary functions used in the translation.

Function	Description
LMSG	Returns the list of all messages of the source OBGG model *GG*. The format used for each message name is msg_*className_msgName*.
LRULE	Returns the list of all rule names of the source OBGG model *GG*. The format used for each message name is rule_*className_ruleName*.
PARTYPE$_{class}$	Returns a list containing tuples of types. Each tuple corresponds to the types of parameters that a message targeted to the given *class* may contain.
ATRIB$_{obj}$	Returns the list of all values of attributes of the given object *obj*.
MSGPARVAL$_{msg}$	Given a message edge *msg*, returns a list with the values of the corresponding message parameters.
PAROBJ$_{class}$	Returns the list of all parameter names used in messages of the *class*. The format for the parameter names is par_*className_msgName_parName*.
TYPEPAROBJ$_{class}$	Returns the list of types of all parameter names used in messages addressed to *class*.
DECL$_{class}$	Returns the list of variables used in the OBGG rules for the *class* (variables of the term algebras associated to the rules of *class*).
TESTATR$_{nr,class}$	Returns a conjunction of conditions over the attributes of the class that must be satisfied for the rule application *nr*.
UPDATEATR$_{nr,class}$	Returns a list of updates over the attributes of the class that occur due to the rule application *nr*.
CREATEOBJ$_{nr,class}$	Returns the list fo commands for creating objects, if needed, as a result of the rule application *nr*.
SENDMSG$_{nr,class}$	Returns the list of command for sending messages as a result of the rule application *nr*.
ATROBJ$_{class}$	Provides a list of pairs type/name of attributes of objects of the given *class*.

Appendix C. Formal definitions of translations between states

Given a global state $(\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at)$, we use the following notation:

$\mathcal{L}_{class.c}$ denotes the local variable assignment of the process corresponding to the object of class *class* with external channel c (there can be only one process with such a configuration). Formally, $\mathcal{L}_{class.c}$ is defined by

$$\mathcal{L}_{class.c} = \mathcal{L} \text{ such that } (\pi, \mathcal{L}, \gamma) \in rng(act) \wedge \mathcal{L}(\texttt{opc\_}class) = (\texttt{chan}, c)$$

$$\frac{\text{let } (\mathcal{L}', \mathcal{G}') := (\mathcal{L}, \mathcal{G})[x \mapsto ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{e}]\!]]}{(\mathtt{x:=e}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{x:=e}_L (\varepsilon, \mathcal{L}', \mathcal{G}', \mathcal{C}, \gamma)}\ (\mathit{VASGN})$$

$$\frac{\text{let } (\mathcal{L}', \mathcal{G}') := (\mathcal{L}, \mathcal{G})[\mathtt{x} \mapsto f[ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{e1}]\!] \mapsto ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{e2}]\!]],\ f \in \boldsymbol{VARR}_n,\ 0 \le i < n}{(\mathtt{x[e1]\ :=\ e2}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{x[e1]:=e2}_L (\varepsilon, \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma)}\ (\mathit{AASGN1})$$

$$\frac{e \in \text{Expr},\ ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{e}]\!] \neq 0}{(\mathtt{e}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{e}_L (\varepsilon, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)}\ (\mathit{EXPR1})$$

$$\frac{ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{e}]\!] = 0}{(\mathtt{e}; \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\text{violation}}_L (\text{error}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)}\ (\mathit{EXPR2})$$

$$\frac{\begin{array}{l} ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{x}]\!] \neq \bot, \text{let } id := ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{x}]\!],\ \text{let } (T, \alpha, k) := \mathcal{C}(id) \\ |\alpha| > 0, \text{let } \alpha = (v_1, \ldots, v_r) \cdot \alpha' : \forall i \in \{1, \ldots, min(r, s)\}.compat_{\mathcal{L},\mathcal{G}}(\mathtt{ei}, v_i) == \text{tt} \\ \text{let } (\mathcal{L}', \mathcal{G}') := (\mathcal{L}, \mathcal{G})[\mathtt{ei} \mapsto v_i | \mathtt{ei} \in \text{VarRef}], \text{let } \mathcal{C}' := \mathcal{C}[id \mapsto (T, \alpha', k)] \end{array}}{(\mathtt{x?e1,\ .\ .\ .\ ,es}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{x?e1,\ldots,es}_L (\varepsilon, \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma)}\ (\mathit{RCV})$$

