

# Formal Specification and Verification of Real-Time Systems using Graph Grammars

Leonardo Michelon<sup>1</sup>, Simone André da Costa<sup>1,2</sup> and Leila Ribeiro<sup>1</sup>

<sup>1</sup>Instituto de Informática  
Universidade Federal do Rio Grande do Sul  
Av. Bento Gonçalves, 9500  
91501-970 - Porto Alegre - RS - BRAZIL  
{lmichelon, scosta, leila}@inf.ufrgs.br

<sup>2</sup>Departamento de Informática  
Universidade Federal de Pelotas  
Campus Universitario, s/n  
96010-900 - Pelotas - RS - BRAZIL  
{simone.costa}@ufpel.edu.br

## Abstract

*The importance of real-time systems has enormously increased in the last decade. Application areas that typically need real-time models include railroad systems, intelligent vehicle highway systems, avionics, multimedia and telephony. To assure that such systems are correct, additionally to prove that they provide the required functionality, time constraints must be satisfied. There are already formal specification methods for real-time systems, but most of them are difficult to use by software developers, that are usually not very familiar with mathematical notation but rather specify systems using the object-oriented paradigm. In this paper we propose a formal approach to specify and analyze real-time systems that has an object-oriented flavor. This approach is based on Object-Based Graph Grammars (OBGGs), a formal description technique suitable for the specification of asynchronous distributed systems, and intuitive even for non-theoreticians. We extend OBGGs to enable explicit modeling of time constraints, and define the semantics of the specifications via transition systems. Finally, we translate timed OBGGs to Timed Automata, a formal notation that is wide spread in the area of real-time systems modeling and allows the automatic verification of properties.*

**Keywords:** Real-time computing, Formal specification and verification, Graph grammars, Timed automata.

## 1. INTRODUCTION

One of the goals of software engineering is to aid the development of correct and reliable software systems. Formal specification methods play an important role in accomplishing this goal [7]. Besides providing means to prove that a system satisfies the required properties, formal methods contribute to its understanding, revealing ambiguities, inconsistencies and incompleteness that could hardly be detected otherwise.

The use of a formal specification method is even more important in the design of real-time systems, frequently used in critical security environments. A real-time system is a system in which performance depends not only on the correctness of single actions, but also on the time at which actions are executed [27, 26]. Application areas that typically need real-time models include railroad systems, intelligent vehicle highway systems, avionics, multimedia and telephony. To assure that such systems are correct, additionally to prove that they provide the required functionality, we have to prove that the time constraints are satisfied.

### 1.1. FORMAL SPECIFICATION OF REAL-TIME SYSTEMS

There are many formal approaches to model real-time systems. Timed Automaton [1, 4] is one of the most

prominent methods for real-time specification. A timed automaton extends a usual automaton by adding several clocks to states and time restrictions to transitions (and states). It enables us to specify both the discrete behavior of control and the continuous behavior of time. Process calculi models including time [21, 2, 6] have also been proposed, adding a set of timing operators to process algebras. They offer a level of abstraction based on processes: a system is viewed as a composition of (interacting) processes. Timed Petri nets [28] and time Petri nets [5] are extensions of the classical Petri nets adding time values to transitions/tokens or time intervals to transitions, respectively.

Although the models discussed above may be adequate for some aspects of a system, they stress the representation of the control structure, lacking a comprehensive representation of data structure and its distribution within a system. Object-oriented models provide such abstraction by joining descriptions of data and processes within one object. Distribution and concurrency appear naturally by viewing objects as autonomous entities. Object-oriented approaches are widely accepted for specification and programming. Thus, various Unified Modeling Language (UML)-based approaches have already been proposed to model time information. HUGO/RT [18] is an automated tool that checks if a UML state machine interacts according to the scenarios specified by a sequence diagram (extended with time constraints). For this verification, state machines are compiled into Timed Automata and sequence diagrams into Observer Timed Automaton. After the translations, the model-checker Uppaal [3] is called to check if the Observer Timed Automaton describes a reachable behavior of the system. Diethers and Hunh [9] propose a similar approach through the Voodoo tool. Nevertheless, these tools restrict the specification of a real-time system. First of all, because they are based on UML state machines and therefore do not offer clocks or priorities, useful concepts for real-time modeling. Besides, even though the proposals intent to be faithful to the UML informal specification, following its semantic requirements, the translation of state machines is not based on a formal semantic. In [20] a translation of timed state machines into a real-time specification language TRIO was proposed, but TRIO is not directly model checkable.

The approach in [19] adds time information to UML classes. Attributes of type *Timer*, for the definition of clocks for classes, and a notation similar to timed automata, to analyze and evaluate clocks in UML state diagrams, are syntactically introduced. A translation from UML into Promela, the input language of the SPIN model-checker, is extended to give semantics to the diagrams. The main difficulty in using this approach is that, since Promela does not have built-in time constructors,

clocks and time constrains have to be encoded and, since the semantics is defined by the Promela code, it might be quite difficult to understand for users not very familiar with this language.

The Omega project also aims to model and verify real-time systems. An extension of a UML subset with time constructs was proposed in [16], and in [22] part of UML was mapped into Communication Extended Timed Automata (input language for the validation tool). Static properties are described as Observer Automata and dynamic properties by UML Observers. This is a very interesting approach, although the user has to learn a variety of different languages and diagrams to completely specify and verify a system.

## 1.2. OUR CONTRIBUTION

In this paper, we propose extending the formal description technique Object-Based Graph Grammars (OBGGs) [11, 23] to specify real-time systems. OBGG is a visual formal specification language suitable for the specification of asynchronous distributed systems. The basic idea of this formalism 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 described via applications of these rules to graphs modeling 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. OBGGs are appealing as specification formalism because they are formal, they are based on simple but powerful concepts to describe behavior, and they have a nice graphical layout that helps non-theoreticians understand an object-based graph grammar specification. Due to the declarative style (using rules), concurrency arises naturally in a specification: if rules do not conflict (do not try to update the same portion of the state), they may be applied in parallel (the specifier does not have to say explicitly which rules shall occur concurrently).

OBGGs can be analyzed through simulation [12] and verification (using the SPIN model-checker) [10]. Compositional verification (using an assume-guarantee approach) is also provided [24]. Moreover, there is an extension of OBGGs to model inheritance and polymorphism [15]. However, OBGGs do not provide explicit time constructs, and therefore are not suited to model and analyze real-time systems. Here we propose a mapping from a timed extension of OBGG specifications to Timed Automata. This way, we can use the available (Timed Automata) verification tools to check properties of timed OBGGs.

Our approach adds time stamps to the messages (allowing to program certain events to happen in the future), extends the appealing formal description technique OBGG, and supports verification of properties written in

temporal logic. The main contributions of this paper are: (i) the proposal of a timed extension to OBGG; and (ii) the translation of the extended OBGG to Timed Automata, leading to automatic verification of properties. The paper is organized as follows: Section 2 presents OBGGs and its timed extension; Section 3 describes the semantics of timed OBGGs; Section 4 reviews Timed Automata; the translation of timed OBGGs to Timed Automata is described in Section 5; Section 6 analyzes the example; and final remarks are in Section 7.

## 2. OBJECT BASED GRAPH GRAMMARS

Object based graph grammar (OBGG) is a formal visual language suited to the specification of object-based systems. We consider object-based systems with the following characteristics: (i) a system is composed of various objects. The state of each object is defined by its attributes, which may be pre-defined values or references to other objects. An object cannot read or modify the attributes of other objects; (ii) objects are instances of classes. Each class includes the specification of its attributes and of its behavior; (iii) objects are autonomous entities that communicate asynchronously via message passing. An OBGG specifies a system in terms of states and changes of states, where states are described by graphs and changes of states are described by rules.

Now, the definitions used for the description of Timed Object-Based Graph Grammars (TOBGGs) are presented. Each formal definition is preceded by an informal description of its meaning. The formal definitions are necessary to follow the translation of TOBGG to Timed Automata, described in Section 5. For the comprehension of the other contributions of this paper, it is possible to skip the formal definitions. Examples of main concepts can be found in Subsection 2.2.

**Graph, Graph Morphism.** A *graph* consists of a set of vertices partitioned into two subsets, of objects and values (of abstract data types), and a set of edges partitioned into sets of message and attribute edges. Values are allowed as object attributes and/or message parameters. Messages, modeled as (hyper)edges, may also have other objects as parameters and must have a single target object. These connections are expressed by total functions assigning to each edge its source and target vertices. Figure 1 illustrates a graph for an object-based system. Values of abstract data type Natural are allowed, for example, as attributes of the Train object and as parameters of the OpenGate message. The Train object also has references to Gate and RSegm objects as attributes. Message OpenGate has the Train object as target and as sources a reference to RSegm object and

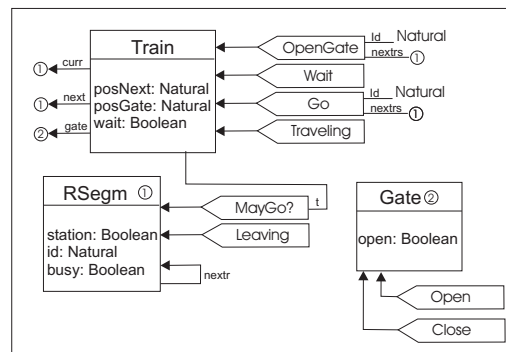


Figure 1. Type Graph

Natural values. A *graph morphism* expresses a structural compatibility between graphs: if an edge is mapped, the corresponding vertices, if mapped, must have the same sources/target vertex; if a vertex is mapped, the attributes must be the same.

