Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 175 (2007) 101–114
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Formal Verification of Object-Oriented Graph Grammars Specifications

Ana Paula Lüdtke Ferreira[1]

Universidade do Vale do Rio dos Sinos
São Leopoldo, Brazil

Luciana Foss, Leila Ribeiro[2]

Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil

Abstract

Concurrent object-oriented systems are ubiquitous due to the importance of networks and the current demands for modular, reusable, and easy to develop software. However, checking the correctness of such systems is a hard task, mainly due to concurrency and inheritance aspects. In this paper we present an approach to the verification of concurrent object-oriented systems. We use graph grammars equipped with object oriented features (including inheritance and polymorphism) as the specification formalism, and define a translation from such specifications to Promela, the input language of the SPIN model checker.

Keywords: Graph grammars, object-oriented programming, model checking.

1 Introduction

Software development techniques have evolved over the years to deal with current developing demands. The paradigms on which those techniques are based (especially objects, events and concurrency) make the modeling and coding processes easier. However, testing and validation of such systems became more complex, mainly due to the nondeterministic behavior of multiple processes competing for the same resources. Object-oriented systems features — inheritance, polymorphism and dynamic binding of method calls — also make static analysis of limited use in the validation process. Thus, correctness assurance of concurrent object-oriented systems is a difficult task.

[1] Email:anap@unisinos.br

[2] Email:{lfoss,leila}@inf.ufrgs.br

1571-0661 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2007.04.020

The first step to enable correctness proofs of a system is to provide a formal specification of it. The semantic model can be analyzed to check whether the desired properties hold. The choice on which specification language to use depends on the application characteristics, but also on the development paradigm chosen. We suggest that, if an object-oriented development process is followed, an adequate formal specification formalism should offer compatible constructs. Object-oriented graph grammars were first presented in [8] as an extension of the algebraic single-pushout approach [13] to encompass object-oriented features such as inheritance, polymorphism, and dynamic binding. The main contribution of this paper is to present a verification method for specifications written as object-oriented graph grammars. This method is based on a translation of such specification to Promela programs. Promela is the input language of the SPIN model checker [10]. Particularly, features like inheritance, polymorphism, and dynamic binding will also be faithfully encoded in Promela.

Our approach for object-oriented verification is a straightforward, translation-based one, and differs from approaches relying on analysis of the specification languages *per se*, such as [2], [12], [1], [15], [16]. We follow a line of work presented in [5] for graph grammars without object-oriented features. However, besides building the translations for inheritance, polymorphism and dynamic binding features, we also do the translation based on a well defined observational semantics [7] which interprets object-oriented graph grammars computations from the object-oriented paradigm view. This article is structure as follows: Sec. 2 presents the main components of object-oriented graphs and grammars. Sec. 3 presents the guidelines followed for the definition of a formal translation from object-oriented graph grammars specifications into Promela programs, followed by an example shown in Sec. 4. The running example is a classic problem in the theory of concurrency, known as the *Dining Philosophers* problem. Final remarks are presented in Sec. 5.

2 Object-oriented graph grammars

Object-oriented systems consist of instances of previously defined classes having an internal structure defined by attributes and communicating among themselves through message passing. An object-oriented system state consists of objects, together with a set of messages yet to be consumed. Messages are the triggers of method executions, and their implementation may be redefined within derived classes. Classes and messages are modeled together in a *class-model graph*. Formally, class identifiers are graph nodes, attributes are modeled as hyperarcs (that is, each class may be connected to many others via an attribute hyperarc), and messages are also modeled as hyperarcs (in which the target is the destination of the message, and sources are its parameters). The inheritance hierarchy is defined by imposing a *strict relation* among the graph nodes. A strict relation is an irreflexive, acyclic, functional relation, with the additional property that there is no infinite chain of elements connected through it (the reflexive and transitive closure of a strict relation is a partial order [7]). Message hyperarcs also possess an order structure, which reflects the possibility of a derived object to override inherited methods of its superclasses. A set carrying a reflexive and transitive closure of a strict relation is called a *strict ordered set.*

Definition 2.1 [Class-model graph] A class-model graph is a tuple $\langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ where $V_{\sqsubseteq} = \langle V, \sqsubseteq_V^*\rangle$ is a strict ordered set of vertices, $E_{\sqsubseteq} = \langle E, \sqsubseteq_E^*\rangle$ is a strict ordered set of (hyper)edges, $L = \{\mathrm{attr}, \mathrm{msg}\}$ is an unordered set of two edge labels, $src, tar : E \to V^*$ are monotonic order-preserving functions, called respectively *source* and *target* functions, $lab : E \to L$ is the edge *labeling* function, such that the following constraints hold:

- *Structural constraints:* for all $e \in E$, the following holds:
  (i) if $lab(e) = \mathrm{attr}$ then $src(e) \in V$ and $tar(e) \in V^*$, and
  (ii) if $lab(e) = \mathrm{msg}$ then $src(e) \in V^*$ and $tar(e) \in V$.
- *Order relations constraints:* for all $e \in E$, the following holds:
  (i) if $(e, e') \in\ \sqsubseteq_E$ then $lab(e) = lab(e') = \mathrm{msg}$,
  (ii) if $(e, e') \in\ \sqsubseteq_E$ then $src(e) = src(e')$,
  (iii) if $(e, e') \in\ \sqsubseteq_E$ then $(tar(e), tar(e')) \in\ \sqsubseteq_V^+$, and
  (iv) if $(e', e) \in\ \sqsubseteq_E$ and $(e'', e) \in\ \sqsubseteq_E$, with $e' \neq e''$, then $(tar(e'), tar(e'')) \notin\ \sqsubseteq_V^*$ and $(tar(e''), tar(e')) \notin\ \sqsubseteq_V^*$.

