Specification of Real-Time Systems with Graph Grammars

Leonardo Michelon[1], Simone André da Costa[1][2], Leila Ribeiro[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre – RS – Brazil

[2]Universidade do Vale do Rio dos Sinos (UNISINOS)
São Leopoldo – RS – Brazil

`lmichelon@inf.ufrgs.br, scosta@unisinos.br, leila@inf.ufrgs.br`

Abstract. *This paper presents a formal approach to specify and analyze real-time systems. We extend Object-Based Graph Grammars, a description technique suitable for the specification of asynchronous distributed systems, to be able to explicitly model time constraints. The semantics of the systems is defined in terms of Timed Automata, allowing the automatic verification of properties.*

Resumo. *Este artigo apresenta uma abordagem formal para a especificação e análise de sistemas de tempo real. Gramáticas de Grafos Baseadas em Objetos são extendidas incluindo primitivas para modelar explicitamente restrições de tempo.. A semântica é definida em termos de autômatos temporais, provendo um método para verificação automática de propriedades.*

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 [Clarke and Wing 1996]. Besides providing means to prove that a system satisfies the required properties, formal methods contribute to its understanding, revealing ambiguities, inconsistencies and incompletions 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 [Stankovic et al. 1996, Stankovic 1988]. 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 Automata [Alur and Dill 1994, Bengtsson and Yi 2004] is one of the most proiminent 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 [Lee et al. 1994, Baeten and Middelburg 2000, Bowman and Derrick 1997] 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 [Walter 1983] and time Petri nets [Berthomieu and Diaz 1991] 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 [Knapp et al. 2002] 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 [Behrmann et al. 2004] is called to check if the Observer Timed Automaton describes a reachable behavior of the system. [Diethers and Huhn 2004] proposes a similar approach through the Vooduu tool. Nevertheless, these tools restrict the specification of a real-time system. First of all, because they are based on UML state machines, that only support *after* and *when* time constructors. They 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 [Lavazza et al. 2001] 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 [Konrad et al. 2004] 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. [Graf et al. 2003] proposes an extension of a UML subset with timing constructs. In [Ober et al. 2004], part of UML is 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) [Dotti and Ribeiro 2000, Ribeiro et al. 2005] 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 then described via applications of these rules to graphs describing the actual states of a system. Rules operate locally on the state-graph, and therefore it is possible that many rules are applied at the same time. OBGGs are appealing as specification formalism because they are formal, they are based on simple but powerful concepts to describe behavior, and at the same time 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 [Duarte et al. 2002] and verification (using the SPIN model-checker) [Dotti et al. 2003]. Compositional verification (using an assume-guarantee approach) is also provided [Ribeiro et al. 2006]. Moreover, there is an extension of OBGGs to model inheritance and polymorphism [Ferreira 2005]. 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 a timed semantics of OBGGs. The paper is organized as follows: Section 2 presents OBGGs and its timed extension; Section 3 reviews Timed Automata; the semantics of timed OBGGs is described in Section 4; Section 5 analyzes the example; and final remarks are in Section 6.

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.

Following, 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 4. For the comprehension of the other contributions of this paper, it is possible to skip the formal definitions. Due to space limitations, the examples are in Subsection 2.2.
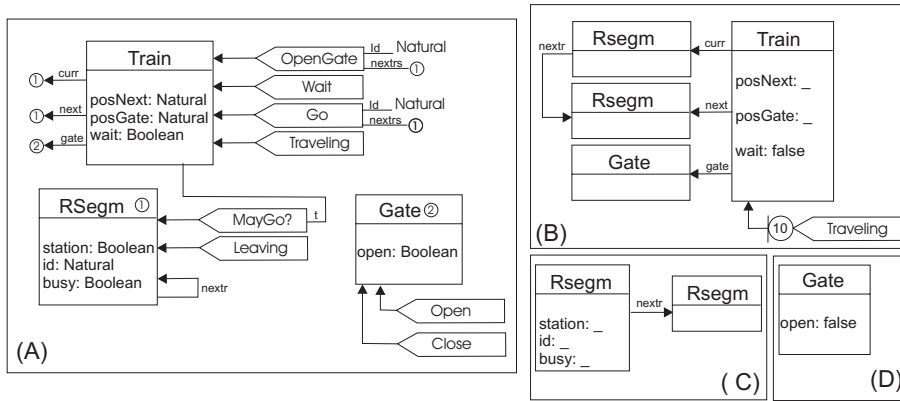
Figure 1. (A) Type Graph, (B) Train Initial Graph Template, (C) RSegm Initial Graph Template and (D) Gate Initial Graph Template