Let  $f : A \rightarrow B$  be a partial function and let  $f^\bullet$  be the corresponding total function  $f^\bullet : \text{dom}(f) \rightarrow B$ , s.t.,  $f(x) = f^\bullet(x)$ ,  $\forall x \in \text{dom}(f)$ . Let  $\text{Spec}$  be an algebraic specification, including sort  $\text{Nat}$  and the usual operations and equations for natural numbers, and  $\mathcal{U} : \text{Alg}(\text{Spec}) \rightarrow \text{Set}$  be the forgetful functor that assigns to each algebra the disjoint union of its carrier sets. It is assumed that the reader is familiar with basic notions of algebraic specification (see, e.g., [14]).

**DEFINITION 1 (GRAPH, GRAPH MORPHISM)** *Given an algebraic specification  $\text{Spec}$ , a **graph**  $G = (V_G, E_G, s^G, t^G, A_G, a^G)$  consists of a set  $V_G$  of vertices partitioned into sets  $oV_G$  and  $vV_G$  (of objects and values, respectively), a set  $E_G$  of (hyper)edges partitioned into sets  $mE_G$  and  $aE_G$  (of messages and attributes, respectively), a total source function  $s^G : E_G \rightarrow V_G^*$ , assigning a list of vertices to each edge, a total target function  $t^G : E_G \rightarrow oV_G$  assigning an object-vertex to each edge, an attribution function  $a^G : vV_G \rightarrow \mathcal{U}(A_G)$ , assigning to each value-vertex a value from a carrier set of  $A_G$ .*

A (*partial*) **graph morphism**  $g : G \rightarrow H$  is a tuple  $(g_V, g_E, g_A)$ , where the first components are partial functions  $g_V = g_{oV} \cup g_{vV}$  with  $g_{oV} : oV_G \rightarrow oV_H$  and  $g_{vV} : vV_G \rightarrow vV_H$  and  $g_E = g_{mE} \cup g_{aE}$  with  $g_{mE} : mE_G \rightarrow mE_H$  and  $g_{aE} : aE_G \rightarrow aE_H$ ; and the third component is a total algebra homomorphism such that the diagrams below commute. A morphism is called *total* if all components are total. This category of graphs and partial graph morphisms is denoted here by **GraphP** (identities and composition are defined componentwise).

$$\begin{array}{ccc}
 \text{dom}(E_G) \xrightarrow{g_E^\bullet} E_H & & \text{dom}(E_G) \xrightarrow{g_E^\bullet} E_H \\
 \downarrow s^G & = & \downarrow s^H \\
 V_G^* \xrightarrow{g_V^*} V_H^* & & oV_G \xrightarrow{g_V^*} oV_H \\
 \\
 \text{dom}(vV_G) \xrightarrow{g_V^\bullet} vV_H & & \\
 \downarrow a^G & = & \downarrow a^H \\
 \mathcal{U}(A_G) \xrightarrow{\mathcal{U}(g_A)} \mathcal{U}(A_H) & & 
 \end{array}$$

**OB-Graphs.** An *OB-graph* is a graph equipped with a morphism *type* to a fixed graph of types [8]. Since types constitute the static part of the definition of a class, we call the graph of types as *class graph*. Two restrictions are imposed to a *class graph* to guarantee that it corresponds to a class in the sense of the object paradigm: the first is that there are no data values in a class graph (they are represented by the name of data types); and the second imposes that each class can have exactly one list of attributes. A morphism between OB-graphs is a partial graph morphism that preserves the typing.

**DEFINITION 2 (OB-GRAPHS)** *Let Spec be a specification. A graph C is called a **class graph** iff (i)  $A_C$  is a final algebra<sup>1</sup> over Spec, (ii) for each object vertex  $v \in oV_C$  there is exactly one attribute hyperedge ( $ae \in aE_C$ ) with target  $v$ . An **OB-graph over C** is a pair  $OG^C = (OG, \text{type}^{OG})$  where  $OG$  is an graph called **instance graph** and  $\text{type}^{OG} : OG \rightarrow C$  is a total graph morphism, called the **typing morphism**. A morphism between OB-graphs  $OG_1^C$  and  $OG_2^C$  is a partial graph morphism  $f : OG_1 \rightarrow OG_2$  such that for all  $x \in \text{dom}(f)$ ,  $\text{type}^{OG_1}(x) = \text{type}^{OG_2} \circ f(x)$ . The category of OB-graphs typed over a class graph C, denoted by **OBGraph(C)**, has OB-graphs over C as objects and morphisms between OB-graphs as arrows (identities and composition are the identities and composition of partial OB-graph morphisms).*

The operational behavior of the system described by a graph grammar is determined by the application of grammar rules to the graphs that represent the states of the system (starting from an initial state).

**Rule.** A rule of an object-based grammar consists of (the numbers in parenthesis at the end of each item correspond to conditions in Definition 3):

- a *finite left-hand side L*: describes the items that must be present in a state to enable the application of the rule. The restrictions imposed to left-hand sides of rules are:

- There must be exactly one message vertex, called trigger message – this is the message handled/deleted by this rule (cond. 2).
- Only attributes of the target object of the trigger message should appear – not all the attributes of this object should appear, only those necessary for the treatment of this message (cond. 3).
- Values of abstract data types may be variables that are instantiated at the time of the application of the rule. Operations defined in the abstract data type specification may be used (cond. 6 and 7).

- a *finite right-hand side R*: describes the items that will be present after the application of the rule. It may consist of:

- 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 (cond. 4 and 5).
- Messages to all objects appearing in R.

- a condition: this condition must be satisfied for the rule to be applied. This condition is an equation over the attributes of left- and right-hand sides.

**DEFINITION 3 (RULE)** *Let C be a class graph, Spec be a specification and X be a set of variables for Spec. A **rule** is a pair  $(r, Eq)$  where Eq is a set of equations over Spec with variables in X and  $r = (r_V, r_E, r_A) : L \rightarrow R$  is a C-typed OB-graph morphism s.t.*

1. *L and R are finite;*
2. *a message is deleted:  $\exists! e \in mE_L$ , called trigger(r),  $\text{trigger}(r) \notin \text{dom}(r_E)$ ;*
3. *only attributes of the target of the message may appear in L:  $(aE_L = \emptyset) \vee ((\exists! e \in aE_L) \wedge t^L(e) = t^L(\text{trigger}(r)))$ ;*
4. *attributes of existing objects may not be deleted nor created:  $\forall o \in oV_L. (\exists e \in aE_L. t^L(e) = o \Rightarrow \exists e' \in aE_R. t^L(e') = r_V(o))$ ;*
5. *objects may not be deleted:  $\forall o \in oV_L. o \in \text{dom}(r_V)$ ;*
6. *the algebra of r is a quotient term algebra over the specification  $\text{Spec}^{\text{Nat}}$  including a set of equations Eq and variables in X;*
7. *attributes appearing in L may only be variables of X:  $\forall v \in vV_L. a^L(v) \in X$ ;*

<sup>1</sup>An algebra in which each carrier set is a singleton.

8. the algebra homomorphism component of  $r$  ( $r_A$ ) is the identity (rules may not change data types).

We denote by  $Rules(C)$  the set of all rules over a class graph  $C$ .

**Object-Based Graph Grammar (OBGG).** An object-based system is composed of:

- a *type graph*: a graph containing information about all the attributes of all types of objects involved in the system and messages sent/received by each kind of object.
- a *set of rules*: these rules specify how the objects will behave when receiving messages. For the same kind of message, we may have many rules. 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. The behavior of an object when receiving a message is specified as an atomic change of the values of the object attributes together with the creation of new messages to any objects.
- an *initial graph*: this graph specifies the initial values of the attributes of the 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 of the execution of the object.

**DEFINITION 4 (OBJECT-BASED GRAPH GRAMMAR)**







An *object-based graph grammar*, short *OBGG*, is a tuple  $(Spec, X, C, IG, N, n)$  where  $Spec$  is an algebraic specification,  $X$  is a set of variables,  $C$  is a class graph,  $IG$  is a  $C$ -typed graph, called *initial or start graph*,  $N$  is a set of rule names,  $n : N \rightarrow Rules(C)$  assigns a rule to each rule name.

**2.1. TIMED OBGG**

Originally, the Graph Grammar formalism does not include the concept of time. Here we incorporate time to the model in order to model real-time systems. There are several choices of where to put the time in a graph grammar: rules, messages, and objects. We have decided to put time stamps on the messages describing when they are to be delivered/handled. In this way, we can program certain events to happen at some specific time in the future. Rules do not have time stamps, that is, the application of a rule is instantaneous. Our idea is that the time assigned to messages models the amount of time these messages need to arrive at their destinations. This choice was made because we intend to have a formalism suitable for reactive systems, that are typically distributed systems with asynchronous communication. In such systems, the

most time consuming operation is communication (that is, transmission times are usually much bigger than computation times), and therefore it is adequate to assume that computations (rule applications) are instantaneous, and that communication (message exchange) consumes time. Moreover, we adopt relative time: time stamps are not to be understood as absolute time specifications of when an event should occur, but rather as an interval of time relative to the current time in which the event should occur.

Syntactically a Timed Object Based Graph Grammar (TOBGG) is an OBGG with an additional time representation at the messages. The time stamps of the messages have the format:  $\langle tmin, tmax \rangle$ , with  $tmin \leq tmax$ , where  $tmin$  and  $tmax$  are the minimum/maximum number of time units, relative to the current time, within which the message should be handled. The possible time-stamps are:

- : this message must be handled in at least  $tmin$  time units and at most in  $tmax$  time units, i.e., in interval  $[tmin + current\ time, tmax + current\ time]$ .
- : if  $tmin$  is omitted, the current time is assumed as the minimum time for this message, i.e., the message must be handled in interval  $[0 + current\ time, tmax + current\ time]$ .
- : if  $tmax$  is omitted, infinite is assumed (i.e., this message has no time limit to be delivered). It must be handled in interval  $[tmin + current\ time, +\infty)$ .
- : if  $tmax = tmin$ , this message must be handled in a specific time during the simulation, i.e., in interval  $[t + current\ time, t + current\ time]$ .
- : if  $tmin$ ,  $tmax$  and the bar  $|$  are omitted, the message can be delivered from the current time on, and has no time limit to be handled, i.e., it must be handled in interval  $[0 + current\ time, +\infty)$ .
- : this notation is equivalent to having  $tmin = tmax = t$ , with  $t$  being the current time. These messages must be handled immediately, i.e., in interval  $[current\ time, current\ time]$ .