Sets $\{e \in E \mid lab(e) = \mathrm{attr}\}$ and $\{e \in E \mid lab(e) = \mathrm{msg}\}$ are denoted by $E|_{\mathrm{attr}}$ and $E|_{\mathrm{msg}}$, respectively.

Structural constraints assure that hyperarcs modeling attributes and messages have the correct source and targets. Inheritance and overriding hierarchies are made explicit by imposing that graph nodes (i.e., classes) and message edges (i.e., methods) are strict ordered sets. Only single inheritance is allowed, since $\sqsubseteq_V$ is required to be a function. The relation between message arcs, $\sqsubseteq_E$, establishes which methods are overridden within the derived object, by mapping them. The restrictions applied to $\sqsubseteq_E$ ensure that methods are redefined consistently, i.e., only message arcs can be mapped (i), their parameters are the same (ii), the method being redefined is located somewhere (strictly) above in the class-model graph (under $\sqsubseteq_V^+$) (iii), and only the closest message with respect to relations $\sqsubseteq_V$ and $\sqsubseteq_E$ can be redefined (iv).

Example 2.2 The class-model graph in Figure 1 depicts an object-oriented system structure for the Dining Philosophers problem. Graph nodes represent classes: *Philosopher*, which is derived into two different types: *Left-HandedPhilosopher* and *Right-HandedPhilosopher* (the inheritance relation is pictured as a dotted arrow); *Fork*, to represent the shared resources the philosophers are competing upon; *Table*, to model both the place where the philosophers sit and from where forks can be picked up; and *ForkHolder*, which can be either a *Philosopher* or a *Table*. The attributes are the information the elements must possess to compute correctly: a *Philosopher* sits at a *Table*, has a left and a right *Fork* to get in order to eat; a *Fork* has an owner, which is a *ForkHolder*. Messages stand for the actions performed by the actors in the program. A *Fork* can be acquired by a *Philosopher*, and released

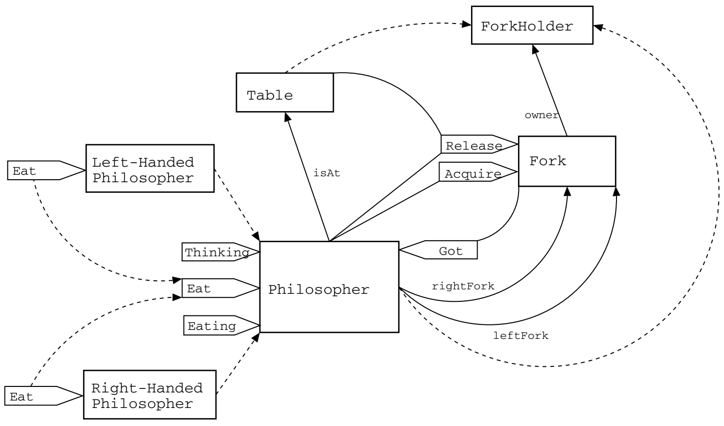

Figure 1. Class-model graph for the Dining Philosophers problem.

by a *Philosopher* to a *Table*. A *Philosopher* can be Thinking, Eating, or receive a message Eat, which sends him to the process of acquiring his forks, and a message Got, to notify that a *Fork* has been acquired. Left-handed and right-handed philosophers override message Eat, which is indicated by the lines connecting both hyperarcs.

Class-model graphs can be used as typing structures for states of object-oriented systems. Before defining such states, which will be *object-oriented graphs*, we will define how to map a graph into a class-model graph, and then impose restrictions to make this mapping compatible with inheritance. Based on the inheritance and overriding relations, we define auxiliary functions that, given a class identifier (node), return the sets of attributes (inherited or not) of this class, and the sets of messages (method triggers) that this class may receive.

Definition 2.3 [$\mathcal{C}$-typed graph] A $\mathcal{C}$-typed graph $G^{\mathcal{C}}$ is a tuple $\langle G, t, \mathcal{C}\rangle$, where $\mathcal{C} = \langle V_{\mathcal{C}_{\sqsubseteq}}, E_{\mathcal{C}_{\sqsubseteq}}, L, src_{\mathcal{C}}, tar_{\mathcal{C}}, lab_{\mathcal{C}}\rangle$ is a class-model graph, $G = \langle V_G, E_G, src_G, tar_G\rangle$ is a hypergraph, and t is a pair of total functions $\langle t_V : V_G \to V_{\mathcal{C}}, t_E : E_G \to E_{\mathcal{C}}\rangle$ such that $(t_V^* \circ src_G) \sqsubseteq_{V_{\mathcal{C}}^*} (src_{\mathcal{C}} \circ t_E)$, and $(t_V^* \circ tar_G) \sqsubseteq_{V_{\mathcal{C}}^*} (tar_{\mathcal{C}} \circ t_E)$. Moreover, we define:

- the *attribute set function* $attr_G : V_G \to 2^{E_G}$ returns for each vertex $v \in V_G$ the set of attribute edges with source v;
- the *message set function* $msg_G : V_G \to 2^{E_G}$ returns for each vertex $v \in V_G$ the set of message edges with target v.
- the *extended attribute set function*, $attr_{\mathcal{C}}^* : V \to 2^E$, where $attr_{\mathcal{C}}^*(v) = \{e \in E \mid lab(e) = \text{attr} \wedge src(e) \in \uparrow v\}$, and $\uparrow v$ is the set of all superclasses of v.
- the *extended message set function*, $msg_{\mathcal{C}}^* : V \to 2^E$, where $msg_{\mathcal{C}}^*(v) = \{e \in E|_{\text{msg}} \mid tar(e) \in \uparrow v \wedge \neg\exists e' \in E|_{\text{msg}} : tar(e') \in \uparrow v \wedge e' \sqsubseteq_E e\}$.

$\mathcal{C}$-typed graphs reflect the inheritance of attributes and methods from the object-oriented paradigm. They are ordinary hypergraphs typed over a class-model graph. However, the typing morphism is more flexible than the traditional one [3]: a $\mathcal{C}$-

typed graph edge e can be incident to any $\mathcal{C}$-typed graph node v as long as its typing edge $t_E(e)$ (in $\mathcal{C}$) is incident to a node type v' (also in $\mathcal{C}$), such that $t_V(v)$ and v' are connected under the inheritance order relation (i.e., $t_V(v) \sqsubseteq^*_{V_\mathcal{C}} v'$). This definition reflects the fact that an object can use any attribute belonging to one of its primitive classes, since it was inherited when the class was specialized.

The *extended attribute set function* returns the set of all attribute arcs whose source is v or any other vertex to which v connected via the inheritance relation $\sqsubseteq^*_V$. The *extended message set function* returns all messages an object of a specific type may receive. Notice that message redefinition within objects, expressed by the overriding relation $\sqsubseteq^*_E$ on the class-model graph, is taken into account, since only the redefined methods can be seen within the scope of a specialized class.

For a $\mathcal{C}$-typed graph $\langle G, t, \mathcal{C} \rangle$, let the total function $t^*_E : 2^{E_G} \to 2^{E_\mathcal{C}}$ be the extension of the typing function to edge (or node) sets. Notation $t^*_E|_{\mathrm{msg}}$ and $t^*_E|_{\mathrm{attr}}$ will be used to denote the application of t^*_E to sets containing exclusively message and attribute (respectively) hyperarcs. Now we can present a definition of the kind of graph which represents an object-oriented system.

Definition 2.4 [Object-oriented graph] Let $\mathcal{C}$ be a class-model graph. A $\mathcal{C}$-typed graph $\langle G, t, \mathcal{C} \rangle$ is an *object-oriented graph* if and only if all squares in the diagram below (in **Set**) commute. If, for each $v \in V_G$, the function $t^*_E|_{\mathrm{attr}}(attr_G(v))$ is injective, $G^\mathcal{C}$ is said a *strict* object-oriented graph. If $t^*_E|_{\mathrm{attr}}(attr_G(v))$ is also surjective, $G^\mathcal{C}$ is called a *complete* object-oriented graph.

$$\begin{tikzcd}
2^{E_G} \arrow[d, "t^*_E|_{\mathrm{msg}}"'] & V_G \arrow[l, "msg_G"'] \arrow[r, "attr_G"] \arrow[d, "t_V"'] & 2^{E_G} \arrow[d, "t^*_E|_{\mathrm{attr}}"] \\
2^{E_\mathcal{C}} & V_\mathcal{C} \arrow[l, "msg^*_\mathcal{C}"'] \arrow[r, "attr^*_\mathcal{C}"] & 2^{E_\mathcal{C}}
\end{tikzcd}$$

The left square on the diagram of Def. 2.4 ensures that a message edge can only target an object if it is typed over one of the edges returned by the extended message set function applied to the object type. It means that the only messages allowed are the least ones in the redefinition chain to which the typing message belongs. This is compatible with the notion of *dynamic binding*, since the method actually called by any object is determined by the actual object present at a certain computation state. Injectivity of all $t^*_E|_{\mathrm{attr}}(attr_G(v))$, $v \in V_G$, expresses that all attribute arcs are typed differently (i.e., an object has no exceeding attribute). Surjectivity means that *all* attributes defined on all levels along the class-model graph (via the inheritance relation on nodes) are present. The definition of a complete object-oriented graph is coherent with the notion of inheritance within the object-oriented framework, since an object inherits all attributes, and exactly those, from its primitive classes.