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(A) 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 RSgm object and 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 , $f^{\bullet}$ denotes the function $f^{\bullet} : dom(f) \rightarrow B$, s.t, $f(x) = f^{\bullet}(x)$, $\forall x \in dom(f)$. If $Spec$ is an algebraic specification, $\mathcal{U} :$ **Alg(Spec)** $\rightarrow$ **Set** is 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., [Ehrig and Mahr 1985]).

DEFINITION 1 (GRAPH, GRAPH MORPHISM) *Given an algebraic specification $Spec$, 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).*

$$dom(E_G) \overset{g_E^{\bullet}}{\longmapsto} E_H \qquad dom(E_G) \overset{g_E^{\bullet}}{\longmapsto} E_H \qquad dom(vV_G) \overset{g_V^{\bullet}}{\longmapsto} vV_H$$

$$s^G \downarrow \quad = \quad \downarrow s^H \qquad t^G \downarrow \quad = \quad \downarrow t^H \qquad a^G \downarrow \quad = \quad \downarrow a^H$$

$$V_G^* \xrightarrow{g_V^*} V_H^* \qquad oV_G \xrightarrow{g_V^*} oV_H \qquad \mathcal{U}(A_G) \xrightarrow{\mathcal{U}(g_A)} \mathcal{U}(A_H)$$

OB-Graphs. An *OB-graph* is a graph equipped with a morphism *type* to a fixed graph of types [Corradini et al. 1996]. 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[1] 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 $OG^C = (OG, type^{OG})$ where OG is an graph called **instance graph** and $type^{OG} : OG \rightarrow C$ 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 $x \in dom(f)$, $type^{OG_1}(x) = type^{OG_2} \circ f(x)$. 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 will be 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.

---

[1]An algebra in which each carrier set is a singleton.

- 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)*, $trigger(r) \notin dom(r_E)$;*
3. *only attributes of the target of the message may appear in L: $(aE_L = \emptyset) \vee ((\exists! e \in aE_L) \wedge t^L(e) = t^L(trigger(r)))$;*
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: $\forall o \in oV_L.o \in dom(r_V)$;*
6. *the algebra of r is a quotient term algebra over the specification $Spec^{Nat}$ including a set of equations Eq and variables in X;*
7. *attributes appearing in L may only be variables of X: $\forall v \in vV_L.a^L(v) \in X$;*
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. 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.
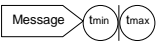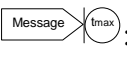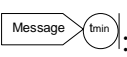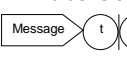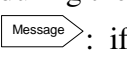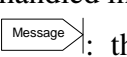
DEFINITION 4 (OBJECT-BASED GRAPH GRAMMAR (OBGG)) *An **object-based graph grammar** is a tuple $OBGG = (Spec, X, C, IG, N, n)$ where $Spec$ is an algebraic specification, X is a set of variables, C is a class graph, IG is a C-typed graph, called **initial or start graph**, N is a set of rule names, $n : N \rightarrow Rules(C)$ assigns a rule to each rule name.*

2.1. Timed OBGG

Originally, the Graph Grammars formalism does not include the concept of time. Here we will 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. 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. This means this message must be handled immediately, i.e., in interval $[current\ time,\ current\ time]$.