$$\frac{\begin{array}{l} ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{x}]\!] \neq \bot, \text{let } id := ev_{\mathcal{L},\mathcal{G}}[\![\mathtt{x}]\!],\ \text{let } (T, \alpha, k) := \mathcal{C}(id) \\ |\alpha| < k, \text{let } T = T_1 \times \ldots \times T_r, \forall i \in \{1, \ldots, min(r, s)\} : type_{\mathcal{L},\mathcal{G}}(\mathtt{ei}) = T_i \\ \text{if } r \le s,\ \text{let } \alpha' := \alpha \cdot (\mathtt{e1}, \ldots, \mathtt{es}, 0, \ldots, 0) \\ \text{if } r > s, \text{let } \mathcal{C}' := \mathcal{C}[id \mapsto (T, \alpha', k)] \end{array}}{(\mathtt{x!e1,\ .\ .\ .\ ,es}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{x!e1,\ldots,es}_L (\varepsilon, \mathcal{L}, \mathcal{G}, \mathcal{C}', \gamma)}\ (\mathit{SND})$$

$$\frac{}{(\mathtt{run\ x(e1,\ .\ .\ .\ ,en)}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\text{run } x(e1,\ldots,en)}_L (\varepsilon, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)}\ (\mathit{RUN})$$

$$\frac{stat \in L, (\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat}_L (\pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}{(\mathtt{atomic}\{\pi\}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\text{atomic}}_L (\pi; \mathtt{end\_atomic}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)}\ (\mathit{ATOM})$$

$$\frac{}{(\mathtt{end\_atomic}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\text{end\_atomic}}_L (\varepsilon, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)}\ (\mathit{EATOM})$$

Fig. A.10. SOS-Rules for Local Semantics: Statements.

$$\frac{stat \in L, (\pi_i, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat}_L (\pi_i', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}{(\mathtt{if\ ::}\pi_1\ \mathtt{::...\ ::}\pi_n\ \mathtt{fi}; \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat}_L (\pi_i'; \pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}\ (\mathit{IF})$$

$$\frac{}{(\mathtt{do}\ \pi\ \mathtt{od}; \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\tau}_L (\mathtt{if}\ \pi\ \mathtt{fi;\ do}\ \pi\ \mathtt{od}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \pi' \cdot \gamma)}\ (\mathit{DO})$$

$$\frac{\gamma = \pi \cdot \gamma'}{(\mathtt{break}; \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{break}_L (\pi, \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}\ (\mathit{BREAK})$$

Fig. A.11. SOS-Rules for Local Semantics: Control.

$ext\_message_{class.c.i}$ denotes the message with name *message* that is in the *i*th position in the external channel *c* of an object of class *class*. Formally,

$$ext\_message_{class.c.i} = buf[i] \text{ such that } \mathcal{C}(c) = (type, buf, len) \wedge message = buf[1](1)$$

$int\_message_{class.c.i}$ denotes the message with name *message* that is in the *i*th position in the array of internal channel for the object of class *class* with external channel *c*. This message is a structure in which the first element is the name of the message (actually, the corresponding number) and the other elements are the parameters of messages of this class. Remember that, for a list of vales *buf*, $buf[j]$ is its *j*th element and for a structure *s*, we denote by $s(j)$ its *j*th element. Formally,

$$\frac{\text{let}\, f : \{1,\ldots,n\} \to \mathsf{Name} \times \textbf{\textit{Type}}, \forall i \in \{1,\ldots,n\}}{(\texttt{proctype}\, x(t_1\, x_1,\ldots,t_n\, x_n)\{\pi\}\pi', \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{\texttt{proctype}\, x(t_1\, x_1,\ldots,t_n\, x_n)\{\pi\}} (\pi', \mathcal{G}, \mathcal{C}, pdef[x \mapsto (\pi, f)], act, at)} \quad (PTYPE)$$