Now, we define timed OBGs, short TOBGs using (typed and attributed) hypergraphs. Let  $Spec^{Nat}$  denotes an algebraic specification including sort  $Nat$  and the usual operations and equations for natural numbers.

**Timed Graph, Timed Graph Morphism.** A *timed graph* consists of a graph with two partial functions, which assign a minimum/maximum time to each message. A *timed message* has the minimum/maximum time defined. A *(partial) timed graph morphism* is a graph morphism that is compatible with time, that is, if a timed message is mapped, the time of the target message must be the same.

**DEFINITION 5 (TIMED GRAPH, TIMED GRAPH MORPHISM)**  
Let  $Spec^{Nat}$  be a specification, a **timed graph**  $TimG = (G, t_{min}^G, t_{max}^G)$  consists of a graph  $G$  and partial functions  $t_{min}^G, t_{max}^G : mE_G \rightarrow \mathbb{N}$ , assigning a minimum/maximum time to each message edge of  $G$ . A **timed message** is a message  $m \in mE_G$  such that  $t_{min}^G(m)$  and  $t_{max}^G(m)$  are defined. For timed messages  $m$ , we require that  $t_{min}^G(m) \leq t_{max}^G(m)$ .

A **(partial) timed graph morphism**  $g : G \rightarrow H$  is a graph morphism  $g = (g_V, g_E, g_A)$ , such that, the diagrams below commute. A morphism is called **total** if both components are total. The category of timed graphs and partial timed graph morphisms is denoted by **TimGraphP** (identities and composition are defined componentwise).

$$\begin{array}{ccc} \text{dom}(g_E) \cap \text{dom}(t_{max}^G) & \xrightarrow{g_E} & mE_H \\ \downarrow t_{max}^G & = & \downarrow t_{max}^H \\ \mathbb{N} & \xrightarrow{id} & \mathbb{N} \end{array}$$

$$\begin{array}{ccc} \text{dom}(g_E) \cap \text{dom}(t_{min}^G) & \xrightarrow{g_E} & mE_H \\ \downarrow t_{min}^G & = & \downarrow t_{min}^H \\ \mathbb{N} & \xrightarrow{id} & \mathbb{N} \end{array}$$

**Timed OB-Graph.** The definition of timed OB-graph is analogous to the untimed case.

**Timed Rule.** A *timed rule* is a rule with an additional requirement: the message in the left-hand side should not have a specific time stamp, i.e.,  $t_{min}$  and  $t_{max}$  are undefined. This means that the aim of the rules shall be to specify how to handle a message, and not when it should be delivered.

**DEFINITION 6 (TIMED RULE)** Let  $timed_{rule}(r : L \rightarrow R, Eq)$  be a rule according to Definition 3. Then  $timed_{rule}$  is a **(timed) rule** if the following condition is satisfied:

1.  $t_{min}^L(trigger(r))$  and  $t_{max}^L(trigger(r))$  are undefined.

**Timed Object-Based Graph Grammar (TOBGG).** The definition of TOBGG is analogous to Definition 4, considering timed graphs and timed rules.

## 2.2. EXAMPLE: RAILROAD SYSTEM

In this section we model a simple railroad system using graph grammars. This example is similar to the one presented in [17] (and in many other papers that propose new specification and verification techniques for real-time systems). In a railroad system, the most important issue that should be guaranteed is that trains do not crash. This is typically achieved by interlocking systems, that only allow trains to enter regions of tracks in which there are no other trains. We kept the example simple to be able to explain in detail how the semantical model - that describes the computations of the system - is constructed. Nevertheless the specification method as well as relevant properties of railroad controlling system can be suitably illustrated in this example.

The railroad system is composed of instances of three entities: Train, Gate and RSegm. We model a system in which there are trains traveling along a railroad (composed of railroad segments), and at some places there are gates. The model shall assure that two trains can not be at the same railroad segment at the same time, and additionally if there is a gate to enter some region of the railroad, this gate shall be opened when a train passes. This apparent simple behavior involves intense exchange of messages since this system is inherently asynchronous. Now we show the model for each component of this system.

**Train Entity:** the graph grammar of the Train Entity is depicted in Figure 1 (type graph), Figure 2 (initial graph template TrainIni) and Figure 3 (rules). The type graph shows that each train has six attributes: its current position (**curr**), a reference to the position the train can move to (**next**), a reference to a gate (**gate**), the identifier of the next position (**posNext**), the identifier of the gate (**posGate**), and a state attribute that describes whether the train is moving or waiting (**wait**).

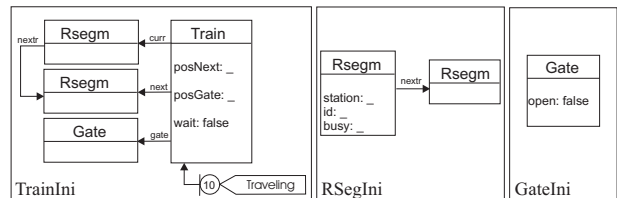


Figure 2. Initial Graphs Templates

Trains may receive four kinds of messages: **Wait**, **Go**, **OpenGate** and **Traveling**. Messages **OpenGate** and

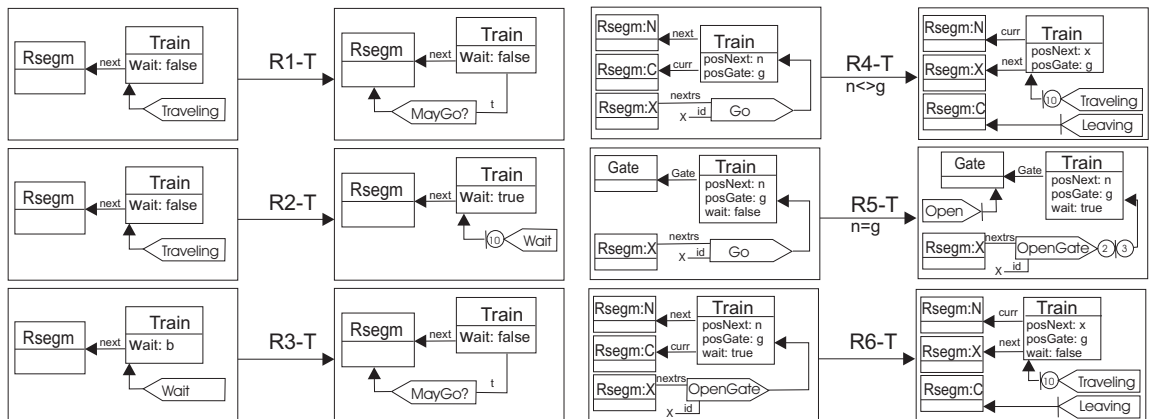


Figure 3. Train Rules

Go have parameters. The initial graph template TrainIni in Figure 2 describes that initially a train must be connected to two instances of RSegm, one of Gate, and has a pending Traveling message (this message will trigger the movements of the train). The attribute wait is initialized to false. To create an object of this class, four arguments are needed to instantiate this template: a natural number (posGate), two railroad segments (curr and next) and one gate (gate).

The behavior of the Train entity is modeled by the rules shown in Figure 3. Rule R1-T describes that, when receiving a message Traveling, a train tries to travel to the railroad segment pointed by its next attribute. This is modeled by the message MayGo? at the right-hand side of the rule asking permission to enter this segment. Rule R2-T chooses to wait at least 10 time units before trying to travel. Since both rules delete the same message, for each message, one will be non-deterministically chosen to be applied. Rule R3-T models that, when receiving a Wait message, the train asks permission to enter the next railroad segment. Rule R4-T describes the movement of a train: it updates its attributes, sends a Traveling message to itself to be received in (at least) 10 time units (simulating the time needed to reach the end of this segment) and sends a message to the segment it was in to inform that this train is leaving. Note that this rule has a condition  $n \langle \rangle g$ , expressing that this movement may only occur if there is no gate  $g$  to enter the next position  $n$ . If there is a gate, message Go will be treated by rule R5-T, that requires the gate to open immediately (the Open message is scheduled to arrive exactly in the next time unit (without delay)). The application of rule R5-T also generates a message OpenGate, that shall arrive between 2 and 3 time units (the time needed for the gate to open), and will trigger rule R6-T, that will then move the train to the next position.

**Railroad Segment Entity:** this graph grammar is depicted in Figure 1 (type graph), Figure 2 (initial graph template RSegIni) and Figure 4 (Railroad Segment Rules). The type graph describes that each railroad segment keeps the information about its identifier (attribute id: Natural), its neighbour (the reference next), its state (busy:Boolean) and the knowledge whether it is a station or not (station:Boolean). The initial graph is given by two consecutive RSegm instances. Instances of RSegm can react to messages MayGo?, telling a train that it can either move to it (rule R2-R) or that it should wait at least 10 time units (rule R1-R), and to messages Leaving, updating its busy attribute (rule R3-R).

**Gate Entity:** the specification of the Gate Entity is shown in Figure 1 (type graph), Figure 2 (initial graph template GateIni) and Figure 4 (Gate Rules). The type graph indicates that gates have one attribute open that describes whether the gate is opened or closed. By rule R1-G, if the gate is requested to open, its closure is scheduled to occur between 5 and 8 time units, i.e. all open requests cause a delay in closing the gate. By rule R2-G, if the gate is opened and there is a close request, attribute open is modified to false.