Now, we will define timed OBGGs, short TOBGGs 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 $t_{min}^G, t_{max}^G : mE_G \to \mathbb{N}$, 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** $g : G \to H$ 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).*

$$dom(g_E) \cap dom(t_{max}) \overset{g_E^{\bullet}}{\longmapsto} mE_H \qquad dom(g_E) \cap dom(t_{min}) \overset{g_E^{\bullet}}{\longmapsto} mE_H$$

$$t_{max}{}^{G\bullet} \Big\uparrow \qquad = \qquad \Big\uparrow t_{max}^{H} \qquad\qquad t_{min}{}^{G\bullet} \Big\uparrow \qquad = \qquad \Big\uparrow t_{min}^{H}$$

$$\mathbb{N} \xrightarrow[id]{} \mathbb{N} \qquad\qquad\qquad \mathbb{N} \xrightarrow[id]{} \mathbb{N}$$

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. The railroad system is composed of instances of three entities: Train, Gate and RSegm.

Train Entity: the graph grammar of the Train Entity is depicted in Figures 1 (A) and (B) (type graph and initial graph template) and 2 (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 (pos-Next), the identifier of the gate (posGate), and a state attribute that describes whether the train is moving or waiting (wait).

Trains may receive four kinds of messages: Wait, Go, OpenGate and Traveling. Messages OpenGate and Go have parameters. The initial graph template in Figure 1 (B) 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 need 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 2. 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
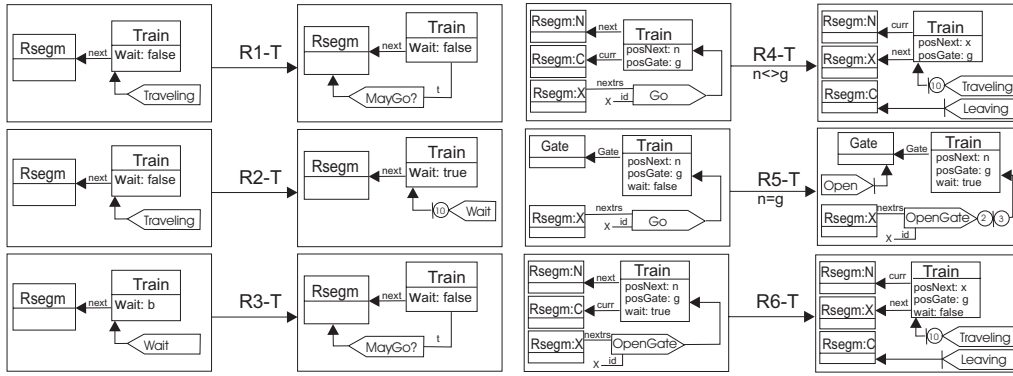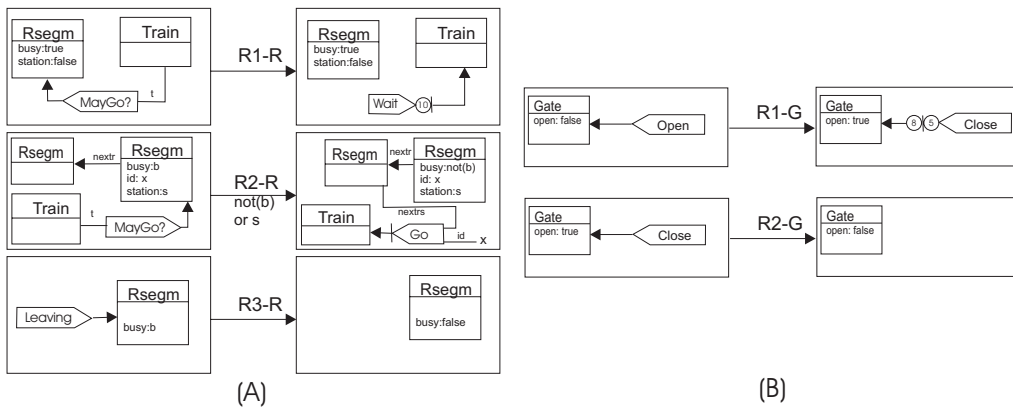
Figure 2. Train Rules