$$\frac{\text{let}\, f : \{\} \to \mathsf{Name} \times \textbf{\textit{Type}}, \text{ let } act' := act[0 \mapsto (\pi, \mathcal{L}_{\perp}, \varepsilon)]}{(\text{init}\{\pi\}\pi', \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{\texttt{init}\{\pi\}} (\pi', \mathcal{G}, \mathcal{C}, pdef[\texttt{init} \mapsto (\pi, f)], act', at)} \quad (INIT)$$

$$\frac{\text{let}\, act(i) = (\pi, \mathcal{L}, \gamma), stat \in L, (\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat}_L (\pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}{(\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{stat.i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act[i \mapsto (\pi', \mathcal{L}', \gamma')], at)} \quad (SPEC)$$

$$\frac{\begin{array}{l} at = \perp, stat \in L - \{\texttt{run}\, x(\ldots), \texttt{atomic}, \texttt{end\_atomic}\}.i \in \mathbb{N} \\ (\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{stat.i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', at) \end{array}}{(\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{stat} (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', at)} \quad (EXEC1)$$

$$\frac{\begin{array}{l} at = i, stat \in L - \{\texttt{atomic}, \texttt{end\_atomic}\}, \\ (\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{stat.i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', at) \end{array}}{(\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{stat} (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', at)} \quad (EXEC2)$$

$$\frac{\begin{array}{l} (\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{\texttt{run}\, x(e_1,\ldots,e_n).i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', at) \\ \text{let } (\pi, f) := pdef(x), act(i) = (\pi_0, \mathcal{L}_0, \gamma_0), \forall i \in \{1,\ldots,n\}.\text{let } v_i := ev_{\mathcal{L}_0,\mathcal{G}}[|e_i|], (t_i, x_i) := f(i) \\ \text{let } \mathcal{L}' := \mathcal{L}_{\perp}[x_i \mapsto (t_i, v_i) \mid i \in \{1,\ldots,n\}], \ \text{let } act'' := act' + (\pi, \mathcal{L}', \varepsilon) \end{array}}{(\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{\texttt{run}\, x(e_1,\ldots,e_n)} (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act'', at)} \quad (RUN)$$

$$\frac{at = \perp, i \in \mathbb{N}, \ (\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{\texttt{atomic}.i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', at)}{(\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{\texttt{atomic}} (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', i)} \quad (AT1)$$

$$\frac{at = i, \ (\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{\texttt{end\_atomic}.i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', at)}{(\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, at) \xrightarrow{\texttt{end\_atomic}} (\varepsilon, \mathcal{G}', \mathcal{C}', pdef, act', \perp)} \quad (AT2)$$

Fig. A.12. SOS-Rules for Global Semantics.

$int\_message_{class.c.i} = buf[1]$ such that $\mathcal{L}_{class.c}(\texttt{opb\_}class) = (\texttt{arr(n,chan)}, f) \wedge f(i) = bc \wedge \mathcal{C}(bc) = (type, buf, len) \wedge message = buf[1](1)$

message$_{class.c}$ denotes the set of messages for the object of class *class* with external channel c:

$$message_{class.c} = \{m \mid \exists i.(m = ext\_message_{class.c.i} \vee m = int\_message_{class.c.i}) \wedge m \neq \varepsilon\}$$

$L_{PROMELA}$ is the set of names used in the attributes of objects or names of message parameters in the PROMELA model;
makeAttrList is a function that makes a list of vertices out of a set of attributes and a global PROMELA state. Essentially, it takes each element of the set of attributes, checks whether it is a basic value, a reference, or an array, and puts the corresponding vertex in the list of vertices;
makeParList is analogous to the previous function, but takes a list of message values as parameter;
makeLabList this function is constructed like the previous ones, but the output is a list of attributes or parameter names. This function must be compatible with the previous ones in the sense that the order of names generated by *makeLabList* is the same as the order of corresponding nodes generated by *makeAttrList* or *makeParList*.

