Conversion to Region Automata

Convert timing constants to integers

- For both RA #1 and RA #1a, $c_x=2$; regions of the automata include: $x=0$, $0<x<1$, $x=1$, $1<x<2$, $x=2$, $x>2$
- An open segment in one dimension is a successor of itself, as is a transition along a principle diagonal in more than one dimension
- RA #1 $\Rightarrow$ RA #1a shows that we can collapse common modes after the outgoing transitions are determined to be the same
- The conversion of forced transition semantics (e.g. $x=1$) is not discussed in the theory (Alur & Dill), though it could be extended to do this by using epsilon transitions
- The conversion can create race conditions or instantaneous transitions
Timed Automata: Semantics, Algorithms and Tools

Johan Bengtsson and Wang Yi

Uppsala University
Email: {johanb,yi}@it.uu.se

Abstract. This chapter is to provide a tutorial and pointers to results and related work on timed automata with a focus on semantical and algorithmic aspects of verification tools. We present the concrete and abstract semantics of timed automata (based on transition rules, regions and zones), decision problems, and algorithms for verification. A detailed description on DBM (Difference Bound Matrices) is included, which is the central data structure behind several verification tools for timed systems. As an example, we give a brief introduction to the tool UPPAAL.

1 Introduction

Timed automata is a theory for modeling and verification of real time systems. Examples of other formalisms with the same purpose, are timed Petri Nets, timed process algebras, and real time logics [BD91, RRR88, Yi91, NS94, AH94, Cha99]. Following the work of Alur and Dill [AD90, AD94], several model checkers have been developed with timed automata being the core of their input languages e.g. [Yov97, LPY97]. It is fair to say that they have been the driving force for the application and development of the theory. The goal of this chapter is to provide a tutorial on timed automata with a focus on the semantics and algorithms based on which these tools are developed.

In the original theory of timed automata [AD90, AD94], a timed automaton is a finite-state Buchi automaton extended with a set of real-valued variables modeling clocks. Constraints on the clock variables are used to restrict the behavior of an automaton, and Buchi accepting conditions are used to enforce progress properties. A simplified version, namely Timed Safety Automata is introduced in [HNSY94] to specify progress properties using local invariant conditions. Due to its simplicity, Timed Safety Automata has been adopted in several verification tools for timed automata e.g. UPPAAL [LPY97] and Kronos [Yov97]. In this presentation, we shall focus on Timed Safety Automata, and following the literature, refer them as Timed Automata or simply automata when it is understood from the context.

The rest of the chapter is organized as follows: In the next section, we describe the syntax and operational semantics of timed automata. The section also addresses decision problems relevant to automatic verification. In the literature, the decidability and undecidability of such problems are often considered to be the fundamental properties of a computation model. Section 3 presents the abstract version of the operational semantics based on regions and zones. Section 4 describes the data structure DBM (Difference
Bound Matrices) for the efficient representation and manipulation of zones, and operations on zones, needed for symbolic verification. Section 5 gives a brief introduction to the verification tool UPAL. Finally, as an appendix, we list the pseudo-code for the presented DBM algorithms.

2 Timed Automata

A timed automaton is essentially a finite automaton (that is a graph containing a finite set of nodes or locations and a finite set of labeled edges) extended with real-valued variables. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then increase synchronously with the same rate. Clock constraints i.e. guards on edges are used to restrict the behavior of the automaton. A transition represented by an edge can be taken when the clocks values satisfy the guard labeled on the edge. Clocks may be reset to zero when a transition is taken.

The first example Fig. 1(a) is an example timed automaton. The timing behavior of the automaton is controlled by two clocks $x$ and $y$. The clock $x$ is used to control the self-loop in the location loop. The single transition of the loop may occur when $x = 1$. Clock $y$ controls the execution of the entire automaton. The automaton may leave start at any time point when $y$ is in the interval between 10 and 20; it can go from loop to end when $y$ is between 40 and 50, etc.