### 3. SEMANTICS OF TIMED OBJECT BASED GRAPH GRAMMARS

As discussed in the previous section, the semantics of graph grammars is based on rule applications. First, we give the formal definitions of how these rule applications are obtained in the untimed graph grammars. Then, we explain how these rule applications can be enhanced to handle time. The resulting semantic model will be a transition system in which states correspond to reachable graphs (equipped with clocks) and transitions describe rule applications or the elapse of time.

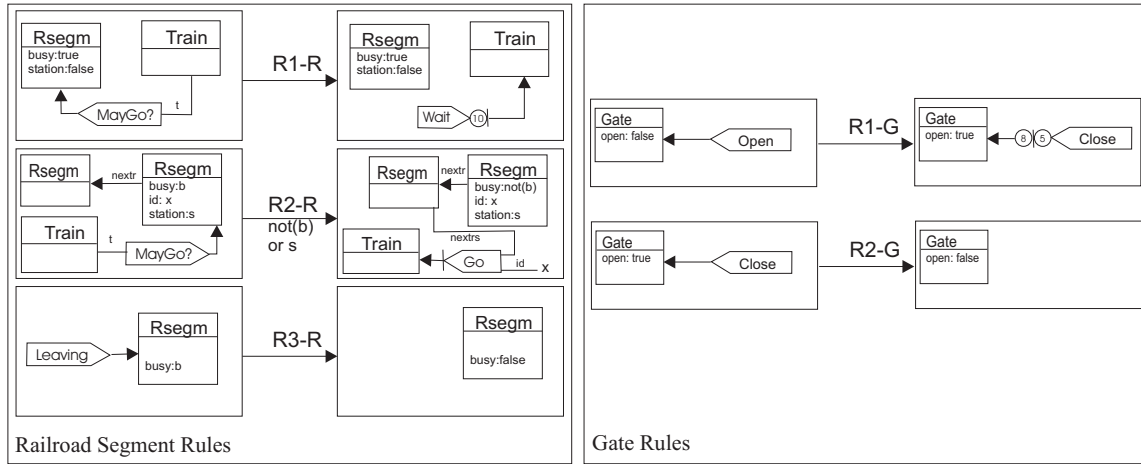


Figure 4. Railroad Segment Rules and Gate Rules

**Rule Application, Computation.** Given a rule  $r$  and a state  $G$ , we say that this rule is applicable in this state if there is a match  $m$ , that is, an occurrence of the left-hand side of the rule in the state. We denote such *rule application* by  $G \xrightarrow{(r,m)} H$ . Graphs  $G$  and  $H$  are called *input* and *output* graphs of this rule application. A rule application means that all items that are in the left-hand side of the rule and not in the right-hand side will be deleted from  $G$ , and all items that are in the right-hand side of  $r$  but not in its left-hand side will be included in  $G$  (formally, this effect can be described by a pushout in suitable categories of graphs). In a graph, there may be various vertices/edges with the same types. This means that the same rule may be applicable to a graph using different matches. A *computation of a graph grammar* is a sequence of rule applications starting with the initial graph of the grammar, and in which the output graph of one rule application is the input graph of the next one. We say that a graph (or state)  $G$  is *reachable* if there is a computation in which the output graph of the last rule application is  $G$ .

**DEFINITION 7 (RULE APPLICATION, COMPUTATION)**  
Let  $OBGG = (Spec, X, C, IG, N, n)$  be an object-based graph grammar,  $(r : L \rightarrow R, Eq)$  be a rule, and  $G$  be a typed graph over  $C$ . A *match* for  $r$  in  $G$  is a total morphism  $m : L \rightarrow G$  in  $OBGraph(C)$ . A *rule application*  $G \xrightarrow{(r,m)} H$  using rule  $r$  and match  $m$  is a pushout in the category  $OBGraph(C)$ . The morphism  $r' : G \rightarrow H$  is called *derived rule*.

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ m \downarrow & (PO) & \downarrow m' \\ G & \xrightarrow{r'} & H \end{array}$$

A *(finite) computation* of  $OBGG$  from  $IG$  to  $H$ , denoted by  $IG \xrightarrow{*} H$ , is a sequence of rule applications  $G_i \xrightarrow{(r_i, m_i)} G_{i+1}$ ,  $i \in \{0, \dots, n\}$ ,  $n \in \mathbb{N}$ , where  $G_0 = IG$ ,  $G_n = H$  and  $r_i \in Rules(C)$  for all  $i \in \{0, \dots, n\}$ . Infinite computations are defined analogously, with  $i \in \mathbb{N}$ . The class of all computations of  $OBGG$  is denoted by  $Comp_{OBGG}$ . The class of all reachable graphs in  $Comp_{OBGG}$  is defined by  $State_{OBGG} = \{G \mid G = IG \vee IG \xrightarrow{*} G \in Comp_{OBGG}\}$ .

An example of a rule application is shown in Figure 5.

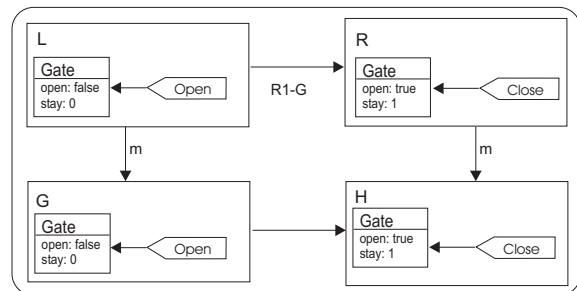


Figure 5. Example of a Rule Application

Typically, the semantics of a system described using a graph grammar is a transition system where the states are graphs and the transitions describe the possible rule applications. This semantics, however, does not take into



consideration any time restrictions. In order to define the transition system that gives semantics to a timed object-based graph grammar, we make an extension of the usual semantics, including clocks on states and allowing only rule applications that respect time restrictions.

Time is handled in the following way: a clock is assigned to each timed message, that is, to each message that is generated with some time constraint (minimum and/or maximum delivery time). This clock is initialized with zero, and, as time advances, the clock eventually reaches the minimum/maximum time. One requirement that is imposed on the semantic model is that all clocks advance simultaneously. This requirement implies that the relations among delivery times of messages in a state are preserved in the subsequent states, and this assures that the time constraints of the system are adequately modeled. Note that this does not impose a serious restriction in practice, since we are just assuming that clocks count the time in the same units and never stop until they are deallocated (in particular, this does not mean that we have a global notion of time).

**Timed State.** In the semantical model proposed here, states are described by tuples  $(G, Clocks_G, mc^G, val^G)$ , where  $G$  is a timed object-based graph,  $Clocks_G$  is a set of clock names, a function  $mc^G$  associates a clock with each timed message of  $G$ , and a function  $val^G$  associates a time (natural number) with each clock.

**DEFINITION 8 (TIMED STATE)** *Given a timed object based graph grammar  $TOBGG = (Spec, X, C, IG, N, n)$  and a timed graph  $G$  over  $C$ . The tuple  $S_G = (G, Clocks_G, mc^G, val^G)$  is called a **timed state** if (i)  $Clocks_G$  is a set of clock names; (ii) the partial function  $mc^G : mE_G \rightarrow Clocks_G$ , called **clock assignment function**, is injective and total on timed messages of  $G$ ; (iii) the total function  $val^G : Clocks_G \rightarrow \mathbb{N}$ , called **interpretation or value function**, assigns a time with each clock of  $S_G$ .*

**Timed Computations.** A state change of a TOBGG can be obtained in two ways: by an application of a rule of a TOBGG or by elapse of time. In both cases, time restrictions must be obeyed: in the case of rule applications, it must be assured that a message will be treated only within its delivery time (this is guaranteed by suitable definition of match for timed rules); in the case of time elapse, the maximum treatment time of all messages should not be violated (this is guaranteed by forbidding computations that would lead to inconsistent states, that is, states that do not satisfy the time restrictions). A timed computation is defined by a sequence of such state changes (corresponding to rule application or time elapse). Therefore, by construction, a timed computation guarantees that time

restrictions will never be violated.

**DEFINITION 9 (TIMED RULE APPLICATION)** *Let  $TOBGG = (Spec, X, C, IG, N, n)$  be a timed object-based graph grammar,  $(r : L \rightarrow R, Eq)$  be a rule, and  $S_G = (G, Clocks_G, mc^G, val^G)$  be a timed state. A **match** for  $r$  in  $S_G$  is a total morphism  $m : L \rightarrow G$  in  $\mathbf{OBGraph}(C)$  such that if  $trigger(r)$  is a timed message then (i)  $val^G(mc^G(m_E(trigger(r)))) \geq t_{min}^G(m_E(trigger(r)))$ ; (ii)  $val^G(mc^G(m_E(trigger(r)))) \leq t_{max}^G(m_E(trigger(r)))$ .*

A **timed rule application**  $S_G \xrightarrow{(r,m)} S_H$  using rule  $r$  and match  $m$  generates a timed state  $S_H = (H, Clocks_H, mc^H, val^H)$  obtained as follows:  $H$  is the graph resulting from application of rule  $r$  at match  $m$  in graph  $G$ ;  $Clocks_H = Clocks_G - \{mc^G(m_E(trigger(r)))\} \cup \{c_i | i \text{ is a message created in } H\}$ ; for each timed message  $msg$  of  $mE_H$

$$mc^H(msg) = \begin{cases} mc^G(msg), & \text{if } msg \text{ was preserved by } r \\ c_{msg} & , \text{ if } msg \text{ was created by } r \end{cases}$$

and for each clock name  $c \in Clocks_H$  corresponding to a message  $msg$  of  $H$

$$val^H(c) = \begin{cases} val^G(c) & , \text{ if } msg \text{ was preserved by } r \\ 0 & , \text{ if } msg \text{ was created by } r \end{cases}$$