Definition 25 (*Transformation from PROMELA Well-Formed States to OB-Graphs*)**.** Given an OBGG *OG* over the class graph C and its transformation into a PROMELA model Π, the transformation of a well-formed state $S = (\varepsilon, \mathcal{G}, \mathcal{C}, pdef, act, \perp) \in \textbf{WFState}_{\Pi}$ to an OB^C-graph H^C is given by the function $\mathcal{T}_{P\to G} : \textbf{WFState}_{\Pi} \to \textbf{OBGraph}(\textbf{C})$ defined by $\mathcal{T}_{P\to G}(S) = (H, type^H)$, where

- $H = (V_H, E_H, s^H, t^H, Lab_H, lab^H, A_H, a^H)$
  - $V_H = objV_H \cup valV_H$, with $objV_H = \{class.c \mid \exists \mathcal{L}_{class.c}\}$ and $valV_H = \mathcal{U}(A_{PROMELA}) = \textbf{\textit{Val}}$

- $E_H = msgE_H \cup atrE_H$, with $msgE_H = \bigcup_{\mathcal{L}_{class.c}} message_{class.c}$ and
  $atrE_H = \{class.c \mid \exists a \in dom(\mathcal{L}_{class.c}).prefix(a) = \text{“}\mathtt{atr}\text{”}\}$
- $t^H : E_H \to objV_H$, defined for all $e \in E_H$ by
  $t^H(e) = \begin{cases} class.c & \text{if } e \in msgE_H \text{ where } e = int\_message_{class.c.i} \vee e = ext\_message_{class.c.i} \\ e & \text{if } e \in atrE_H \end{cases}$
- $s^H : E_H \to V_H^*$, defined for all $e \in E_H$ by
  $s^H(e) = \begin{cases} makeAttrList(A, S) & \text{if } e = class.c, \text{ where } A = \{at \in dom(\mathcal{L}_{class.c}) \mid prefix(at) = \mathtt{atr}\} \\ makeParList(P, S) & \text{if } e = x\_message_{class.c.i} = (message, p_1, \ldots, p_n), \text{ with } x \in \{int, ext\} \wedge \\ & P = \{par(i) \mid 1 \le i \le n \wedge par(i) \ne \bot\} \end{cases}$
- $Lab_H = L_{PROMELA}$
- $lab^H : e_H \to L^*_{PROMELA}$, defined for all $e \in E_H$ by
  $lab^H(e) = \begin{cases} makeLabList(A) & \text{if } e = class.c, \text{ where } A = \{at \in dom(\mathcal{L}_{class.c}) \mid prefix(at) = \mathtt{atr}\} \\ makeLabList(P) & \text{if } e = x\_message_{class.c.i} = (message, p_1, \ldots, p_n), \text{ with } x \in \{int, ext\} \wedge \\ & P = \{par(i) \mid 1 \le i \le n \wedge par(i) \ne \bot\} \end{cases}$
- $A_H = A_{PROMELA}$
- $a^H : valV_H \to \mathcal{U}(A_{PROMELA})$, defined for all $v \in valV_H$ by
  $a^H(v) = v$

- $type^H = ((type^H_V, type^H_E, type^H_A) : H \to C$
  - $type^H_V : \forall v \in V_H.type^H_V(v) = class$, if $v = class\_c$ or $type^H_V(v) = s$, if $v \in A_{H_s}$;
  - $type^H_E : \forall e \in E_H.type^H_E(e) = message$, if $e = x\_message_{class.c.i}$, with $x \in \{int, ext\}$, or $type^H_E(e) = class$, if $e = class.c$;
  - $type^H_A : \forall v \in A_{H_s}.type^H_A(v) = s$

Let $(H, type^H)$ be an OB^C-graph, for the following definitions, we use the notation

$elems_H$ is an enumeration containing an element obj for each object of H ($obj \in objV_H$), an element obj_n for each object of H and $1 \le n \le bsize$. We denote by $elem_H(i)$ the ith element of this enumeration;