![Fig. 1. Timed Automata and Location Invariants](image)

Timed Büchi Automata A guard on an edge of an automaton is only an enabling condition of the transition represented by the edge; but it can not force the transition to be taken. For instance, the example automaton may stay forever in any location, just
idling. In the initial work by Alur and Dill [AD90], the problem is solved by introducing Büchi-acceptance conditions; a subset of the locations in the automaton are marked as accepting, and only those executions passing through an accepting location infinitely often are considered valid behaviors of the automaton. As an example, consider again the automaton in Fig. 1(a) and assume that end is marked as accepting. This implies that all executions of the automaton must visit end infinitely many times. This imposes implicit conditions on start and loop. The location start must be left when the value of y is at most 20, otherwise the automaton will get stuck in start and never be able to enter end. Likewise, the automaton must leave loop when y is at most 50 to be able to enter end.

Timed Safety Automata A more intuitive notion of progress is introduced in timed safety automata [HNSY94]. Instead of accepting conditions, in timed safety automata, locations may be put local timing constraints called location invariants. An automaton may remain in a location as long as the clocks values satisfy the invariant condition of the location. For example, consider the timed safety automaton in Fig. 1(b), which corresponds to the Büchi automaton in Fig. 1(a) with end marked as an accepting location. The invariant specifies a local condition that start and end must be left when y is at most 20 and loop must be left when y is at most 50. This gives a local view of the timing behavior of the automaton in each location.

In the rest of this chapter, we shall focus on timed safety automata and refer such automata as Timed Automata or simply automata without confusion.

2.1 Formal Syntax

Assume a finite set of real-valued variables $C$ ranged over by $x, y$ etc. standing for clocks and a finite alphabet $\Sigma$ ranged over by $a, b$ etc. standing for actions.

Clock Constraints A clock constraint is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ for $x, y \in C, \sim \in \{\leq, <, =, >, \geq\}$ and $n \in N$. Clock constraints will be used as guards for timed automata. We use $B(C)$ to denote the set of clock constraints, ranged over by $g$ and also by $D$ later.

**Definition 1 (Timed Automaton)** A timed automaton $A$ is a tuple $\langle N, l_0, E, I \rangle$ where

- $N$ is a finite set of locations (or nodes),
- $l_0 \in N$ is the initial location,
- $E \subseteq N \times B(C) \times \Sigma \times 2^C \times N$ is the set of edges and
- $I : N \rightarrow B(C)$ assigns invariants to locations

We shall write $l \xrightarrow{g,a,r} l'$ when $(l, g, a, r, l') \in E$. 
Networks of Timed Automata A network of timed automata is the parallel composition \( A_1 \cdots A_n \) of a set of timed automata \( A_1, \ldots, A_n \), called processes, combined into a single system by the CCS parallel composition operator with all external actions hidden. Synchronous communication between the processes is by hand-shake synchronization using input and output actions; asynchronous communication is by shared variables as described later. To model hand-shake synchronization, the action alphabet \( \Sigma \) is assumed to consist of symbols for input actions denoted \( a? \), output actions denoted \( a! \), and internal actions represented by the distinct symbol \( \tau \).

An example system composed of two timed automata is shown in Fig. 14. The network models a time-dependent light-switch (to the left) and its user (to the right). The user and the switch communicate using the label \( \text{press} \). The user can press the switch (\( \text{press!} \)) and the switch waits to be pressed (\( \text{press?} \)). The product automaton, i.e. the automaton describing the combined system is shown in Fig. 15.

![Fig. 14. Network of Timed Automata](image)

The semantics of networks is given as for single timed automata in terms of transition systems. A state of a network is a pair \( \langle l, u \rangle \) where \( l \) denotes a vector of current locations of the network, one for each process, and \( u \) is as usual a clock assignment remembering the current values of the clocks in the system. A network may perform two types of transitions, delay transitions and discrete transitions. The rule for delay transitions is similar to the case of single timed automata where the invariant of a location vector is the conjunction of the location invariants of the processes. There are two rules for discrete transitions defining local actions where one of the processes makes a move on its own, and synchronizing actions where two processes synchronize on a channel and move simultaneously.

