Kronos: A Verification Tool for Real-Time Systems. (Kronos User's Manual Release 2.2) *

Sergio Yovine †

Abstract

The main purpose of this paper is to explain *how* to use KRONOS, a tool for formally checking whether a real-time system meets its requirements. KRONOS is founded on the theory of *timed automata* and *timed temporal logics*.

1 Introduction

KRONOS [9, 10, 8, 11, 26, 20, 6, 5] is a software tool built with the aim of assisting designers of real-time systems to verify whether their designs meet the specified requirements. One major objective of KRONOS is to provide a verification engine to be integrated into design environments for real-time systems in a wide range of application areas. Real-time communication protocols [9, 10], timed asynchronous circuits [20, 6], and hybrid systems [23, 10] are some examples of application domains where KRONOS has already been used.

KRONOS has been applied to analyze real-time systems modeled in several process description formalisms such as ATP [22], AORTA [7], ET-LOTOS [9], and T-ARGOS [19]. Several projects are currently under development to connect KRONOS to specification languages such as SHIFT [1], DIADEM [12], and TCES (*Timed Condition/Event Systems*) [17].

1.1 Formalisms

The mathematical foundations of KRONOS come from the theory of *timed automata* and *timed temporal logics*. Timed automata [3] are finite-state machines equipped with a set of variables, called *clocks*, that measure the time elapsed between *events*. A timed automaton models the behavior of a single *process* or *component* of the system. The *locations* of the automaton correspond to the

*A version of this paper will appear in Springer International Journal of Software Tools for Technology Transfer, Vol. 1, Nber. 1/2, October 1997.

†Partially supported by European Contract KIT 139 HYBSYS.

different control points of the process. A *transition* from one location to another corresponds to the execution of a *statement*. *Timing constraints* such as propagation delays, execution times and response times, are expressed as predicates on the values of the clocks.

A complex real-time system made up of m cooperating and communicating processes is modeled as the parallel composition of the corresponding m timed automata. To model inter-process *communication*, the transitions of the timed automata are labeled with sets of identifiers interpreted as *synchronization* channels.

KRONOS provides a specification framework that integrates both *logical* and *behavioral* approaches to verification. Correctness criteria can be specified in two different ways: as *formulas* of the timed temporal logic TCTL [2, 16] (logical or model-checking approach) or as timed automata (behavioral approach).

For the logical approach, KRONOS implements model-checking algorithms that check whether the system satisfies a property given as a formula of TCTL. These algorithms can be classified in two general categories according to the strategy applied to explore the state-space: *backward analysis* and *forward analysis*. Backward analysis algorithms perform a backward search of the reachability graph to compute the set of predecessors of a given set of states. Forward analysis algorithms construct the set of successors by performing a forward exploration of the graph[8].

For the behavioral approach, KRONOS provides an algorithm that constructs an automaton where time has been *abstracted away* in such a way that the causality relationship between events is preserved [26]. Checking whether two timed automata are *equivalent* amounts to checking whether the time-abstracted automata constructed by KRONOS are *bisimilar* [21]. In this paper we use the tool ALDEBARAN [14, 15] to check for bisimilarity.

1.2 Overview of the paper

The main purpose of this paper is to explain *how* to use the tool KRONOS. The underlying theories, methods and algorithms, as well as the experimental results obtained so far, can be found in [22, 16, 9, 10, 8, 11, 26, 20, 6, 5].

This paper is structured as follows. All concepts and formalisms are first *informally* presented. Formal definitions are given in the appendix. Throughout the paper we use a simple case study, namely the CSMA/CD protocol [25, 18] to illustrate both the formalisms and the verification approaches.

In section 2 we introduce timed automata, and we use them in section 3 to model the CSMA/CD protocol. In section 4 we present the logical and behavioral approaches supported in KRONOS. Section 5 overviews some features and options of KRONOS. Section 6 is devoted to the verification of several properties of the protocol.

2 Model: syntax and informal semantics

In this section, we introduce the syntax for describing timed automata and informally explain its meaning. Precise definitions can be found in A.

2.1 Syntax

In KRONOS, the *timed automaton* specifying the behavior of a component C is described in a file, say C.tg. A timed automaton is a decorated finite graph consisting of the following features.

Locations. The nodes of the graph are called *locations*. They represent the control points or discrete modes of the system. In KRONOS, each location is identified by the statement loc:n where n is a natural number in the interval $[0, \mathtt{N}-1]$, and N is the number of locations, defined by the statement #locs N.

Clocks. A component is equipped with a finite set of real-valued variables that measure time. These variables are called *clocks*. We assume that each automaton of the system is equipped with its own set of clocks, that is, clocks are *private* variables of each component. The set of clocks is defined by the statement #clocks <id> ... <id> where <id> is a string (an *identifier*).