M_{obj} denotes a list of messages that may be received by objects obj (message edge $e \in msgV_C$ such that $t^C(e) = type^H(obj)$);

msg_type_{obj} denotes the pattern for any message sent to an object obj: $msg\_type_{obj} = (\mathtt{mtype}, p_1 \ldots p_m)$, with p_1 to p_i being the types of sources of message $M_{obj}(1)$, $p_{i+1}top_j$ being the types of targets of message $M(2)$, and so on. The source of a message edge is a list of nodes, its source is an attribute node v, the corresponding p is the name of carrier set associated to this node ($type^H(a^H(v))$), if it is an object, the corresponding p is $\mathtt{chan}$;

$msgs_{obj}$ denotes a list of messages for object obj. Each message corresponds to a message edge $e \in msgE_H$ connected to obj and has the form $(msg, p_1 \cdots p_n)$, where msg is the name of the message (given by $type^H(e)$), and the p_is are constructed as follows: for all i that are outside the range of $M(j) = msg$, the value is $\bot$; for each i in the range, the corresponding value must be obtained from the source of e (recall that $s^H(e)$ is a list). If a tentacle of e points to an attribute node, the corresponding p is the value associated to this attribute node, if it points to an object, the value will be the position of obj in $elems_H$.

Definition 26 (*Transformation from OB^C-Graphs to PROMELA Well-Formed States*). Given an OBGG OG over the class graph C, a rule name nr (a rule name or zero), a maximum buffer size $bsize$ and the transformation of OG into a PROMELA model Π, the transformation of an OB^C-graph $(H, type^H)$ into a PROMELA WF-state $S = (\pi, \mathcal{G}, \mathcal{C}, pdef, act, at) \in \mathbf{WFState}_\Pi$ is given by the function $\mathcal{T}_{G\to P} : \mathbf{OBGraph(C)} \to \mathbf{WFState}_\Pi$ defined by $\mathcal{T}_{G\to P}(H, type^H) = S$, where

- $\pi = \varepsilon$;
- $\mathcal{G}(\mathtt{event\_RuleName} = (\mathtt{mtype}, nr)$;
- $\mathcal{C}(i) = \begin{cases} (msg\_type_{obj}, msgs_{obj}, bsize) & \text{if } elem_H(i) = obj; \\ (msg\_type_{obj}, \varepsilon, 1) & \text{if } elem_H(i) = obj_n \end{cases}$, for $0 \le i \le |elems_H|$.
- $pdef(class) = (\pi_{class}, par_{class})$, for all $class \in objV_C$, where π_{class} is the code for the body of Definition 17 (instantiated for $class$) and par_{class} is defined as

$$par_{class}(i) = \begin{cases} (\mathtt{opc\_}class), \mathtt{chan}), & \text{if } i = 0; \\ (\mathtt{atr\_}name, type), & \text{otherwise, with } name \in lab?H(e) \text{ at position } i,\ e \in atrE_C, \\ & t^C(e) = class,\ type \text{ corresponds to the type of the vertex} \\ & \text{pointed by the } i\text{th tentacle of } e \end{cases}$$

- $act(i) = (\pi_{obj}, \mathcal{L}_{obj}, \varepsilon)$, for all i that is a position of an obj in $elems_H$, where π_{obj} is the code fragment `do ::atomic` $\{\pi_{msg}\}$ `:: atomic` $\{\pi_{app}\}$ `od` from Definition 17 (instantiated for object obj) and $\mathcal{L}_{obj}$ is defined for

$var \in \{\texttt{opc\_}class, \texttt{opb\_}class, \texttt{atr\_}name\}$, where $\exists obj \in objV_H, e \in atrE_H.type^H(obj) = class$ and $t^H(e) = obj, name \in lab(e)$:

$$\mathcal{L}_{obj}(var) = \begin{cases} (\texttt{chan}, c), & \text{if } var = \texttt{opc\_}class \wedge \exists obj.type^H(obj) = class, \\ & c \text{ is the position of } obj \text{ in } elems_H \\ (\texttt{arr(BSIZE,chan)}, l) & \text{if } var = \texttt{opb\_}class \wedge \exists obj.type^H(obj) = class, \\ & l \text{ is a list of channels corresponding to positions on } obj_n \text{ in } elems_H \\ (type, val) & \text{if } var = \texttt{atr\_}name \wedge \exists obj \in objV_H, e \in atrE_H.t^H(e) = obj, \\ & name \in lab(e) \text{ at position } i, type \text{ corresponds to the type of the} \\ & \text{vertex pointed by the } i\text{th tentacle of } e.val \text{ is the value of the} \end{cases}$$