Example 2.5 Figure 2 shows a complete object-oriented graph, typed over the class-model graph portrayed in Figure 1. Let the three elements called Kant, Hegel and Nietzsche be *Right-HandedPhilosopher*, and the other elements be typed as their names indicate. According to the typing class-model graph, a *Right-HandedPhilosopher* has no attribute at all, and also does not receive a mes-

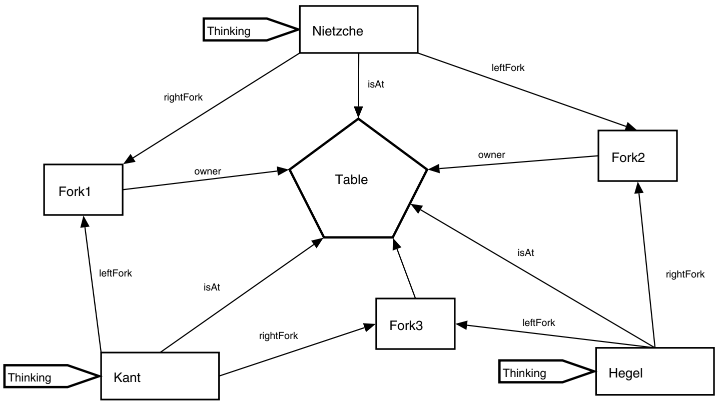

Figure 2. The initial graph for the Dining Philosophers problem.

sage typed as Thinking. However, since its parent class *Philosopher* has those arcs connected to it, they can be connected to any derived object, thus allowing inheritance of elements. To see that, consider the attribute *isAt* of the right-handed philosopher Kant. The edge it is mapped to by the typing morphism has as source an element of class *Philosopher*, and so $(t_V^* \circ src_G)(\text{isAt}) = \mathit{Right\text{-}HandedPhilosopher} \sqsubseteq_{V_\mathcal{C}^*} \mathit{Philosopher} = (src_\mathcal{C} \circ t_E)(\text{isAt})$, and the morphism is allowed.

Relationships between $\mathcal{C}$-typed graphs can be described by morphisms.

Definition 2.6 [$\mathcal{C}$-typed graph morphism] Let $G_1^\mathcal{C} = \langle G_1, t_1, \mathcal{C}\rangle$ and $G_2^\mathcal{C} = \langle G_2, t_2, \mathcal{C}\rangle$ be two $\mathcal{C}$-typed graphs typed over the same class-model graph $\mathcal{C} = \langle V_\sqsubseteq, E_\sqsubseteq, L, src, tar, lab\rangle$. A *$\mathcal{C}$-typed graph morphism* $h : G_1^\mathcal{C} \to G_2^\mathcal{C}$ between $G_1^\mathcal{C}$ and $G_2^\mathcal{C}$, is a pair of partial functions $h = \langle h_V : V_{G_1} \to V_{G_2}, h_E : E_{G_1} \to E_{G_2}\rangle$ such that the diagram below (in category **SetP**) commutes, for all elements $v \in dom(h_V)$, $(t_{2V} \circ h_V)(v) \sqsubseteq_{V_\mathcal{C}} t_{1V}(v)$, and for all elements $e \in dom(h_E)$, $(t_{2E} \circ h_E)(e) \sqsubseteq_{E_\mathcal{C}} t_{1E}(e)$. If $(t_{2E} \circ h_E)(e) = t_{1E}(e)$ for all elements $e \in dom(h_E)$, the morphism is said to be *strict*.

$$\begin{tikzcd}
E_{G_1} \arrow[d, "{src_{G_1}, tar_{G_1}}"'] & dom(h_E) \arrow[l, hook', "h_E?"'] \arrow[r, mapsto, "h_E!"] & E_{G_2} \arrow[d, "{src_{G_2}, tar_{G_2}}"] \\
V_{G_1}^* \arrow[rr, "h_V^*"] & & V_{G_2}^*
\end{tikzcd}$$

A graph morphism is a mapping which preserves hyperarcs sources and targets. A typed graph morphism also preserves (node and edge) types. Ordinary typed graph morphisms [3], however, cannot describe correctly morphisms on object-oriented systems because the existing inheritance relation among objects causes that actions available for objects of a certain kind are valid to *all* objects derived from it. So, an object can be viewed as not being uniquely typed, but having a type *set* (namely, the set of all types it is connected via the inheritance relation). Defining a graph morphism compatible with the underlying order relations assures that polymorphism can be applied consistently.

The behavior of object-oriented systems (implementation of methods) will be

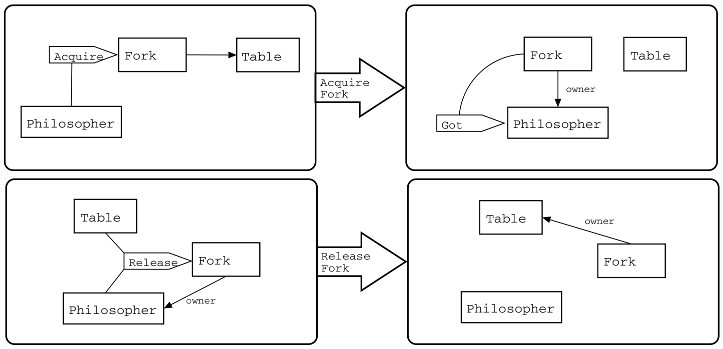

Figure 3. *Fork* rules for the Dining Philosophers problem.

modeled by rules, in which the left- and right-hand sides are object-oriented graphs. Besides structural restrictions (imposed by the fact that rules as $\mathcal{C}$-typed graph morphisms), some others are necessary to assure compatibility with the concepts of the object paradigm. Particularly, a rule left-hand side contains exactly one element of type message, and this particular message must be deleted by the rule application, i.e., each rule represents an object reaction to a message which is consumed in the process. This demand poses no unreasonable restriction, since systems may have many rules specifying reactions to the same type of message (nondeterminism) and many rules can be applied in parallel if their triggers are present at an actual state and the referred rules are not in conflict [6]. At most one object having attributes will be allowed on the left-hand side of a rule, along with the requirement that this same object must be the target of the above cited message. This restriction implements the principle of *information hiding*, which states that the internal configuration (implementation) of an object can only be visible, and therefore accessed, by itself. The rule morphism must be invertible, to assure that an object does not have its type changed along the computation. Finally, there must be a bijection between the edges on both sides, and so an object does not gain or loose attributes as the computation evolves.