**DEFINITION 10 (TIMED COMPUTATION)** A **(finite) timed computation** of a timed object-based graph grammar  $TOBGG = (Spec, X, C, IG, N, n)$  between timed states  $S_{IG}$  and  $S_H$ , denoted by  $S_{IG} \xrightarrow{*} S_H$ , is a sequence of transitions  $S_{G_i} \xrightarrow{lab} S_{G_{i+1}}$ ,  $i \in \{0, \dots, n\}$ ,  $n \in \mathbb{N}$ , where  $G_0 = IG$ ,  $S_{G_n} = S_H$  and  $lab$  can be:

$(r_i, m_i)$ : in this case,  $G_i \xrightarrow{(r_i, m_i)} G_{i+1}$  must be a rule application of rule  $r$  at match  $m_i$ ;

$\delta$ : a non-negative value corresponding to the elapse of time from state  $S_{G_i}$  to state  $S_{G_{i+1}}$ . In this case,  $S_{G_{i+1}}$  is obtained as follows:  $G_{i+1} = G_i$ ,  $Clocks_{G_{i+1}} = Clocks_{G_i}$ ,  $mc^{G_{i+1}} = mc^{G_i}$ ,  $val^{G_{i+1}}(c) = val^{G_i}(c) + \delta$ , for all  $c \in Clocks_{G_{i+1}}$ . This transition may only occur in case the following restriction is satisfied: for all timed message  $msg$  of  $G_i$  with corresponding clock  $c_{msg}$ ,

$$t \leq t_{max}^{G_i}(msg), \text{ where}$$

$$val^{G_i}(c_{msg}) < t \leq val^{G_{i+1}}(c_{msg}).$$

Infinite computations are defined analogously, with  $i \in \mathbb{N}$ . The class of all timed computations of TOBGG is denoted by  $Comp_{TOBGG}$ . The class of all reachable graphs in  $Comp_{TOBGG}$  is defined by  $State_{TOBGG} = \{G \mid G = IG \vee IG \xrightarrow{*} G \in Comp_{TOBGG}\}$ .

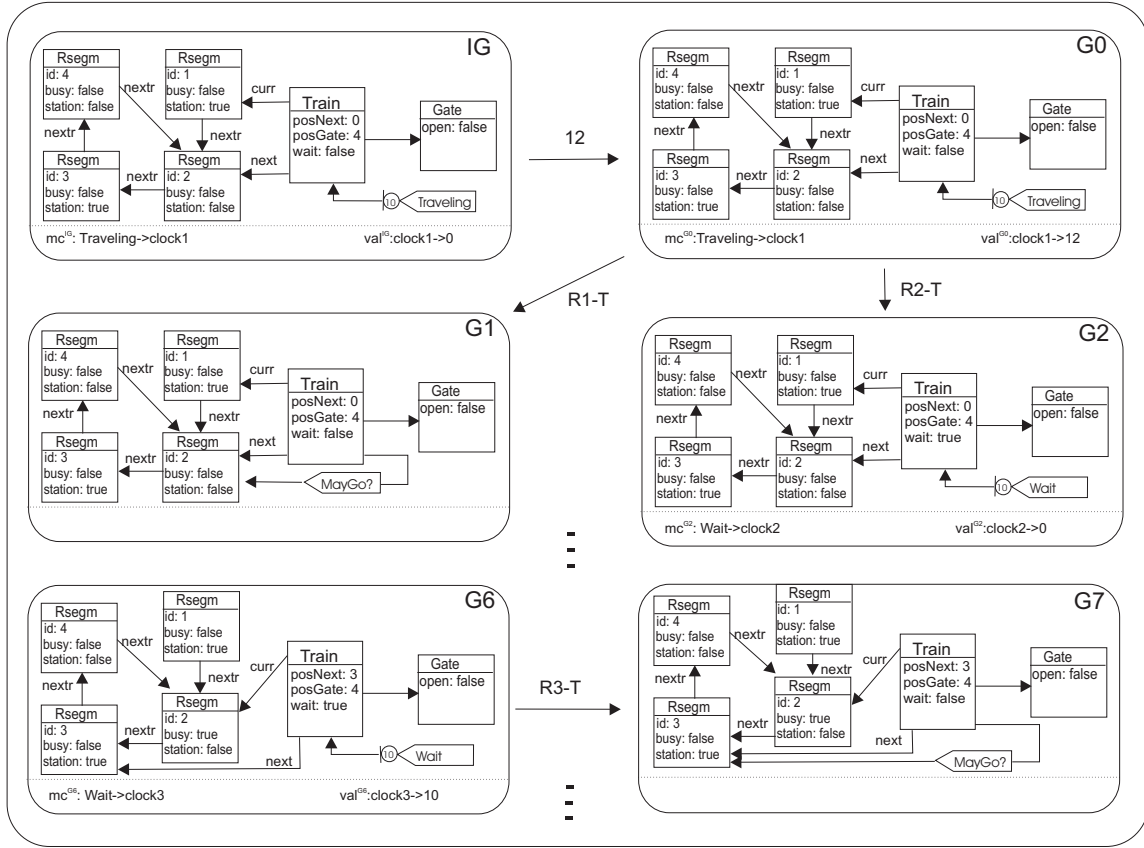


Figure 6. (Part of the) Transition System for the Railroad System

### Semantics of TOBGG.

Now, the semantics of timed object-based graph grammars can be defined: it is a transition system in which states are timed states and transitions correspond to rule applications that respect time restrictions (minimum/maximum delivery times of the message handled by the rule), or transitions that update clocks (all clocks simultaneously). The latter should also respect message delivery time restrictions: a clock can only be updated if the maximum time of the corresponding message is not violated. By construction, this transition system corresponds to the class of all timed computations of a TOBGG.

**DEFINITION 11 (SEMANTICS OF TOBGG)** Given a timed object-based graph grammar  $TOBGG = (Spec, X, C, IG, N, n)$ , its semantics is the transition system  $TS = (IS, States, Lab, Tran)$  defined by:

**Initial State:**  $IS = (IG, Clocks_{IG}, mc^{IG}, val^{IG})$  where  $Clocks_{IG} = \{c_{msg} | msg \text{ is a timed message of } IG\}$ ;  $mc^{IG}(msg) = c_{msg}$ , for all timed messages  $msg$  of  $IG$ ;  $val^{IG}(c) = 0$ , for all clocks  $c \in Clocks_{IG}$ .

**States:** The set of states contains all states  $S$  that are reached by timed computations that is  $States_{TOBGG}$  (see Def. 10).

**Transitions:** A transition  $S1 \xrightarrow{lab} S2$  is in  $Tran$  if there exists a timed computation  $tcomp$  of  $TOBGG$  starting at  $IS$  that contains this transition.

**Lab :** is the set of all labels of transitions in  $Tran$ .

Figure 6 illustrates part of the transition system obtained for the Railroad System presented in Subsection 2.2. Rectangles represent states and arrows model transitions (state changes). States are composed by a graph  $G$  and by functions  $mc^G$  and  $val^G$  (represented in the lower part of rectangles), that associate, respectively, a clock with each timed message of graph  $G$  and a time with each clock. Transitions labeled by a non-negative value correspond to state changes due to elapse of time and transitions labeled by a rule name correspond to state changes caused by the application of the respective rule. For example, from the initial state  $IG$  to state  $G0$  the transition labeled by 12 just updates  $clock1$ , and the transition from  $G0$  to  $G2$  corresponds to the application of rule  $R2-$

T. In the latter case, *Traveling* is the trigger message (it is consumed) and a timed message *Wait* is created. The clock associated with this message, *clock2*, is initialized with zero.

To be able to perform automatic verification of timed object-based graph grammars, we translate each TOBGG to an equivalent timed automaton, and use the existing tools to verify properties of timed automata to check the TOBGG. Since our semantic definition for TOBGG was highly inspired by timed automata, the comparison of the transition systems generated by TOBGG and timed automata is straightforward.

#### 4. TIMED AUTOMATA

Timed Automata [1, 4] are used to specify and verify real-time systems. To express the behavior of a system with time restrictions, Timed Automata extend Nondeterministic Automata with a finite set of clocks. In this model states and transitions are associated to clock constraints. A clock constraint is a conjunction of atomic constraints, which compare clock variables with a constant value (a nonnegative rational value). Formally, let  $x$  be a clock in a set  $Clocks$  of clock variables and  $c$  be a constant in  $\mathbb{Q}$ , then the set  $\phi(Clocks)$  of *clock constraints*  $\varphi$  is defined by the grammar:

$$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2,$$

A clock constraint associated to a state (named *invariant*) indicates how many time units the system may remain on a certain state. The constraint of a transition represents its activation conditions. Moreover, each transition is associated to a set (possibly empty) of clocks that are reset with the occurrence of this transition.

**DEFINITION 12 (TIMED AUTOMATON)** A *timed automaton*  $TA$  is a tuple  $(L, L^0, \Sigma, Clocks, I, E)$ , where:

- $L$  is a set of states;
- $L^0 \subseteq L$  is a set of initial states;
- $\Sigma$  is a set of labels;
- $Clocks$  is a finite set of clocks;
- $I$  is a mapping that labels each state  $s$  in  $L$  with some clock constraint in  $\phi(Clocks)$ ;
- $E \subseteq L \times \Sigma \times 2^{Clocks} \times \phi(Clocks) \times L$  is a set of transitions. Each tuple  $(s, a, \varphi, \lambda, s')$  represents a transition from state  $s$  to a state  $s'$  labeled with  $a$ .  $\varphi$  is a clock constraint over  $Clocks$  that specifies when the transition is enabled (it may be the empty constraint  $\varepsilon$ ), and the set  $\lambda \subseteq Clocks$  gives the clocks to be reset with this transition.