- $at = \perp$

References

[1] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific Publishing Co, 1997.
[2] F.L. Dotti, L. Ribeiro, Specification of mobile code systems using graph grammars, in: 4th International Conference on Formal Methods for Open Object-Based Distributed Systems, in: IFIP Conference Proceedings, vol. 177, Kluwer, USA, 2000, pp. 45–63.
[3] H. Ehrig, Introduction to the Algebraic Theory of Graph Grammars, in: 1st Graph Grammar Workshop, in: Lecture Notes in Computer Science (LNCS), vol. 73, Springer, 1979, pp. 1–69.
[4] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools, World Scientific Publishing Co, 1999.
[5] H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 3: Concurrency, Parallelism, and Distribution, World Scientific Publishing Co, 1999.
[6] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, A. Schiper, S.S.S. Krakowiak, Consensus in asynchronous distributed systems: a concise guided tour, in: Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems, in: Lecture Notes in Computer Science (LNCS), Springer, 1999, pp. 33–47.
[7] F.L. Dotti, L. Foss, L. Ribeiro, O.M. Santos, Especificação e verificação formal de sistemas distribuídos, in: 17^o Simpósio Brasileiro de Engenharia de Software, Brasil, 2003, pp. 225–240 (in Portuguese).
[8] O.M. Santos, F.L. Dotti, L. Ribeiro, Verifying object-based graph grammars, Electronic Notes in Theoretical Computer Science 109 (2004) 125–136.
[9] F. Dotti, L. Ribeiro, O. dos Santos, F. Pasini, Verifying object-based graph grammars: an assume-guarantee approach, Software and Systems Modeling 5 (3) (2006) 289–311. doi:10.1007/s10270-006-0014-z.
[10] B. Copstein, M.C. Móra, L. Ribeiro, An environment for formal modeling and simulation of control systems, in: 33rd Annual Simulation Symposium, IEEE Computer Society, USA, 2000, pp. 74–82.
[11] F.L. Dotti, L.M. Duarte, B. Copstein, L. Ribeiro, Simulation of mobile applications, in: Communication Networks and Distributed Systems Modeling and Simulation Conference, SCS, USA, 2002, pp. 261–267.
[12] O.M. Mendizabal, F.L. Dotti, L. Ribeiro, Stochastic object-based graph grammars, Electronic Notes in Theoretical Computer Science 184 (2007) 151–170. doi:10.1016/j.entcs.2007.03.020.
[13] B. Plateau, K. Atif, Stochastic automata network of modeling parallel systems, IEEE Transactions on Software Engineering 17 (10) (1991) 1093–1108. doi:10.1109/32.99196.
[14] L.R. Leonardo Michelon, Simone André da Costa, Formal specification and verification of real-time systems using graph grammars, Journal of the Brazilian Computer Society 13 (4) (2007) 51–68.
[15] F. Pasini, F.L. Dotti, Code generation for parallel applications modelled with object-based graph grammars, Electronic Notes on Theoretical Computer Science 184 (2007) 113–131. doi:10.1016/j.entcs.2007.03.018.
[16] M. Snir, S.W. Otto, D.W. Walker, J. Dongarra, S. Huss-Lederman, MPI: The Complete Reference, MIT Press, Cambridge, MA, USA, 1995.