Figure 3. (A) Railroad Segment Rules, (B) Gate Rules

segment it was in to inform that this train is leaving. Note that this rule has a condition n<>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 Figures 1(A), 1(C) and 3 (type graph, initial graph template and rules, respectively). The type graph describes that each railroad segment keeps the information about its identifier (attribute id: Natural), its neighbour (the reference nextr), 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 (A) and (D) and rules in Figure 3 (B). 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.
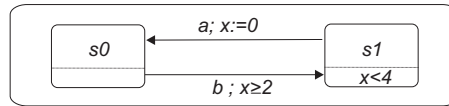
Figure 4. Timed Automaton

3. Timed Automata

Timed Automata ([Alur and Dill 1994], [Bengtsson and Yi 2004]) 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 X of clock variables and c be a constant in $\mathbb{Q}$, then the set $\phi(X)$ of *clock constraints* φ 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 7 (TIMED AUTOMATON) *A timed automaton TA is a tuple $(L, L^0, \Sigma, X, I, E)$, where:*

- *L is a set of states;*
- *$L^0 \subseteq L$ is a set of initial states;*
- *Σ is a set of labels;*
- *X is a finite set of clocks;*
- *I is a mapping that labels each state s in L with some clock constraint in $\phi(X)$;*
- *$E \subseteq L \times \Sigma \times 2^X \times \phi(X) \times L$ 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. φ is a clock constraint over X that specifies when the transition is enabled (it may be the empty constraint ε), and the set $\lambda \subseteq X$ gives the clocks to be reset with this transition.*

Figure 4 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 [Alur and Dill 1994]. 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.

4. Semantics of Timed Object Based Graph Grammars

As discussed in Section 2, the semantics of graph grammars is based on rule applications. Due to space limitations, we will not give the formal definitions of how these rule applications are obtained (via pushouts in the corresponding category). Instead, we will explain how this application is done and how time is handled. Then, we will define formally how to obtain a timed-automata that grasps this idea of behavior of timed OBGGs.
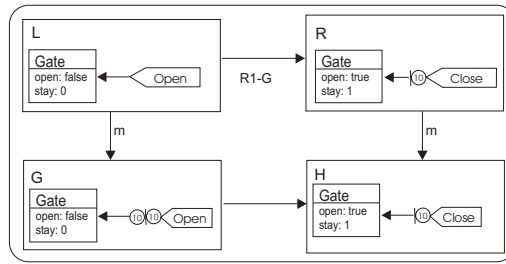
Figure 5. Example of a Rule Application

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 (formally, this is modeled by the existence of a total graph morphism from the left-hand side of the rule to the state graph). We denote such *rule application* by $G \overset{(r,m)}{\Longrightarrow} H$. An example of a rule application is shown in Figure 5. Graphs G and H are called *input* and *output* graphs of this rule application. Since in the left-hand side graph L the t_{min} and t_{max} attributes are undefined (see Definition 6), there are no time constrains on when this rule is applicable. These constrains will be handled by the timed automata (that will define the semantics of a TOBGG), rule applications will be used just to obtain the states of the automata.

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. 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 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 will show how to build a timed-automata that has the desired behavior and respects the required time constraints. If the initial state, the set of rules and the set of reachable states of a grammar are finite[2], a finite timed automata will be generated.

Time will be handled in the following way: a clock will be 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, this clock will eventually reach the minimum/maximum time. Since all clocks advance simultaneously in timed automata, the relations among the 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. However, this means that we can only define a timed automata for grammars that reach states with an arbitrary number of messages (because the number of clocks in an automata is fixed). If we consider finite-state systems (that are the ones that are possible to automatically verify), this imposes no restriction.

To define this automata, states will be described by pairs $(G, msgclock^G)$, where

---

[2]We assume that we have only one representative for each isomorphism class of graphs in the set of reachable states.