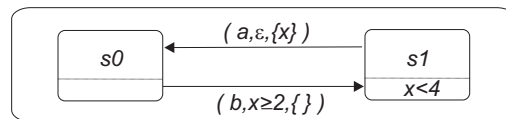


Figure 7. Timed Automaton

Figure 7 shows an example of a timed automaton where  $s0$  and  $s1$  represent the states of the system. The clock constraint  $x < 4$  in state  $s1$  means that the system can remain in this state while the clock value  $x$  is less than four. The transitions are  $(s0, b, x \geq 2, \{ \}, s1)$  and  $(s1, a, \varepsilon, \{x\}, s0)$ .

To each timed automaton  $TA$  we can associate a corresponding transition system [1]. The possible transitions are the ones specified in  $TA$ , and transitions that increment the clocks (all clocks are incremented simultaneously). All transitions and reachable states must satisfy the time restrictions.

Formally, the semantics of a timed automaton  $A$  is defined by associating a transition system  $S_A$  with it. Each state of  $S_A$  is a pair  $(s, val)$ , such that  $s$  is a location of  $A$  and  $val$  is a clock interpretation for  $Clocks$  such that  $val$  satisfies the invariant  $I(s)$ . The set of all states of  $A$  is denoted by  $Q_A$ . A state  $(s, val)$  is an initial state if  $s$  is an initial location of  $A$  and  $val(x) = 0$  for all clocks  $x$ . There are two types of transitions in  $S_A$  [1]:

**Time elapse:** for a state  $(s, val)$  and a real-valued time

$$\delta \geq 0, (s, val) \xrightarrow{\delta} (s, val + \delta) \text{ if for all } \delta \geq \delta' \geq 0, \text{ } val + \delta' \text{ satisfies the invariant } I(s);$$

**Automaton transition:** for a state  $(s, val)$  and a transition  $(s, a, \phi, \lambda, s')$  such that  $v$  satisfies  $\phi$ ,  $(s, val) \xrightarrow{a} (s', val[\lambda := 0])$ .

Thus,  $S_A$  is a transition system with label-set  $\Sigma \cup \mathbb{R}$ .

#### 5. TRANSLATION OF TIMED OBJECT BASED GRAPH GRAMMARS TO TIMED AUTOMATA

In this section we define formally how to obtain a timed-automaton that grasps this idea of behavior of timed OBGs. If the initial state, the set of rules and the set of reachable states of a grammar are finite<sup>2</sup>, a finite timed automaton is generated. However, we can only define a timed automaton for grammars that reach states with a bounded number of messages (because the number of clocks in an automaton is fixed). If we consider

<sup>2</sup>We assume that we have only one representative for each isomorphism class of graphs in the set of reachable states.

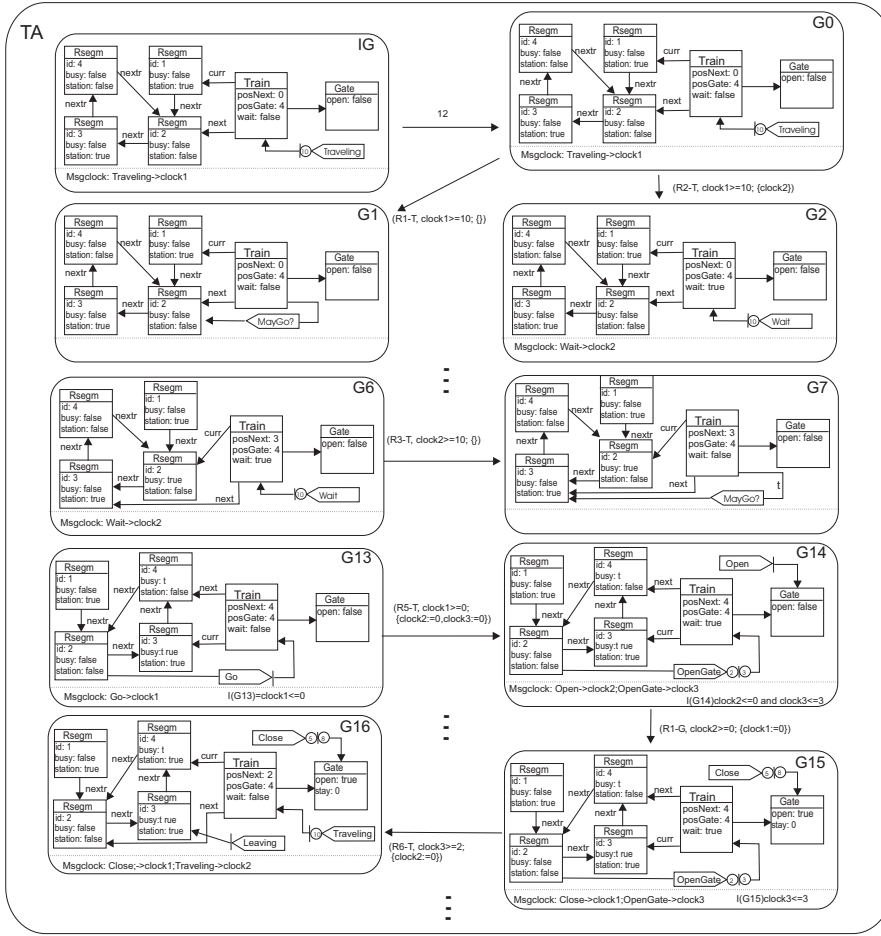


Figure 8. (Part of) the Timed Automaton for the Railroad System

finite-state systems (that are the ones that are possible to automatically verify), this imposes no restriction. To define this automaton, states are described by pairs  $(G, msgclock^G)$ , where  $G$  is a timed object-based graph and  $msgclock^G$  is defined as follows:

**Clock assignment function (msgclock):** Given a graph  $G$  and a set of clocks  $Clocks$ , the clock assignment function  $msgclock^G : mE_G \rightarrow Clocks$  is a partial injective function that assigns a clock to each timed message of  $G$  (analogous to definition of function  $mc$  in Def. 8);

**$x$ -tmessage bounded graph grammar:** Grammar  $GG$  is  $x$ -tmessage bounded, where  $x$  is a natural number, if there is no reachable state of  $GG$  in which there are more than  $x$  timed messages.

**DEFINITION 13 (TRANSLATION OF TOBGG TO TA)**  
Let  $TOBGG = (Spec, X, C, IG, N, n)$  be an  $x$ -

*tmessage bounded timed object-based graph grammar. The translation of TOBGG to the timed automaton  $TA = (L, L^0, \Sigma, Clocks, I, E)$  is given by:*

- $L = \{(G, msgclock^G) | G \text{ is reachable in TOBGG and } msgclock^G \text{ is a clock assignment function}\};$
- $L^0 = (IG, msgclock^{IG});$
- $\Sigma = N \times Mor(timedOBGraph(C))^3;$
- $Clocks = \{clock_1, \dots, clock_x\};$
- $I(G, msgclock^G)$  is the conjunction of all formulas  $val(msgclock^G(msg)) \leq t_{max}^G(msg)$ , with  $msg \in mE_G$  and  $t_{max}^G(msg) \in \mathbb{N};$
- $E$  is the set of all transitions  $((G, msgclock^G), a, \varphi, \lambda, (H, msgclock^H))$  such that

<sup>3</sup> $Mor(timedOBGraph(C))$  is the class of all morphisms in the category **timedOBGraph(C)**. Labels of transitions of  $TA$  are pairs of rule names and matches (that are morphisms in **timedOBGraph(C)**).

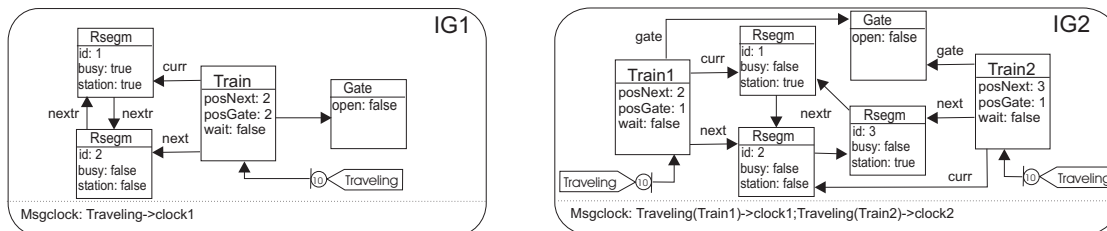


Figure 9. Examples of Initial States

- $\exists$  a rule application  $G \xrightarrow{(r,m)} H$ , let  $trigger(r) = msg$  and  $msgclock^G(msg) = c$ ;
- $a = (r, m)$ ,
- $\varphi = \begin{cases} (val(c) \geq t_{min}^G(msg)), & \text{if } t_{min}^L(msg) \text{ is defined} \\ \varepsilon & , \text{if } t_{min}^L(msg) \text{ is undefined} \end{cases}$
- $\lambda$  is the set of clocks assigned to the timed messages created by the rule application.

The clock constraint (invariant)  $(val(c) \leq t_{max}(msg))$  on states  $(I(s))$  assures that the system may stay in state  $s$  at most until the clocks of all timed messages of  $s$  are less than their respective maximum time limits (because after this time, there will be at least one message in  $s$  that was not delivered in time). The clock constraint on transitions  $\varphi$  assures that, if a message has a minimum time to be delivered ( $t_{min}^L(msg)$ ), it will not be processed before this time, if a message does not have such restriction, it may be processed at any time (the restrictions on maximum times for delivery are modeled by function  $I$ , as described above). Moreover,  $\lambda$  indicates which clocks shall be reset with the transitions, these are all the clocks used in the newly created timed messages in the target state of the transition.