Let \( l_i \) stand for the \( i \)th element of a location vector \( l \) and \( l[l_i'/l_i] \) for the vector \( l \) with \( l_i \) being substituted with \( l_i' \). The transition rules are as follows:

- \( \langle l, u \rangle \xrightarrow{d} \langle l, u + d \rangle \) if \( u \in I(l) \) and \( u + d \in I(l) \), where \( I(l) = \bigwedge I(l_i) \)
- \( \langle l, u \rangle \xrightarrow{\tau} \langle l[l_i'/l_i], u' \rangle \) if \( l_i \xrightarrow{\tau} l_i' \), \( u \in \rho \), \( u' = [\tau \mapsto 0]u \), \( u' \in I(l[l_i'/l_i]) \)
- \((l, u) \xrightarrow{r_i} (l'[l_i/l_i][l'_j/l_j], u')\) if there exist \(i \neq j\) such that

1. \(l_i \xrightarrow{g_i \cdot \alpha_i \cdot r_i} l'_i, l_j \xrightarrow{g_j \cdot \alpha_j \cdot r_j} l'_j\) and \(u \in g_i \wedge g_j\), and

2. \(u' = (r_i \cup r_j \rightarrow 0) \circ u\) and \(u' \in I((l'[l_i/l_i][l'_j/l_j])\).

Note that a network is a closed system which may not perform any external action. In fact, the CCS hiding operator is embedded in the above rules.

**Shared Integer Variables** Clocks may be considered as typed variables with type `clock`. In UPPAAL, one may also use integer variables and arrays of integers, each with a bounded domain and an initial value. Predicates over the integer variables can be used as guards on the edges of an automaton process and the integer variables may be updated using resets on the edges. In the current version of UPPAAL, the syntax related to integer variables resembles the standard C syntax. Both integer guards and integer resets are standard C expressions with the restriction that guards cannot have side-effects.

The semantics of networks can be defined accordingly. The clock assignment \(u\) in the state configuration \((l, u)\) can be extended to store the values of integer variables in addition to clocks. Since delay does not affect the integer variables, the delay transitions are the same as for networks without integer variables. The action transitions are extended in the natural way, i.e., for an action transition to be enabled the extended clock assignment must also satisfy all integer guards on the corresponding edges and when a transition is taken the assignment is updated according to the integer and clock resets.
Timed Automata with Asynchronous Processes: Schedulability and Deidability

ELENA FERGIAN, PAUL PETTENSON, AND WANG YI

tasks in P are known. Thus, each task P is characterized as a pair of natural numbers defined (P, D) with C ≤ D, where C is the worst-case execution time of P and D is the relative deadline for P. We shall use (P, D) and (P′, D′) to denote the worst-case execution time and relative deadline of P respectively.

As in timed automata, assume a finite alphabet Act ranged over by a, b, . . . and a finite set of clock-related circles C ranged over by c1, c2, . . . . We use RC(c) ranged over by c to denote the set of clock-related formulas of atomic statements in the form x1 < c or x1 > c, and x1 = c, where x1, x2 ∈ Act ∧ c ∈ C and c > 0.

C, D are natural numbers. The elements of RC(c) are called clock constraints.

Definition 1. A timed automaton extended with tasks, over actions Act, clocks C and tasks P is a tuple (N, E, I, M) where

- (N, E, I) is a timed automaton where
  - N is a finite set of locations ranged over by L, m, n
  - c ∈ C is the initial location and
  - E, N ∈ RC(c) x Act x N x N is the set of edges
  - F : N → RC(c) is a function assigning each location with a clock constraint
- (x, y) ∈ P is a partial function assigning location with tasks

Intuitively, in a given transition an automaton determines an event triggering a task, assumes the task, and the time on the transition specifies all the possible actual times of the event (or the occurrence of the task). Whenever a task is triggered, it will be put in the scheduling (or task) queue for execution (corresponding to the ready queue in operating systems).