Object-oriented graph grammars are composed by a class-model graph, an initial state (a complete object-oriented graph) and a set of object-oriented rules.

Example 2.7 An object-oriented graph grammar for the Dining Philosophers problem is presented in Figures 3, 4, and 5. All object-oriented rules left- and right-hand sides are object-oriented graphs typed over the class-model graph portrayed in Figure 1. However, in order to make the presentation clearer, all nodes and edges are named after their types, making the typing morphism explicit.

The semantics of an object-oriented graph grammar is based on rule applications. Matches and direct derivations are defined in the same way as the single-pushout approach: a match is a total $\mathcal{C}$-typed graph morphism, and a direct derivation is the pushout of the match and rule arrows in the category of object-oriented graphs and their morphisms [9]. Instead of using the usual transition system induced by the application of rules starting at the initial graph of the system (states are graphs and transitions are graph morphisms), we defined an abstract semantics based on

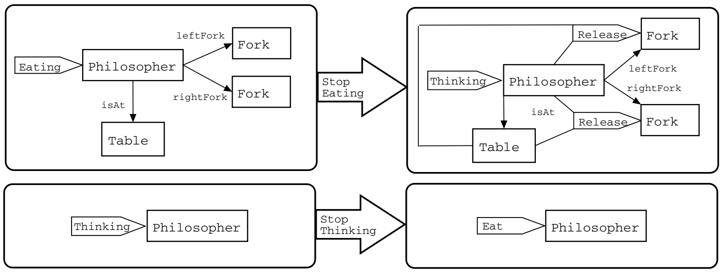

Figure 4. *Philosopher* rules for the Dining Philosophers problem.

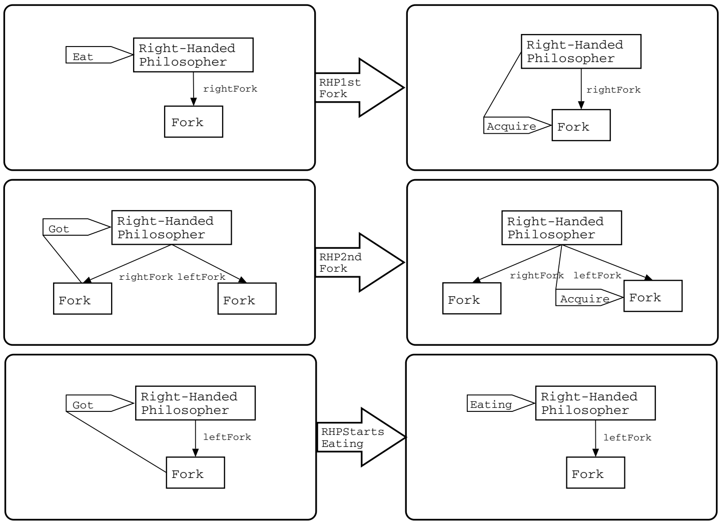

Figure 5. *Right-HandedPhilosopher* rules for the Dining Philosophers problem.

observations. This semantics holds information about events happening in a system (message exchange among objects), and forgets about system structure. Therefore, although we are not able to express properties based on object states, we are still allowed to investigate properties of objects based on how they respond to the rules applied to them. The abstract semantics is given by a labeled transition system where its states are the graphs generated by rule applications in the grammar, and the transition between two states is labeled with the name of the rule applied together with the object identity the rule was applied to.

Definition 2.8 [Object-oriented graph grammar transition semantics] Let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C}\rangle$ be an object-oriented graph grammar. The transition semantics of $\mathcal{G}$ is given by the labeled transition system $\mathcal{T}^{\mathcal{G}} = \langle S, s_0, L, \rightarrow\rangle$, where $S = \{G^{\mathcal{C}} \mid I^{\mathcal{C}} \Rightarrow^* G^{\mathcal{C}}\}$ is the set of states, $s_0 = I^{\mathcal{C}}$ is the initial state, $L = \{\langle p, o\rangle \in P^{\mathcal{C}} \times V_G \mid G \in S^{\mathcal{T}} \wedge p \in \Pi^E_{t_G(o)}\}$, is the set of labels, where $\Pi^E_{t_G(o)}$ is the grammar set of productions that can be applied to an object of type $t_G(o)$, $\rightarrow$ is the transition relation, and object-oriented graphs $G^{\mathcal{C}}$ and $H^{\mathcal{C}}$ are related under $\rightarrow$ if there is an object-oriented graph production $r : L^{\mathcal{C}} \rightarrow R^{\mathcal{C}} \in P^{\mathcal{C}}$, an object-oriented match $m : L^{\mathcal{C}} \rightarrow G^{\mathcal{C}}$ such that $G^{\mathcal{C}} \overset{r,m}{\Rightarrow} H^{\mathcal{C}}$.