Now, this construction is illustrated by an example. Figure 8 shows a (partial) timed automaton (TA) that was obtained from translation of example in Subsection 2.2. The initial state of TA is the state IG. G1, G2, G6, G7, G13, G14, G15 and G16 are some states of the automaton TA (that are obtained with rules application from IG). The time constraints were inserted following Definition 13. The complete timed automaton for this example has 36 states and 40 transitions. In the graph representations the matches are omitted.

Due to the construction of the timed automaton based on rule applications over reachable states of TOBGG, this semantics is compatible with a traditional semantics based on sequences of rule applica-

tions presented in Section 3: whenever there is a transition  $((G, msgclock^G), (r, m), \varphi, \lambda, (H, msgclock^H))$  in the timed automaton, there is a corresponding transition from graph  $G$  to graph  $H$  corresponding to applying rule  $r$  using match  $m$  to graph  $G$ . The converse is also true because timed rule applications always respect the minimum/maximum delivery times of messages, that is, a timed rule application  $S_G \xrightarrow{(r,m)} S_H$  can always be translated to a corresponding transition  $((G, msgclock^G), (r, m), \varphi, \lambda, (H, msgclock^H))$ , where the clock assignment functions are basically the same (up to renaming of the used clocks). Note that, in the TOBGG transition system there is no limit in the number of created clocks, whereas in timed automaton there is a fixed set of clocks. Therefore, this full semantical compatibility can be achieved only for grammars that do not generate an unbounded number of timed messages (since there is a clock for each timed message, such grammars would have to be translated to timed automata with infinite number of clocks, and this is not possible according to the definition of timed automata). The transitions that correspond to time elapse in both semantical models are defined basically in the same way: they must guarantee that maximum delivery times of messages will not be violated. Thus, the resulting transition systems will have exactly the same time elapse transitions.

In Figure 9 we present two other possibilities of initial states (IG1, IG2) for the example in Subsection 2.2. Note that if we change the initial state, we are actually specifying a new system, since the properties that may or not hold are highly dependent on the state the system starts to execute. Graph IG1 presents a situation in which there is only one train traveling in a circular railroad consisting of two segments (called 1 and 2), and with a gate that should be opened to allow the passage from segment1 to segment 2. The timed automaton that represents the TOBGG with the initial graph IG1 has 21 states and 24 transitions. Graph IG2 models a circular railroad with 3 segments in which two trains are traveling. It also contains one gate. For the TOBGG with initial graph IG2, the timed automa-

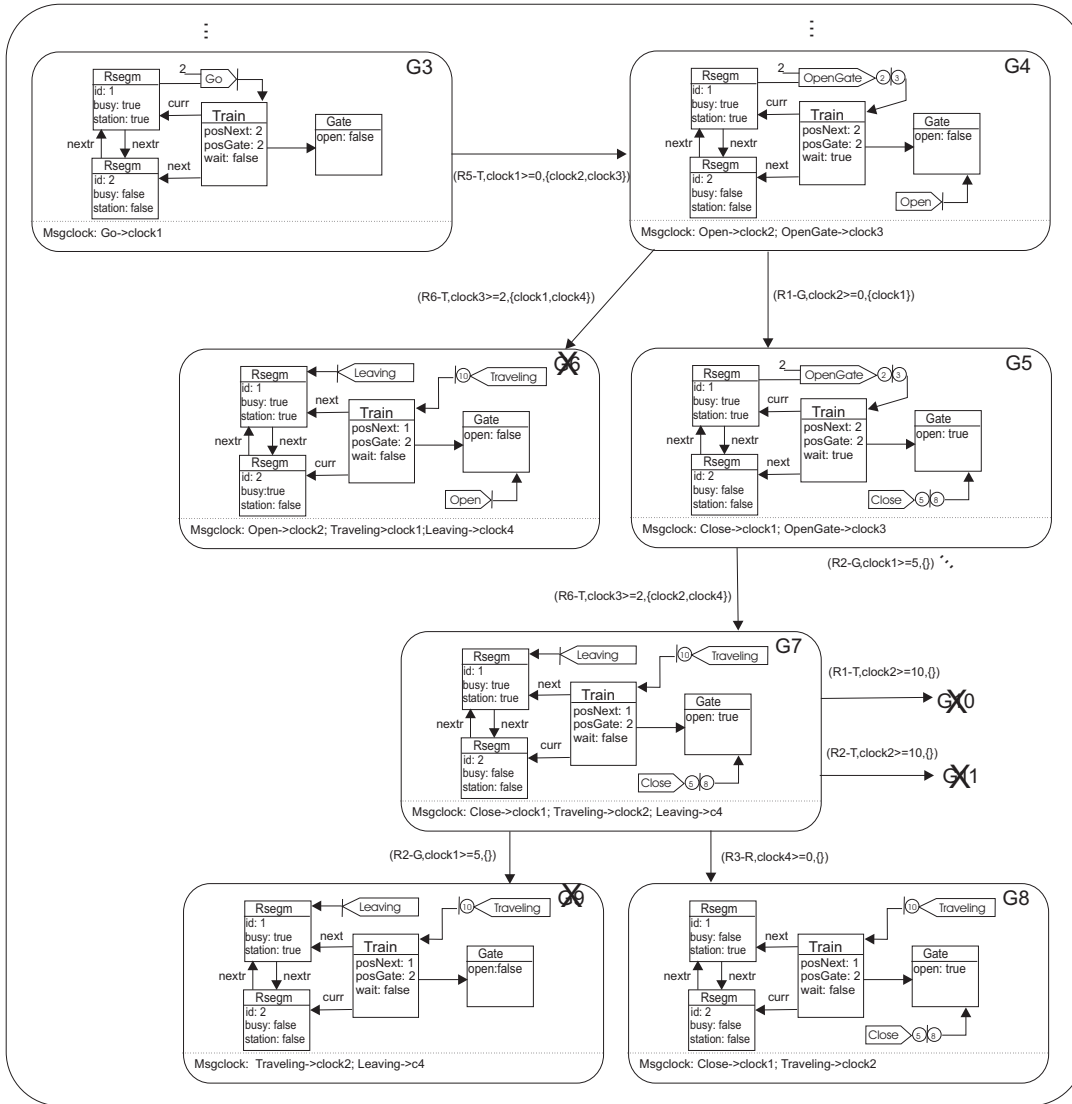


Figure 10. (Part of the) Timed Automaton for the Initial Graph IG1

ton has 123 states and 264 transitions. These numbers do not consider unreachable states and transitions.

## 6. VERIFICATION

For simulation and verification of properties of real-time systems we chose to use Uppaal (version 3.4.11) [3], a toolkit developed by Uppsala University and Aalborg University. Uppaal is a tool for validation (via graphical simulation) and verification (via automatic model-checking) that has timed automata as the input language and a subset of CTL as the specification language. The simulator can be used in three ways: the user can run the system manually and choose which transitions to per-

form, the random mode can be toggled to let the system run on its own, or the user can go through a trace (saved or imported from the verifier) to see how certain states are reachable. The verifier is designed to check a subset of CTL (Computation Tree Logic). The formulas to be checked must be in one of the following formats:

- $A[]\phi$ : for all paths,  $\phi$  always hold;
- $E \langle \rangle \phi$ : there exists a path where  $\phi$  eventually holds;
- $A \langle \rangle \phi$ : for all paths,  $\phi$  will eventually hold;
- $E[]\phi$ : there exists a path where  $\phi$  always holds;

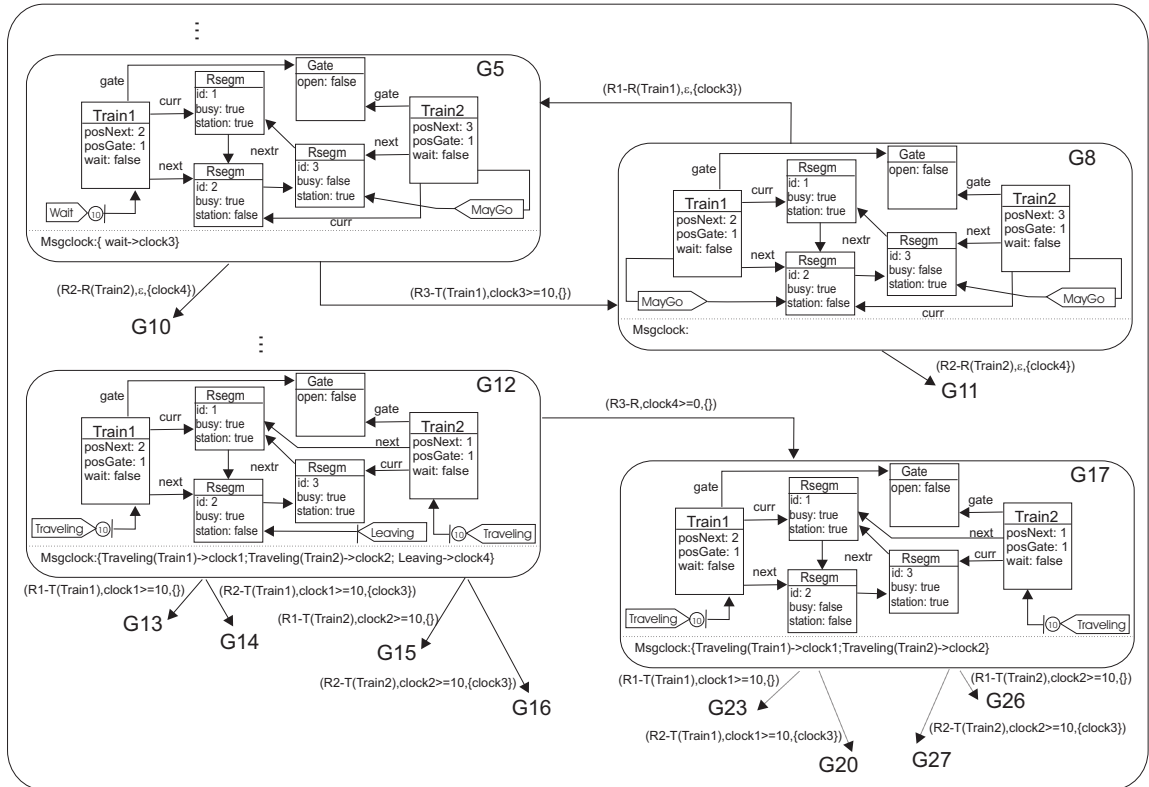


Figure 11. (Part of the) Timed Automaton for the Initial Graph IG2

- $\phi - - > \psi$ : whenever  $\phi$  holds  $\psi$  will eventually hold.

where  $\phi \in \psi$  are Boolean expressions that can refer to states, integer variables and clocks constraints. The word **deadlock** can be used to verify deadlocks.