Synchronization events. A component interacts with the other components of the system by synchronizing over a set of *events*. The set of *synchronization* events is defined by the statement #sync <id> ... <id>.

Transitions. To move from one location to another the automaton executes a *transition*. A transition is a labeled arc of the graph. The number T of transitions is defined by the statement #trans T. For each control location, the set of outgoing transitions is defined by the keyword trans: followed by a list of definitions of the form

<g> => <id> ... <id> ; <r>, ... , <r> ; goto n.

<g> is called the *guard* of the transition. Guards are used to model the timing conditions that constrain the execution of transitions. A guard is a predicate over the clocks. In a guard, the value of a clock and the difference between the values of two clocks can be compared to integer constants. More precisely, a guard is a conjunction of atoms of the form <id$_1$> # c, or <id$_1$>-<id$_2$> # c, where <id$_1$> and <id$_2$> are clocks, # is a symbol in the set $\{<, \leq, >, \geq, =\}$, and c is an integer constant.

<id> ... <id> is the set of events associated with the transition. If some of these events are synchronization events, the transition is called a *synchronizing* transition. Otherwise, it is called an *internal* transition.

<r>, ... , <r> is called the *assignment* of the transition. An assignment changes the value of another clock, either by setting it to 0 or to the current

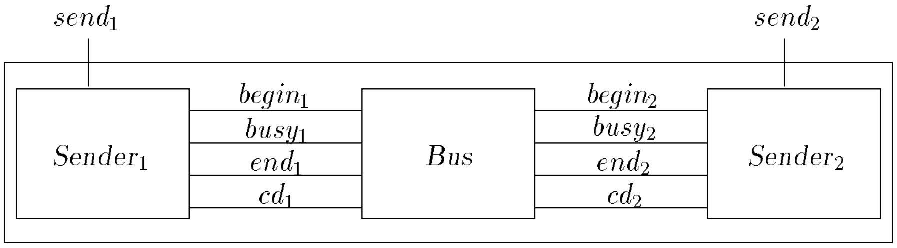

Figure 1: Structure of the system.

value of a clock. Syntactically, each <r> is of the form <id$_1$> := 0, or <id$_1$> := <id$_2$> , where <id$_1$> and <id$_2$> are clocks.

Location invariant. Each control location has to be labeled with a predicate over the values of the clocks. This predicate, called *location invariant*, is defined by the statement `invar: <g>` where <g> is as above.

Boolean propositions. Each control location can be labeled with a set of identifiers. These identifiers represent the *boolean variables* whose value is *true* at the location. The labeling function is defined by the statement `prop: <id> ... <id>`. The identifier `init` is a keyword used to distinguish the initial control location of the component. More than one control locations can be labeled `init`. In this case, the component non-deterministically starts in one of these locations. If no location is labeled `init`, by default, `loc:0` is the initial control location.[1]

2.2 Semantics

Informally, the behavior of a timed automaton is as follows.

States. At any time, the *state* of a component is determined by the location and the values of the clocks. States have to satisfy the location invariant.

Evolutions. The state of a component evolves as follows. The component may change its control location and the values of some of the clocks by executing an enabled transition, or it may decide to let some amount of time pass while it remains in the same location without violating the location invariant.

Time passing. The invariant must hold when the location is entered and during all the time the component stays in it. When the location invariant is *true*, the component *may* remain at the location forever without ever taking a transition. Otherwise, the component *must* take a transition *before*

[1] Notice that propositions can also be used to associate *names* with locations.

the location invariant is violated. That is, location invariants are used to impose upper bounds on the sojourn time of a component in a location. In other words, they allow to model *urgency*.

Execution of transitions. A transition is said to be *enabled* in a state, that is, it is ready to be taken, if the values of the clocks satisfy the guard. When a transition is taken, the values of the clocks are modified according to the associated assignment. The execution of a transition is assumed to be *atomic* and *instantaneous*.

2.3 Behavior of the global system

In KRONOS, a system is specified as a set of files, say `C_1.tg, ..., C_m.tg`, each one specifying the behavior of a component. The behavior of the global system is as follows.

Global states. A global control location is an m-tuple of locations of the components. At any time, the global state of the system is determined by a global control location and the values of the clocks of all the components at that time.

Global evolutions. The system may change from a global state to another either by letting time pass or by executing a transition.

Time passing. In order to let some time pass, *all* the components *must* agree in the amount of time that can elapse. This means that the location invariants of all the components must be respected.

Execution of transitions. The system changes its global location if one of the components individually executes an enabled internal transition or if a subset of components execute enabled synchronizing transitions. A set of synchronizing transitions can be executed simultaneously if every synchronization event name of a transition in the set belongs to at least another different transition also in the set. The set of synchronizing transitions is *maximal*. Maximality means that non-participating components are not willing to synchronize, that is, they do not have transitions labeled with some synchronization event name appearing in the set. In other words, if a synchronization event name e appears in a global transition, *all* the automata declaring e as synchronization event must be participating in the transition.