3 Translation

The input language of SPIN [10] is Promela (PROtocol/PROcess MEta LAnguage) which is a specification language to model state transition systems. As formalized in object-oriented graph grammars, inheritance, polymorphism, and dynamic binding will be encoded in Promela, which originally does not have any object-oriented features. The complete translation algorithm [7] is rather long and will be presented here informally. Objects are modeled as Promela processes, and message exchange between objects through asynchronous communication channels. To overcome the FIFO policy of buffered channels in Promela, the same solution from [5] is used: a local buffer is used to "shuffle" received messages and so maintain the nondeterministic rule application semantics. The inheritance relation appears as a global array visible to any program element. Subclass polymorphism is coded through an inspection in this array, to assure that rule matches only occur if the matched elements are correctly related. Dynamic binding is implemented as a message dispatch procedure within each object process definition. Differently from classic object-oriented programming languages implementations [14], where a virtual table determines which method should be called in execution time, our approach uses a little computational reflection [17], in the sense that each object (process) is aware of its own type, and that information is made available to other entities when they have access to the object (as an attribute, or as a message parameter). So an object can decide, at run time, the adequate message to send based on the actual type of the message receiver.

Each initial graph node is transformed into a process, having as parameters all the targets of its attributes (using an arbitrary total order imposed on each object attributes). Each initial message is put into the proper object channel, together with its parameters (the sources of each message arc in the initial graph). Therefore, targets of attribute edges become processes parameters, and message parameters become processes local variables. A process code (the object behaviour) consists of an infinite loop that continuously tests (nondeterministically) if either a new message has arrived at the object main channel — in which case the message is retrieved and placed in some empty slot of the local message buffer — or if there is a message in the local buffer waiting to be consumed. In the latter case, the message is atomically retrieved from the buffer and the production it refers to is applied. In case neither the object channel nor the local message buffer contain any messages to be consumed, the process will jump to the beginning of the main loop, and stay blocked until a new message arrives.

The matching procedure tests if (i) all attributes are typed correctly, and (ii) all attribute values are correct. For instance, consider the first rule in Figure 3. This match will only be possible if the holder of the attribute vertex (of type *Fork*) is an object typed as *Table*. If it is a *Philosopher*, the match will not occur, because those two elements are not related by inheritance (although they both derive from a *ForkHolder*). Type testing is performed by inspection on the aforementioned global inheritance array. Now, consider the second rule from that same figure. The match is possible only if the *Philosopher* passed as parameter of message Release is the same one holding the fork. Therefore, an equality test is carried on between objects which are sources or targets of distinct arcs.

The choice on which production to apply is performed by a conditional test for all rules to which a match (for the received message) exist. Since a conditional test in Promela has a nondeterministic result if more than one conditional is true, the choice of which production to apply is also nondeterministic, as required by the grammar semantics.

Rule application can be described as: (i) object attributes are modified according to the rule morphism; (ii) a global variable `event_RuleName` is set with the applied production name; (iii) the set of variables `event_`x, for all classes to which the type of the production attribute vertex is related by inheritance, are set to the object identity; (iv) finally, all messages appearing in the right-hand side of the applied production are created, and it is particularly relevant, since it is this procedure which performs dynamic binding. Steps (i) to (iii) are performed without generating intermediate states, to assure correctness of property verification. If no rule is applied (because no match were possible for any production implementing the received message), then the message is put back in the local buffer, and marked as inspected. An already inspected message will not be retrieved for application until a new message arrives. Since only an object can change its own state, a match for this message could only happen after another production is actually applied. This procedure also helps to decrease the program state space for the verification process.

The right message to send is based on the type of the actual object which is receiving the message. Since the lower set of any node (respecting the inheritance hierarchy) is finite and does not change along the program execution, a conditional structure takes care of it. For instance, consider the second rule in Figure 4. A message `Eat` is sent to a *Philosopher*. This message, however, is redefined by all *Philosopher* subclasses, so one must know the element type to send the correct message. The code generated is illustrated by the following pseudo-code:

```
if (receiving object message channel is not full)
  if
    . receiving object type is a Philosopher ->
         send message Eat for the Philosopher
    . receiving object type is a Left-HandedPhilosopher ->
         send message Eat for the Left-HandedPhilosopher
    . receiving object type is a Right-HandedPhilosopher ->
         send message Eat for the Right-HandedPhilosopher
```

The whole rule application procedure is performed atomically. Therefore, from the time a message is taken out of the local buffer to the time a rule application is completed — by either applying the rule or by putting the message back to the local buffer, if no match exist for that rule — no other process can interleave with that execution, because of the `atomic` keyword. The atomicity of the rule application process is necessary, to mimic the way rules are applied in the graph grammar, where the whole matching and application procedure is performed in a single step. Furthermore, if interleaving was allowed, errors could appear: if a process is stopped between finding a match for a production and the application of that production, meanwhile the state graph could be altered in a way that turns the rule application impossible; therefore a match/application procedure is considered a critical region of any object behaviour.