Clearly extended timed automata are as least as expressive as timed automata. In particular, if M is the empty mapping, we will have ordinary timed automata. It is a rather general and expressive model. For example, it can model time-triggered periodic tasks as a simple automaton as shown in Figure 2(a) where P is a periodic task with computing time 2, deadline 8 and period 4. More generally, it can model systems containing both periodic and sporadic tasks as shown in Figure 2(b) which is a system consisting of 4 tasks operating on location L with periods 2, 3 and 4 respectively (specified by the constraints x1=2 and x2=20, by P2 and P4 are periodic or event driven by event a and b respectively).

In general, there may be a number of released tasks running logically in parallel. For example, if in a system of C1, C2, C3, task C1 may be released before the preceding tasks C2 is finished because there is no constraint on the arrival time of C1. The system that the queue contains at least P1 and C3. In fact, instances of all four task types may appear in the queue at the same time.

1) Tasks may have other parameters such as fixed priority for scheduling and other resource requirements, e.g., memory consumption. Since this is beyond the scope of this paper, we only consider computing time and deadlines.

2) Note that M is a partial function meaning that some of the locations may have no tasks. Also, that we may also associate a location with a set of tasks instead of a single one. It will not introduce technical difficulties.

Shared Variables. To have a more general model, we may introduce data variables shared among automata and tasks. For example, shared variables can be used to model precedence relations and synchronization between tasks. Note that the sharing will not add technical difficulties as long as their domains are finite. For simplicity, we will not consider sharing in this paper. The only requirement on the completion of a task is given by the deadline. The time when a task is finished does not affect the current behavior specified in the automaton.

Fig. 2. Modeling Periodic and Sporadic Tasks.

Parallel Composition. To handle concurrency and synchronization, a parallel composition of extended timed automata may be defined as follows.

In the same way as for ordinary timed automata (e.g., see [10]), note that the parallel composition here is only to operators to construct models of systems based on their components. It is nothing to do with parallel processors. A product automaton may be scheduled to run on a one- or multi-processor system.

Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of running tasks with highest priority (or earliest deadlines) and firing the other tasks waiting to run. Durate transitions correspond to the arrival of a new task instance.

We represent the values of clocks as functions, i.e., clock assignment C to the non-negative real field R≥0. We denote by V the set of clock assignment C. Naturally, a semantic state of an automaton is a tuple (L, c, v) where L is the current location, v ∈ V denotes the current values of clocks, and c is the current clock value. We assume that the task queue takes the form [P1, v1(1), P2, v2(1), ..., Pk, vk(1)] where Pk, v(1) denotes a released instance of the task type Pk with remaining computing time c and relative deadline d.

Assume that there is a fixed number of processors running the released task instances according to a certain scheduling strategy, e.g., PTS (fixed priority scheduling) or EDF (Earliest Deadline First), which sorts the task queue whenever new tasks arrive according to task parameters, e.g., deadlines. An active task
Fault Diagnosis for Timed Automata

Stavros Tripakis
VERIMAG (www-verimag.imag.fr)

1 Introduction

In this paper we study the problem of fault diagnosis in the context of timed automata. Our work is inspired from [JL95, SS96], who have studied the problem in the context of discrete event systems (DES) [SS96].

We stick to the terminology used in the above papers, although we find the term fault detection, rather than diagnosis, more appropriate. Indeed the objective is to design an observer for a given plant, such that this observer can detect errors in the behavior of the plant. The method is model-based: it is assumed that the plant belongs to a known model. The observer will be based on this model. Essentially, the observer will learn all possible events that the plant can be, given the current sequence of observations.

More details follow.