[17] F.L. Dotti, L. Ribeiro, O.M. Santos, Specification and analysis of fault behaviours using graph grammars, in: Applications of Graph Transformations with Industrial Relevance, AGTIVE 2003, in: Lecture Notes in Computer Science (LNCS), vol. 3062, Springer, 2004, pp. 120–133.
[18] F.L. Dotti, O.M. Mendizabal, O.M. dos Santos, Verifying fault-tolerant distributed systems using object-based graph grammars, in: Dependable Computing, Second Latin-American Symposium, LADC 2005, in: Lecture Notes in Computer Science (LNCS), vol. 3747, Springer, 2005, pp. 80–100.
[19] F.C. Gärtner, Transformational approaches to the specification and verification of fault-tolerant systems: formal background and classification, Journal of Universal Computer Science 5 (10) (1999) 668–692.
[20] L. Ribeiro, F.L. Dotti, R. Bardohl, A formal framework for the development of concurrent object-based systems, in: Formal Methods in Software and Systems Modeling, in: Lecture Notes in Computer Science (LNCS), vol. 3393, Springer, 2005, pp. 385–401.
[21] F.L. Dotti, L.M. Duarte, L. Foss, L. Ribeiro, D. Russi, O.M. Santos, An environment for the development of concurrent object-based applications, Electronic Notes in Theoretical Computer Science 127-1 (2005) 3–13.
[22] L. Duarte, F. Dotti, Development of an active network architecture using mobile agents — a case study, Tech. Rep. TR-043, FACIN - PPGCC - PUCRS, 2004.
[23] F.L. Dotti, L. Foss, L. Ribeiro, O.M. Santos, Verification of object-based distributed systems, in: 6th International Conference on Formal Methods for Open Object-based Distributed Systems, in: Lecture Notes in Computer Science (LNCS), vol. 2884, Springer, 2003, pp. 261–275.
[24] G.J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
[25] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580. doi:10.1145/363235.363259.
[26] R. Milner, An algebraic definition of simulation between programs, Tech. Rep. CS-TR-71-205, Stanford University, Stanford, CA, USA, 1971.
[27] R.M. Burstall, An algebraic description of programs with assertions, verification and simulation, SIGACT News 14 (1972) 7–14. doi:10.1145/942580.807068.
[28] S.L. Gerhart, Correctness-preserving program transformations, in: POPL'75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, New York, NY, USA, 1975, pp. 54–66. doi:10.1145/512976.512983.
[29] D. Harel, A. Puneli, J. Stavi, A complete axiomatic system for proving deductions about recursive programs, in: STOC'77: Proceedings of the ninth annual ACM symposium on Theory of computing, ACM, New York, NY, USA, 1977, pp. 249–260. doi:10.1145/800105.803415.
[30] C.B. Jones, The early search for tractable ways of reasoning about programs, IEEE Annals History of Computing 25 (2) (2003) 26–49. doi:10.1109/MAHC.2003.1203057.
[31] D.C. Schmidt, Guest editor's introduction: model-driven engineering, IEEE Computer 39 (2) (2006) 25–31.
[32] P. Baldan, A. Corradini, B. König, A framework for the verification of infinite-state graph transformation systems, Information and Computation 206 (7) (2008) 869–907.
[33] S.A. da Costa, L. Ribeiro, Formal verification of graph grammars using mathematical induction, Electronic Notes in Theoretical Computer Science 240 (2009) 43–60.