4 Verification

Property verification in SPIN can be done using a multiplicity of methods, among which there is LTL [11] property verification. Meaningful events to verification of the Dining Philosophers problem can be stated, for instance, as "philosopher X starts to eat", or "fork Y is grabbed by a philosopher". We will use the already presented object-oriented graph grammar for the Dining Philosophers problem as the running example. For reasons of space, we will verify only the liveness property stated as "anytime a philosopher decides to eat, he eventually does so". We will show that this property is false in the provided model.

SPIN performs model-based verification, which means that properties can only be defined over states, and not over transitions. The translation we propose defines a set of global variables to allow verification over events: (i) one global variable for each class belonging to the class-model graph over which the grammar is typed, to identify the last object of that type that had a production applied to it (if a message is received by an object, and consumed by some rule application, then the object identity is assigned to the respective variable), and (ii) one global variable to identify which *rule* was applied, and it is updated every time such action occurs. Notice that rule application is not equivalent to message consumption. Although each rule application corresponds exactly to a response to a received message, there can be multiple (different) rules implementing actions for the same type of message. This variable is necessary if one is interested in verify possible orders in which rules can be applied.

The XSpin tool allows that propositions can be defined in a C-like way, using the preprocessor macro `#define`. Those properties can be defined in terms of the actual objects belonging to the system initial graph. Since we are only interested in the behaviour of the philosophers, a proposition to identify each of them is defined as in *#define isKant (event_Philosopher == Kant)*. Propositions for events of interest can be defined using the global variable for rule identification, as in *#define aPhilWantsToEat (event_RuleName == rule_Philosopher_StopThinking)* or in *#define aPhilStartsToEat (event_RuleName == rule_Philosopher_StartsEating)*. In order to discover if a known event occurs with a specific object, propositions such as *#define philKantWantsToEat (isKant && aPhilWantsToEat)* and *#define philKantStartsToEat (isKant && aPhilStartsToEat)* can be defined.

Using the propositions defined above, LTL properties about the system behaviour can be written. Property "anytime a philosopher decides to eat, he eventually does so" can be stated, for philosopher Kant as *[] (philKantWantsToEat − > <> philKantStartsToEat)* where symbols `<>` and `[]` stand for the usual linear temporal logic quantifiers $\Diamond$ (eventually) and $\Box$ (always). For all philosophers, it can be stated as *[] ((philKantWantsToEat − > <> philKantStartsToEat) && (philHegel-WantsToEat − > <> philHegelStartsToEat) && (philNietzscheWantsToEat − >

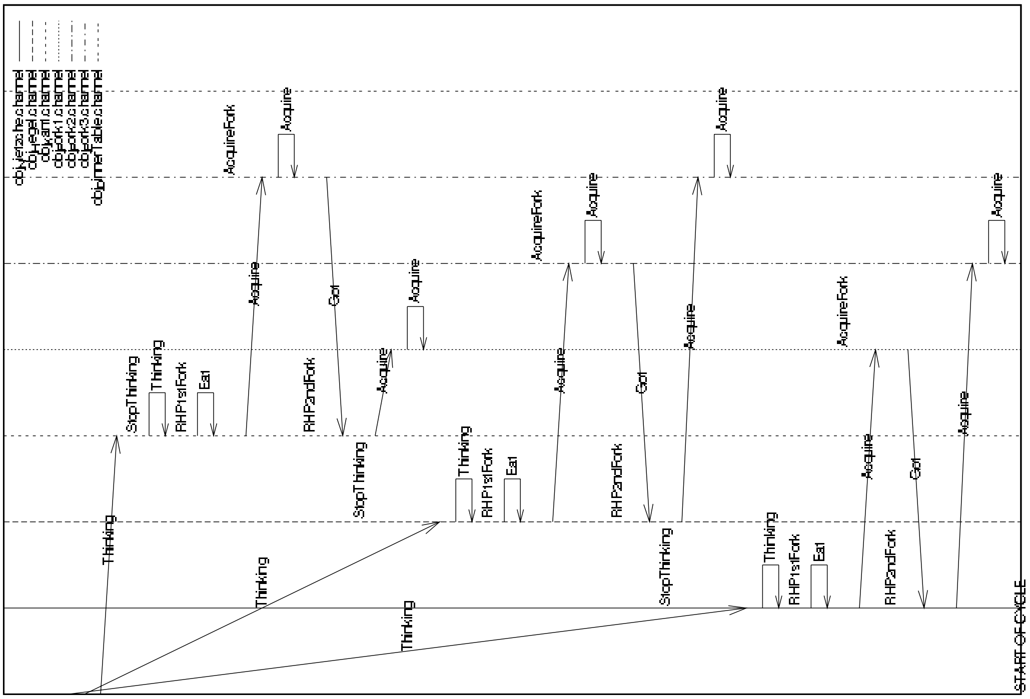

Figure 6. Counterexample of the absence of deadlock property

<> *philNietzscheStartsToEat)).*

This last property is not true within the model provided. Figure 6 shows a graphical counterexample (taken from the model checker output, and generated by the system developed in [4]) of it. The counterexample shows three philosophers (Nietzsche, Hegel, and Kant) and their respective forks. The processes are indicated by the horizontal lines, and the arrows indicate the messages arriving and departing from each process. Notice that a deadlock situation is set: each philosopher have grabbed one fork, and a message was sent to the other fork in an attempt to acquire it. However, since each fork now has a philosopher owning it, rule `AcquireFork` cannot ever be applied again, and all philosophers will wait forever.