In the DES framework, the fault diagnosis problem is as follows. We are given the description of the behavior of a plant, in the form of a finite state automaton. A behavior of the plant corresponds to a run of the automaton that is a sequence of events. An event is either observable or unobservable. One or more special unobservable events model faults that may occur during the operation of the plant. The objective is to design a generator.

The generator is just a function which takes sequences of observable events and decides whether the model behavior continued a fault or not. The diagnosis scheme is to announce a fault at the first step after the fault occurred. Once a fault is announced, the generator stops announcing it (i.e., on-line fault repairs are not modeled).

Not every plant is diagnosable. For example, a plant with two behaviors, $a, b$ and $a, a, b, a$, is not diagnosable. If $a, b$ is unobservable and $a$ is the fault, then the diagnosis cannot observe whether the plant is in state $a$ or in state $a, b$.

Our method has been to extend the above framework to time-automata [SS96]. Such an extension is needed, since it permits us to model plants with timed behavior. For example, "a followed by $b$ within a delay of $t$ time units". It also allows for diagnosers to track events not only as the sequences of events observed, but also on the times delays between these events. That is, the diagnosis does not only observe events, but can also measure the time elapsed between two consecutive events and, consequently, between any two events.

For example, consider the plant modeled by the timed automaton of Figure 1. The plant has two sets of behaviors, faulty behaviors (where $c_0, c_1$ occur) and non-faulty behaviors. Events $a$ and $b$ are observable, the behavior is diagnosable. Indeed, in all behaviors, $a$ and $b$ occur, in that order. In every faulty behavior, the delay between $a$ and $b$ is greater than 3 time units, whereas in every non-faulty behavior, the delay is at most 3. Thus, a diagnosis observing $a$ and $b$, and measuring their inter-event delay can tell whether a fault occurred or not.

The contributions of this paper are as follows. First, we propose a notion of diagnosability for timed automata and give necessary and sufficient conditions for a timed automaton to be diagnosable. Second, we show how diagnosability can be algorithmically checked, by reducing the problem to checking whether a certain timed automaton has a non-accepting. Finally, we show that diagnosability is in PSPACE. Third, we show how to effectively build a diagnoser for a diagnosable timed automaton. Although the size of the observer is infinite (and in fact, non-enumerable), the diagnoser function is computable, using standard techniques used in timed automata verification tools.
unknown. Each transition $e = (q, q', a, c, X) \in E$: for a special function on the structure of $A$, if $e = (q, q, a, c, X) \in E$, then $e = (q, q', a, c, X) \in E$. A timed sequence over a set of events $E$ is a finite or infinite sequence of events in $E$ with at most one delay in event $e$. We observe that between any two events in $E$, there is exactly one delay (possibly 0). For example, if $a$ and $b$ are events, $a, b, c, d, e$ and $1, 1$, we validate timed sequences, where $a$ and $b$ and $c$ and $d$ are not.

If $p$ is a finite timed sequence, then $\text{time}(p)$ denotes the sum of all delays in $p$. If $p$ is infinite, then $\text{time}(p)$ denotes the limit of the sum (possibly $\infty$). We note that a non-zero timed sequence is necessarily infinite, although it might contain only a finite number of events.

We define a projection operator $f$ as follows. Given a finite or infinite timed sequence $p$ and a set of events $\mathcal{E}$, $f(p, \mathcal{E}) = \{e \in p : e \in \mathcal{E}\}$, where $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$, $e_i \neq e_j$ for $i \neq j$, and $e_i \in p$ for each $i = 1, 2, \ldots, n$. We define $\text{time}(p, \mathcal{E}) = \text{time}(f(p, \mathcal{E}))$. Note that, in the case of $f(p, \mathcal{E})$, the sum of all delays is not necessarily finite. We observe, however, that for each event $e_i$, $\text{time}(e_i) = \text{time}(e_{i+1})$. Given a timed sequence $p$, we define a sequence $p_{\mathcal{E}}$ by setting $e_i \in p_{\mathcal{E}}$ if and only if $e_i \in p$ and $e_i \in \mathcal{E}$. We observe that a timed sequence $p$ is a sequence of events in $E$ and, as such, is an event sequence.