[34] A. Rensink, Á Schmidt, D. Varró, Model checking graph transformations: a comparison of two approaches, in: 2nd International Conference on Graph Transformation, ICGT 2004, in: Lecture Notes in Computer Science (LNCS), vol. 3256, Springer, Italy, 2004, pp. 226–241.
[35] H. Kastenberg, A. Rensink, Model checking dynamic states in GROOVE, in: Model Checking Software, 13th International SPIN Workshop, in: Lecture Notes in Computer Science (LNCS), vol. 3925, Springer, 2006, pp. 299–305.
[36] S. Leue, G. Holzmann, V-Promela: a visual, object oriented language for SPIN, in: 2nd International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Computer Society, USA, 1999, pp. 14–23.
[37] J. Lilius, I.P. Paltor, vUML: a tool for verifying UML models, in: 14th International Conference on Automated Software Engineering, IEEE Computer Society, USA, 1999, pp. 255–258.
[38] J. Chen, H. Cui, Translation from adapted uml to promela for corba-based applications, in: Model Checking Software, 11th International SPIN Workshop, in: Lecture Notes in Computer Science (LNCS), vol. 2989, Springer, 2004, pp. 234–251.
[39] D. Varró, Automated formal verification of visual modeling languages by model checking, Software and Systems Modeling 3 (2) (2004) 85–113.
[40] E. Börger, R. Stärk (Eds.), Abstract State Machines: A Method for High-Level System Design and Analysis, Springer, 2003.
[41] C. Demartini, R. Iosif, R. Sisto, Modeling and validation of Java multithreading applications using SPIN, in: G. Holzmann, E. Najm, A. Serhrouchni (Eds.), Proc. of the 4th SPIN workshop, France, 1998.
[42] J.C. Corbett, et al., Bandera: extracting finite-state models from Java source code, in: 22nd International Conference on Software Engineering, ACM Press, Ireland, 2000, pp. 439–448.
[43] G.D. Castillo, Towards comprehensive tool support for abstract state machines: the ASM workbench tool environment and architecture, in: International Workshop on Current Trends in Applied Formal Methods, in: Lecture Notes in Computer Science (LNCS), vol. 1641, Springer, 1999, pp. 311–325.
[44] K. Winter, R. Duke, Model checking Object-Z using ASM, in: 3rd International Conference on Integrated Formal Methods, in: Lecture Notes in Computer Science (LNCS), vol. 2335, Springer, 2002, pp. 165–184.
[45] M. Sirjani, A. Movaghar, A. Shali, F.S. de Boer, Modeling and verification of reactive systems using rebeca, Fundamenta Informticae 63 (4) (2004) 385–410.
[46] F. Alavizadeh, A.H. Nekoo, M. Sirjani, Reuml: a uml profile for modeling and verification of reactive systems, in: Proceedings of the Second International Conference on Software Engineering Advances, ICSEA 2007, IEEE Computer Society, 2007, p. 50.
[47] E.W. Dijkstra, Hierarchical ordering of sequential processes, Acta Informatica 1 (1971) 115–138.
[48] G. Ricart, A.K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, Communications of the ACM 24 (1) (1981) 9–17. doi:10.1145/358527.358537.
[49] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini, Algebraic approaches to graph transformation. part ii: single pushout approach and comparison with double pushout approach, in: Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations, World Scientific Publishing Co, 1997, pp. 247–312.
[50] A. Corradini, U. Montanari, F. Rossi, Graph processes, Fundamenta Informaticae 26 (3/4) (1996) 241–265.
[51] L. Ribeiro, Parallel composition and unfolding semantics of graph grammars, Ph.D. thesis, Technical University of Berlin, Germany, 1996.
[52] M.B. Dwyer, G.S. Avrunin, J.C. Corbett, Patterns in property specifications for finite-state verification, in: 21st International Conference on Software Engineering, IEEE Computer Society Press, USA, 1999, pp. 411–420.
[53] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems—Specification, Springer-Verlag, Germany, 1992.
[54] M. Chechik, D.O. Păun, Events in property patterns, in: 5th and 6th International SPIN Workshops, in: Lecture Notes in Computer Science (LNCS), vol. 1680, Springer, 1999, pp. 154–167.
[55] M.B. Dwyer, G.S. Avrunin, J.C. Corbett, Property specification patterns for finite-state verification, in: 2nd Workshop on Formal Methods in Software Practice, ACM Press, USA, 1998, pp. 7–15.
[56] F.L. Dotti, et al., An environment for the development of concurrent object-based applications, Electronic Notes in Theoretical Computer Science 127 (2005) 3–13.
[57] Research Bell-Labs, SPIN version 3.3: Language reference, 2003. http://spinroot.com/spin/Man/promela.html.
[58] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, USA, 1985.
[59] C. Weise, An incremental formal semantics for PROMELA, in: 3rd International SPIN Workshop, The Netherlands, 1997.
[60] R. Milner, Communication and Concurrency, Prentice Hall International Ltd, 1995.