5 Conclusions

Object-oriented graph grammars provide a graph-based specification framework for object-oriented systems, where special partial orders represent the inheritance and overriding hierarchies, making polymorphism and dynamic binding built-in features of the formalism.

We have presented a (sketch) translation from object-oriented graph grammars specifications into Promela programs. All object-oriented features are translated into Promela: inheritance appears as a global array; polymorphism is implemented in the matching procedure through an inspection on this array; dynamic binding is implemented through the message dispatching mechanism, which checks the message receiver type to determine the correct message to send; information hiding and encapsulation appear naturally on the translation, since a single process implements each system object. The translation of graph rules applications establish the existence of matches before the rule can be applied, and the choice of which message to consume and which production to apply is nondeterministic, as required by the defined grammar semantics.

We are not currently dealing with object creation and deletion, but it is a straightforward extension to this translation, which is currently being automatized (using and extension of the XML-based languages GXL and GTXL [18]). An effort can be done to customize the translation to the application characteristics, in order to reduce the produced state space.

The translation proposed is arguably semantically sound, in the sense that no graph system behaviour is removed or introduced by the translation. Even if there are states in the Promela program that do not correspond to any graph belonging to the grammar language, those states can always be translated if they are not part of a rule application procedure. If they are, they cannot interleave with any other process execution, since rule application is performed atomically, and hence it suffices to leave the atomic block for the Promela state can be translated to a graph state. A formal proof of this translation soundness is being prepared for publication. Finally, we have used a modeling for the Dining Philosophers problem to illustrate how verification can be performed, and how errors can be found using our approach.

References

[1] Baldan, P., A. Corradini and B. König, *Verifying finite-state graph grammars: An unfolding-based approach*, in: *CONCUR*, 2004, pp. 83–98.

[2] Burkart, O. and Y.-M. Quemener, *Model-checking of infinite graphs defined by graph grammars*, Technical Report 995, IRISA — Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (1996).

[3] Corradini, A., U. Montanari and F. Rossi, *Graph processes*, Fundamentae Informatica **26** (1996), pp. 241–265.

[4] dos Santos, O. M., "Verificação Formal de Sistemas Distribuídos Modelados na Gramática de Grafos Baseada em Objetos," Masters thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre (2004), 89p.

[5] Dotti, F. L., L. Foss, L. Ribeiro and O. M. Santos, *Verification of object-based distributed systems*, in: E. Najm, U. Nestmann and P. Stevens, editors, *Proceedings of the 6th IFIP TC6/WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, Lecture Notes in Computer Science **2884** (2003), pp. 261–275.

[6] Ehrig, H., R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, *Algebraic approaches to graph transformation. Part II: single-pushout approach and comparison with double pushout approach*, , **1 (Foundations)**, World Scientific, Singapore, 1996 pp. 247–312.

[7] Ferreira, A. P. L., "Object-oriented graph grammars," PhD thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil (2005), 156p.

[8] Ferreira, A. P. L. and L. Ribeiro, *Towards object-oriented graphs and grammars*, in: E. Najm, U. Nestmann and P. Stevens, editors, *Proceedings of the 6th IFIP TC6/WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, Lecture Notes in Computer Science **2884** (2003), pp. 16–31.

[9] Ferreira, A. P. L. and L. Ribeiro, *Derivations in object-oriented graph grammars*, in: H. Ehrig, G. Engels, F. Parisi-Presicce and G. Rozenberg, editors, *Proceedings of the 2nd International Conference on Graph Transformations (ICGT 2004)*, Lecture Notes in Computer Science **3256** (2004), pp. 416–430.

[10] Holzmann, G. J., *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), pp. 1–17.

[11] Huth, M. R. A. and M. D. Ryan, "Logic in Computer Science: Modelling and reasoning about systems," Cambridge University Press, Cambridge, 2000.

[12] Koch, M., "Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems," PhD Thesis, Technische Universität Berlin, Berlin (1999).

[13] Löwe, M., "Extended Algebraic Graph Transformation," Ph.D. thesis, Technischen Universität Berlin, Berlin (1991).

[14] Pratt, T. W. and M. V. Zelkowitz, "Programming languages : design and implementation," Prentice-Hall, Upper Saddle River, 1996, 3 edition, 654p.

[15] Rensink, A., *Towards model checking graph grammars*, in: M. Leuschel, S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)* (2003), pp. 150–160.

[16] Rensink, A., *The GROOVE simulator: A tool for state space generation*, in: J. Pfalz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Lecture Notes in Computer Science **3062** (2004), pp. 479–485.

[17] Smith, B. C., "Reflection and Semantics in a Procedural Language," PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA (1982), mIT-LCS-TR-272.

[18] Winter, A., B. Kullbach and V. Riediger, *An overview of the GXL graph exchange language*, in: S. Diehl, editor, *International Seminar on Software Visualization*, Lecture Notes in Computer Science **2269** (2001), pp. 324–336.