Definition 1 (Faulty and faulty runs) A run $p = \gamma_1, \gamma_2, \ldots$ is called faulty if for some $i = 1, 2, \ldots$, $\gamma_i = f$; let $j$ be the smallest such that $\gamma_i = f$; then $p = \gamma_1, \gamma_2, \ldots, \gamma_{j-1}$ is a faulty run of $p$. Assume that a system is fault-free if it does not contain any faulty runs.

Definition 2 (Finite and infinite runs) A run $p = \gamma_1, \gamma_2, \ldots$ is called finite if for some $i = 1, 2, \ldots$, $\gamma_i = f$; let $j$ be the smallest such that $\gamma_i = f$; then $p = \gamma_1, \gamma_2, \ldots, \gamma_{j-1}$ is a faulty run of $p$. Assume that a system is fault-free if it does not contain any faulty runs.

The following lemmas state an important property of the model, which will be used in the sequel.

Lemma 3 For all $e \in E$, if $A$ is a delta-faulty run, then $A$ is a faulty run.

Proof. Our proof relies on the finite graph $G = (V, E)$ of $A$, where $V$ is the set of states and $E$ is the set of edges.

3 Diagnosers and diagnosability

In this section, we define diagnosability as the existence of a diagnoser for a given plant. Informally, a diagnoser is a function that, given an observation, detects whether the observed behavior of the plant was faulty or not. We illustrate in the introduction that plants are diagnosable. We give necessary and sufficient conditions for diagnosability. Intuitively, a plant is diagnosable if any pair of faulty/faulty behavior of events can be distinguished by their projections in observables behaviors.

Let $\mathcal{E}$ be the set of all finite timed sequences over $E$.

Definition 3 (Diagnosers) Given a TM $A$ over $\mathcal{E}$ with sets of observable/observable events $\Sigma_1, \Sigma_2 \subseteq \mathcal{E}$, and $\Delta \in \mathcal{N}$, a $\Delta$-diagnoser for $A$ is a function $D : \mathcal{E} \rightarrow \{0, 1, \ldots, \}$

such that

1. For any non-faulty finite run $p$ of $A$, $D(p, \mathcal{E}) = 0$.
2. For any delta-faulty finite run $p$ of $A$, $D(p, \mathcal{E}) = 1$. 

If a $\Delta$-diagnoser exists in $\Delta$ then we say that $A$ is $\Delta$-diagnosable. We say that a plant is diagnosable if there exists some $\Delta \subseteq \mathcal{N}$ such that $A$ is $\Delta$-diagnosable.

Definition 4 (Necessary and sufficient conditions for $\Delta$-diagnosability) Let $A$ be a TM over $\mathcal{E}$ with sets of observable/observable events $\Sigma_1, \Sigma_2 \subseteq \mathcal{E}$, and $\Delta \subseteq \mathcal{N}$. $A$ is $\Delta$-diagnosable if for any two finite runs $p, q$ of $A$, if $p \neq q$, then $D(p, \mathcal{E}) = D(q, \mathcal{E})$.

Proof. Assume $A$ is $\Delta$-diagnosable and $D$ is a diagnoser for $A$. Suppose $p, q$ are faulty runs. Then, $D(p, \mathcal{E}) = 1$. Since $D$ is a function, it must be that $D(p, \mathcal{E}) = D(q, \mathcal{E})$. In the opposite direction, assume the conditions holds. We define function $G$ as follows:

$$G(e) = \begin{cases} 1, & \text{if there exists } r \in \mathcal{E} \text{ such that } D(p, \mathcal{E}) = 1 \text{ and } \gamma_i = e, \\ 0, & \text{otherwise}. \end{cases}$$

Then, by definition, if $p$ is a faulty finite run of $A$, $D(p, \mathcal{E}) = 1$. Now, suppose $p$ is a non-faulty finite run of $A$ and let $\pi = D(p, \mathcal{E})$. Then there exists some faulty finite run $\rho$ of $A$ such that $\pi = D(\rho, \mathcal{E})$. But this would contradict the condition. Thus, $D(\pi) = 0$.