Safety properties mean that something bad will never happen. To check these properties in Uppaal, we use the forms  $A[]\phi$  or  $E[]\phi$ . For example, the property  $A[]$  not deadlock was checked for the railroad system of section 2.2 (with initial graph IG (Figure 8)) and was satisfied. This indicates the absence of deadlock for all paths of the system.

Reachability properties are the simplest forms of properties. They specify whether a given property  $\phi$  can be satisfied in some reachable state. The form  $E \langle \langle \phi \rangle \rangle$  is used to check these properties. For example, we can check if there is one path in which state G7 of automaton TA in Figure 8 (that represents the situation when train asks permission to pass to the railroad segment identified with id 3) is reachable and clock2 has a value less than 10 time units ( $E \langle \langle \text{TA.G7 and clock2} < 10 \rangle \rangle$ ). This property was checked and was not satisfied because when the system reaches state G7, the value of clock2 must be already greater than 10 (this is the condition of the transi-

tion to reach this state).

Liveness properties are used to check if something good will eventually happen. These properties are expressed by the forms  $A \langle \langle \phi \rangle \rangle$  and  $\phi - - > \psi$ . For example,  $\text{TA.G13} - - > \text{TA.G16}$  means that if state G13 of TA occurs, then G16 will occur, too. This establishes that if the train can travel to a railroad segment that has a gate (situation represented by state G13) the gate will eventually open and the train will travel to this segment (represented by state G16). This property is satisfied by the example.

Graph IG1 of Figure 9 illustrates a state with one train, two railroad segments and one gate. Considering this initial state and the same rules for the behavior of trains, gate and railroad segments, we obtain a TOBGG whose corresponding timed automaton is partially depicted in Figure 10. For this system, we verified the following properties:

- $A[]$  not deadlock: the system never deadlocks, that is, in any reachable state, there is always a rule that can be applied.
- $\text{TA1.G3} - - > \text{TA1.G4}$ : the train always requests the opening before passing a gate. In this case, G3 corresponds to the state in which there is a gate to en-

ter the next railroad segment and G4 is the state in which the gate receives the opening request.

- $E \langle \rangle TA1.x$ : the unreachability of some states. In this formula,  $x$  corresponds to a state which is not reachable due to clock constraints. For example, considering  $x = G6$ , the property is not satisfied. State G4 which originates G6 possesses two timed messages: Open that must be handled immediately and OpenGate that must be handled in at least 2 time units and in at most 3 time units. Therefore, respecting the clock constraints only state G5 can be reached. The property  $x = G9$  is not satisfied either, since state G7 which originates G9 has the message Leaving which must be handled in current time.

For the automaton that represents the TOBGG with IG2 as initial graph (Figure 9, with two trains, three railroad segments and one gate), we verified the following properties (Figure 11 shows part fo the timed automaton obtained for this example):

- $A \square$  not deadlock: the system never deadlocks.
- $TA2.G8 \dashv \dashv > TA2.G5$ : a train doesn't enter a railroad segment with another train. Here, G8 represents a state in which a train tries to enter a segment (which is not a station) which is already occupied by another train. In G5 the train receives a message to wait until the other train leaves the segment.
- $TA2.G12 \dashv \dashv > TA2.G17$ : when a train leaves a railroad segment, the segment is released. This property establishes that if a train has left a segment (situation represented by state G12), the segment will be eventually released (represented by state G17).

## 7. FINAL REMARKS

In this paper we introduced Timed Object-Based Graph Grammars (TOBGGs), an extension of Object-Based Graph Grammars that includes the notion of time, allowing the specification and analysis of real-time systems. Time stamps on the messages describe when they are to be delivered/handled. A translation of TOBGG to Timed Automata provided a way to verify properties over TOBGG specifications using the Uppaal model-checker. The main reason for extending Object-Based Graph Grammars is that, besides being formal, they are quite intuitive even for people not used to formal description languages. This is an advantage of graph grammars comparing to other specification methods such as real-time process algebras and Timed Petri Nets, since the object-based style is really appealing and is the most widely used in practice. The specification style proposed

here is specially suitable for reactive systems, and this feature is not common in other object-based formalisms because they typically offer only synchronous message passing schemes. Our choice of having asynchronous communication allows a natural description of reactive systems, but has the drawback that when synchronous communication is needed, the specifier has to explicitly introduce state variables and messages with corresponding rules to assure that an object remains blocked until a message arrives indicating that another object is synchronized with it (simulating this way a synchronous message passing scheme). The choice of specification method shall always take into account the main characteristics of the application being modeled, and also the features offered by the specification formalism. For inherently synchronous systems, formalisms based on process algebras may be more adequate. For asynchronous systems, TOBGGs have the advantage over timed Petri nets or timed automata because of the object-based style (that naturally leads to a modular structure of a system).

Since there exists already a rich theory of concurrency for graph grammars [25, 13], it would be interesting to investigate to which extent main results can be obtained in the case of timed OBGs. Another topic of future work is how to lift constructions like the compositional verification method introduced in [24] and the inheritance and polymorphism concepts [15] to TOBGGs.

## ACKNOWLEDGMENTS

This is an extended version of the paper "Specification of real-time systems with graph grammars" presented at the 20th Brazilian Symposium on Software Engineering (SBES 2006). This work was partially supported by CNPq.

## REFERENCES

- [1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [2] J. Baeten and C. Middelburg. *Process algebra with timing: Real time and discrete time*, chapter 10, pages 627–684. Elsevier, 2001.
- [3] G. Behrmann, A. David, and K. G. Larsen. Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of LNCS, pages 200–236. Springer, 2004.
- [4] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of LNCS, pages 87–124. Springer, 2004.



- [5] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, 1991.
- [6] H. Bowman and J. Derrick. Extending lotos with time: A true concurrency perspective. In *AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software*, volume 1231 of *LNCS*, pages 382–399. Springer, 1997.
- [7] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
- [8] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.
- [9] K. Diethers and M. Huhn. Voodoo: Verification of object-oriented designs using uppaal. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 139–143. Springer, 2004.
- [10] F. L. Dotti, L. Foss, L. Ribeiro, and O. M. Santos. Verification of distributed object-based systems. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 261–275. Springer, 2003.
- [11] F. L. Dotti and L. Ribeiro. Specification of mobile code systems using graph grammars. In *Formal Methods for Open Object-Based Distributed Systems*, pages 45–64. Kluwer, 2000.
- [12] L. M. Duarte, F. L. Dotti, B. Copstein, and L. Ribeiro. Simulation of mobile applications. In *Proc. of Communication Networks and Distributed Systems Modeling and Simulation Conference*, volume 1, pages 1–15, 2002.
- [13] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.
- [14] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification: Equations and Initial Algebra Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
- [15] A. P. L. Ferreira. *Object-Oriented Graph Grammars*. PhD thesis, Universidade Federal do Rio Grande do Sul, 2005.
- [16] S. Graf, I. Ober, and I. Ober. A real-time profile for uml. *International Journal on Software Tools for Technology Transfer*, 8(2):113–127, 2006.
- [17] A. Haxthausen and J. Peleska. Formal development and verification of a distributed railroad control system. *IEEE Transactions on Software Engineering*, 26(8):687–701, August 2000.
- [18] A. Knapp, S. Merz, and C. Rauh. Model checking - timed uml state machines and collaborations. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 395–416. Springer, 2002.
- [19] S. Konrad, L. A. Campbell, and B. H. C. Cheng. Automated analysis of timing information in uml diagrams. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 350–353. IEEE Computer Society, 2004.
- [20] L. Lavazza, G. Quaroni, and M. Venturelli. Combining uml and formal notations for modelling real-time systems. *SIGSOFT Softw. Eng. Notes*, 26(5):196–206, 2001.
- [21] I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proc. of the IEEE*, 82(1):158–171, 1994.
- [22] I. Ober, S. Graf, and I. Ober. Validation of uml models via a mapping to communicating extended timed automata. In *SPIN*, volume 2989 of *LNCS*, pages 127–145. Springer, 2004.
- [23] L. Ribeiro, F. Dotti, and R. Bardohl. A formal framework for the development of concurrent object-based systems. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 385–401. Springer, 2005.
- [24] L. Ribeiro, F. L. Dotti, O. Santos, and F. Pasini. Verifying object-based graph grammars: An assume-guarantee approach. *Software and Systems Modeling*, 6(3):289–311, Sept. 2006.
- [25] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
- [26] J. A. Stankovic. A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, 1988.
- [27] J. A. Stankovic et al. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4):751–763, 1996.

- [28] B. Walter. Timed petri-nets for modelling and analyzing protocols with real-time characteristics. In *Protocol Specification, Testing, and Verification*, pages 149–159. North-Holand, 1983.