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 X, the clock assignment function $msgclock^G : mE_G \rightarrow X$ is a partial injective function that assigns a clock to each timed message of G;

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 8 (SEMANTICS OF A **TOBGG**) *Let* $TOBGG = (Spec, X, C, IG, N, n)$ *be an x-tmessage bounded timed object-based graph grammar. The **semantics** of $TOBGG$ is the timed automata $TA = (L, L^0, \sum, X, I, E)$ where:*

- $L = \{(G, msgclock^G) | G$ *is reachable in $TOBGG$ and*

  $msgclock^G$ *is a clock assignment function*$\}$*;*
- $L^0 = (IG, msgclock^{IG})$*;*
- $\sum = N$*;*
- $X = \{clock_1, ..., clock_x\}$*;*
- $I(G, msgclock^G)$ *is the conjunction of all formulas* $msgclock^G(msg) \leq t_max^G(msg)$*, with* $msg \in mE_{IG}$ *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*
  - $\exists$ *a rule application* $G \xRightarrow{(r,m)} H$*, let* $trigger(r) = tr$*;*
  - $a = (r, m)$*,*
  - $\varphi = \begin{cases} (msgclock^G(tr) \geq t_{min}^G(tr)) & \text{, if } min = t_{min}^L(tr) \\ \\ \varepsilon & \text{, if } t_{min}^L(tr) \text{ is undefined} \end{cases}$
  - λ *is the set of clocks assigned to the timed messages created by the rule application.*

The clock constraint (invariant) $(msgclock(msg) \leq t_{max}(msg))$ 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 φ assures that, if a message has a minimum time to be delivered $(t_{min}^L(trigger(r)))$, 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, λ 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 will be illustrated by an example. Figure 6 shows a (partial) timed automata (TA) that was obtained from translation of example in Section 2.2. The initial state of TA is the state IG. G3, G4, G6, G7, G13, G14, G15 and G16 are some states of the automata TA (that are obtained with rules application from IG). The time constraints were inserted following Definition 8. The complete timed automata for this example has 36 states and 40 transitions.

Due to the construction of the timed automata based on rule applications over reachable states of $TOBGG$, this semantics would be compatible with a traditional semantics based on sequences of rule applications in the following sense: whenever there is
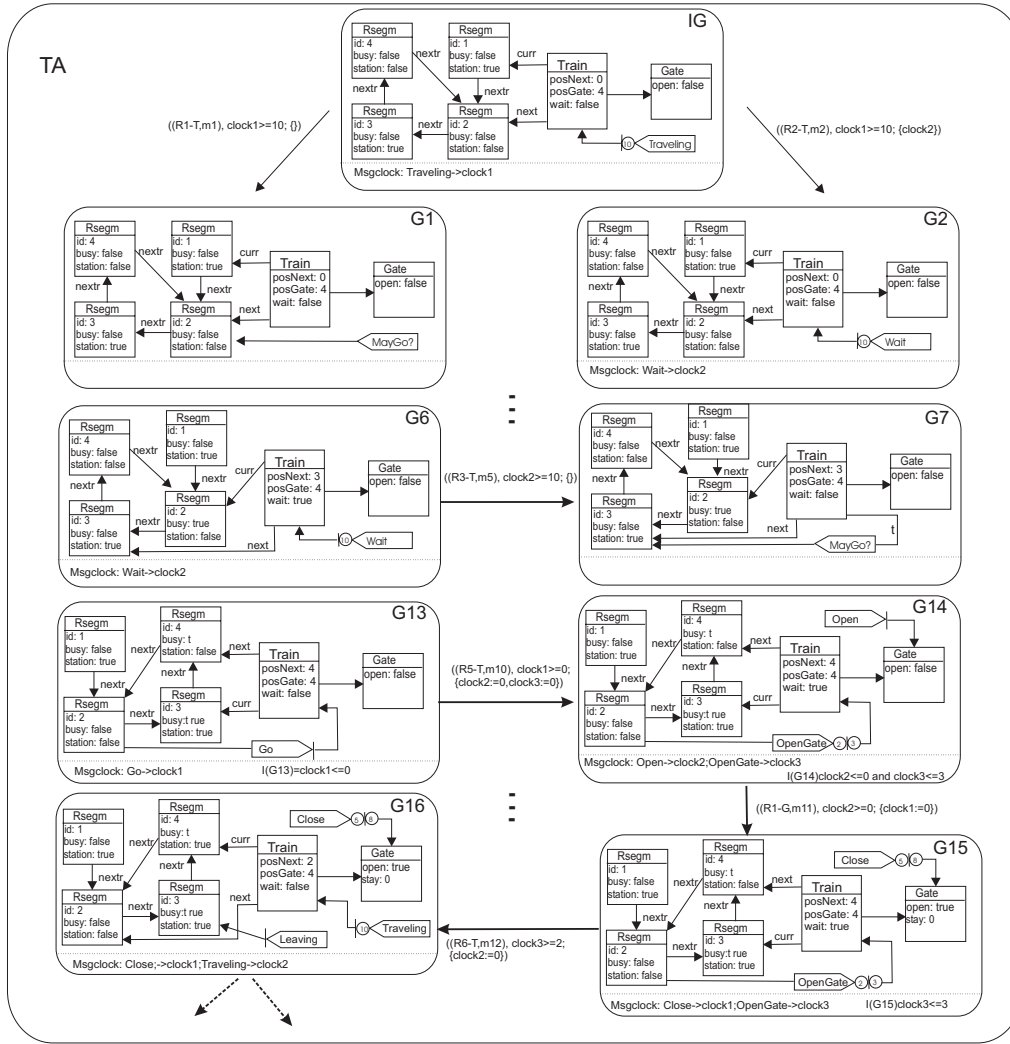
Figure 6. (Part of the) Timed Automata for the Railroad System