Example 1 Assume that events $a$ and $b$ are observable and $b$ is a non-faulty event. If $b = 0$, then there exists some faulty finite run of $A$ such that $\pi = D(\rho, \mathcal{E})$. But this would contradict the condition. Thus, $D(\pi) = 0$.

Figure 2. A non-diagnosable timed automaton.

We now come to the main result of this section: the diagnosability of a given plant.

Figure 2. A non-diagnosable timed automaton.
Verification of distributed systems by timed automata
Author: Jan Krákora and Zdeněk Hanzálek (CTU)

Goals:
The goal of the part of component WP7 is design tool for verification of real time distributed
discrete event system focusing on CAN model by timed automata and specification of verified
properties by temporal logic. The crucial problem is to verify the time properties and logic
properties of complex applications (e.g. CSEK RTOS, CAN).

Description:
Let us assume the distributed real time control system consisting of application processes (designed by application developer)
running under Real-Time Operating System while using several processors interconnected via standard broadcast communication
based on the Controller Area Network (CAN). Structure of the application is depicted in figure below.

The crucial problem is to verify both,

• time properties (e.g. message response time, response time) and
• logic properties of the applications (e.g. deadlock, mutual exclusion)

Classical approaches deal separately either with the processor sharing or with the
bus sharing. The task schedulability on monoprocessor and multiprocessor systems
is widely studied subject (e.g. Rate Monotonic Scheduling (RMS)).

Prediction of the worst-case message latencies for CAN was presented by Tindell and
Burns. In similar way, CAN operates the fixed priority scheduling algorithm and authors
assume the rate monotonic priority assignment. The message worst case response time is influenced not only by its
length but also by the maximal length of one lower priority message since a high priority message cannot interrupt the message that is already
transmitted. Moreover due to the priority based bus arbitration method the message worst case response time is influenced by all
higher priority messages, each of them considering their occurrence ratio.

\[ R_m \leq J_m + \sum_{i \in \text{Proj}(m)} \lambda_i |C_i| \]

This approach is based on model checking while using timed automata and
temporal logics. Using this approach we model parts of the distributed system (application
SW, operating system and communication bus) by automata. The automata use
synchronization primitives enabling their interconnection.

The component part also incorporate priority based pre-emptive and non-pre-emptive
scheduling, inter-task communication primitives and interrupt handling. That is
why the operating system part is neglected and one application processes per one
processor is assumed in this paper.

Verification of the CAN model developed here can be directly compared to the results
in Tindell’s & Burns approach and it can be simply enlarged.

Demo phases – Case study
We assume the configuration where four processors are running one application process (per each processor) and transmit
one type of message via CAN. This system is modelled and checked by temporal logic. Examples of verification formulae is
shown below:

Is the system deadlock free? \( \rightarrow \) A \( \neg \) (not deadlock)

Does the processor with the highest priority message ever lose the bus access? \( \rightarrow \) (transceiver1_request_denied)

What is the worst-case response time \( R_m \) of the message? \( \rightarrow \) A (\( R_m \text{ finished} \)) (\( R_m \text{ response_time} \leq R_m \))

Conclusions and results:
This part of component WP7 shows design stage as an approach of communication protocol modelling by timed automata. Model of
entire distributed application can be obtained simply by interconnection with real-time operating system and application software
automata. The resulting model is suitable for verification of desired/undesired states in the time critical distributed applications like
automotive IT. The full study has been accepted to the INCOM 2004 conference. Example of a complex ABS case study is
mentioned in FISITA 2004 automotive conference paper.