a transition $((G, msgclock^G), (r, m), \varphi, \lambda, (H, msgclock^H))$ in the timed automata, there is a corresponding transition from graph G to graph H corresponding to applying rule r using m to graph G. The converse might not be true because there may be rule applications that do not obey the time restrictions (violate either minimum or maximum delivery time of messages). Thus, some graphs (states) that are reachable in computations of a grammar will not be reachable in the corresponding timed automata. Note that there is no incompatibility of semantical definitions here, since we have defined the semantics of timed object-based graph grammars in terms of timed automata. The definition of a semantics purely based on rule applications, and corresponding proof of equivalence, is left for future work.

In Figure 7 we present two other possibilities of initial states (IG1, IG2) for the example in Section 2.2. The timed automaton that represents the TOBGG with the initial graph IG1 has 21 states and 24 trasitions. For the TOBBG with initial graph IG2, the timed automaton has 123 states and 264 transitions. These numbers do not consider unreachable states and transitions.
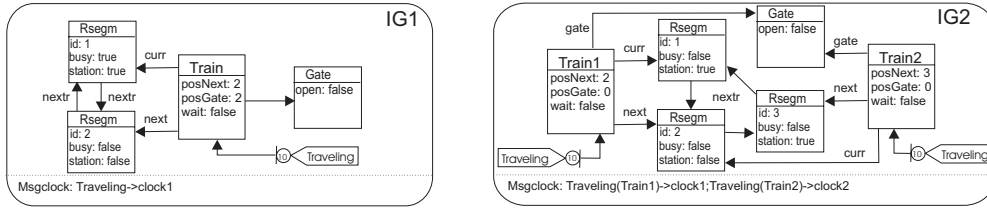
Figure 7. Examples of initial states

5. Verification

For simulation and verification of properties of real-time systems we chose to use Uppaal [Behrmann et al. 2004], a toolkit developed by Uppssala 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 perform, 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 forms:

- $A[]\phi$: for all paths, ϕ always hold;
- $E <> \phi$: there exists a path where ϕ eventually holds;
- $A <> \phi$: for all paths, ϕ will eventually hold;
- $E[]\phi$: there exists a path where ϕ always holds;
- $\phi - - > \psi$: whenever ϕ holds ψ will eventually hold.

where ϕ e ψ 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 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 6)) and was satisfied. This indicates the absence of deadlock for all paths of the system.

Reachability properties are the simplest form of properties. They specify whether a given property ϕ can be satisfied in some reachable state. The form $E <> \phi$ is used to check these properties. For example, we can check if there is one path in which state G7 of automata TA in Figure 6 (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 ($E <>$ TA.G7 and clock2< 10). This property was checked and was not satisfied because when the system reaches state G7, the value of clock2 must be already bigger than 10 (this is the condition of the transition to reach this state).

Liveness properties are used to check if something good will eventually happen. These properties are expressed by the forms $A <> \phi$ and $\phi - - > \psi$. For example, TA.G13$-- >$TA.G16 that means if state G13 of TA occurs, then G16 will occur, too. This means that if the train can travel to a railroad segment that have 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 for the example.

For the automaton that represents the TOBGG with IG1 as initial graph (Figure 7, with one train, two railroad segments and one gate), we verified the following properties: the system never deadlocks; the train always requests the opening before passing a gate; and the unreachability of same states.

For the automaton that represents the TOBGG with IG2 as initial graph (Figure 7, with two trains and three railroad segments without gate), we verified the following properties: the system never deadlocks; a train never enters in a railroad segment with another train; when a train leaves a railroad segment, the segment is released.

6. 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. The semantics of TOBGG was defined in terms of Timed Automata. This 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 usual semantics of graph grammars is the set of computations generated by rule applications, we plan to define a semantics for TOBGG analogously, without a translation to timed automata. This would involve including clocks in the state graphs and using conditions for rule applications to assure that time restrictions are not violated. In this case, to be able to use the Uppal verification tool, we would have to provide a proof of the equivalence of the two semantics. Another topic of future work is how to lift constructions like the compositional verification method introduced in [Ribeiro et al. 2006] and the inheritance and polymorphism concepts [Ferreira 2005] to TOBGGs.

References

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

Baeten, J. and Middelburg, C. (2000). Process algebra with timing: Real time and discrete time. In *Handbook of Process Algebra*, chapter 4. Elsevier.

Behrmann, G., David, A., and Larsen, K. G. (2004). Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 200–236. Springer.

Bengtsson, J. and Yi, W. (2004). Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer.

Berthomieu, B. and Diaz, M. (1991). Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273.

Bowman, H. and Derrick, J. (1997). 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.

Clarke, E. M. and Wing, J. M. (1996). Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643.

Corradini, A., Montanari, U., and Rossi, F. (1996). Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265.

Diethers, K. and Huhn, M. (2004). Vooduu: 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.

Dotti, F. L., Foss, L., Ribeiro, L., and Santos, O. M. (2003). 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.

Dotti, F. L. and Ribeiro, L. (2000). Specification of mobile code systems using graph grammars. In *Formal Methods for Open Object-Based Distributed Systems*, pages 45–64. Kluwer.

Duarte, L. M., Dotti, F. L., Copstein, B., and Ribeiro, L. (2002). Simulation of mobile applications. In *Proc. of Communication Networks and Distributed Systems Modeling and Simulation Conference*, volume 1, pages 1–15.

Ehrig, H. and Mahr, B. (1985). Fundamentals of algebraic specification: Equations and initial algebra semantics. In *EATCS Monographs on Theoretical Computer Science*, volume 6. Springer.

Ferreira, A. P. L. (2005). *Object-Oriented Graph Grammars*. PhD thesis, Universidade Federal do Rio Grande do Sul.

Graf, S., Ober, I., and Ober, I. (2003). Timed annotations with UML. In *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems*.

Knapp, A., Merz, S., and Rauh, C. (2002). 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.

Konrad, S., Campbell, L. A., and Cheng, B. H. C. (2004). 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.

Lavazza, L., Quaroni, G., and Venturelli, M. (2001). Combining uml and formal notations for modelling real-time systems. *SIGSOFT Softw. Eng. Notes*, 26(5):196–206.

Lee, I., Brémond-Grégoire, P., and Gerber, R. (1994). A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proc. of the IEEE*, 82(1):158–171.

Ober, I., Graf, S., and Ober, I. (2004). Validation of uml models via a mapping to communicating extended timed automata. In *SPIN*, volume 2989 of *LNCS*, pages 127–145. Springer.

Ribeiro, L., Dotti, F., and Bardohl, R. (2005). 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.

Ribeiro, L., Dotti, F. L., Santos, O., and Pasini, F. (2006). Verifying object-based graph grammars: An assume-guarantee approach. Accepted for publication in Software and Systems Modeling.

Stankovic, J. A. (1988). A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19.

Stankovic, J. A. et al. (1996). Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4):751–763.

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