Product automaton. The behavior of the global system can also be described by a single timed automaton, called the *product automaton*. Informally, this automaton is as follows.

- The locations are m-tuples of locations.

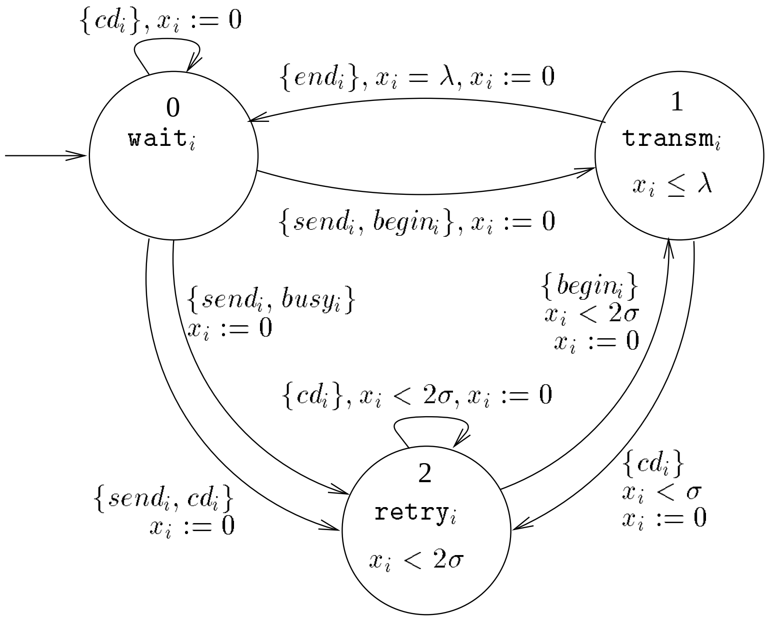

Figure 2: $Sender_i$.

- The set of clocks is the union of the set of clocks of the components.
- The set propositions associated with a global location is the union of the sets of propositions of the corresponding locations.
- The global location invariant is the conjunction of the invariants of the components.
- The guard of a global transition is the conjunction of the guards of the participating transitions.
- The assignment of a global transition is the union of the assignments of the participating transitions.[2]

The precise set of rules for constructing the product automaton are given in Appendix A.

3 Modeling the CSMA/CD protocol

In any broadcast network with a single bus, which is sometimes referred to as a *multi-access bus*, the key issue is how to assign the use of the bus when many stations compete for it. Several protocols are known to solve this problem. One

[2]Recall that clocks are private to processes.

of these protocols is the *Carrier Sense, Multiple Access with Collision Detection*, or CSMA/CD for short [25, 18].

A very abstract and simplified description of the protocol is the following. When a station has data to send, it first listens to the bus. If it is idle (i.e., no other station is transmitting), the station begins sending its message. However, if it detects a busy bus, it waits a random amount of time and then repeats the operation. When a collision occurs, because several stations transmit simultaneously, then all of them detect it, abort their transmissions immediately and wait a random time to start all over again. If two messages collide then they are both lost.

We formally model here the CSMA/CD protocol for two sender processes located in two stations linked by a single bidirectional bus. We suppose that the bus is a 10Mbps Ethernet with a worst case propagation delay σ of 26μs. Messages have a fixed length of 1024 bytes, and so the time λ to send a complete message, including the propagation delay, is 808μs. For simplicity, we make the following assumptions: the bus is error-free, only the transmission of messages is modeled, and no buffering of incoming messages is allowed.

3.1 Structure of the system

The structure of the system is depicted in Figure 1. Each box represents a component of the system. The lines represent the synchronizations between the components. $Sender_i$ and Bus synchronize through the following *events*:

- $begin_i$ — $Sender_i$ starts sending the message,
- $busy_i$ — the bus is sensed busy by $Sender_i$,
- end_i — $Sender_i$ completes the transmission,
- cd_i — a collision is detected by $Sender_i$.

These events are the synchronization events. Besides, we will use the events $send_1$ and $send_2$ to model the reception of messages from the upper layer.

3.2 Timed automata of the senders

The behavior of process $Sender_i$ is depicted in Figure 2. A sender can be in one of the following control locations:

- $\texttt{wait}_i$ — waiting for a message from the upper layer,
- $\texttt{transm}_i$ — sending a message,
- $\texttt{retry}_i$ — waiting to retry after detecting a collision or a busy bus.

Locations are graphically represented as circles labeled with the *name* of the location and its invariant. Transition are represented as directed arcs between the *source* and the *target* locations. Each transition is labeled with a *set of*

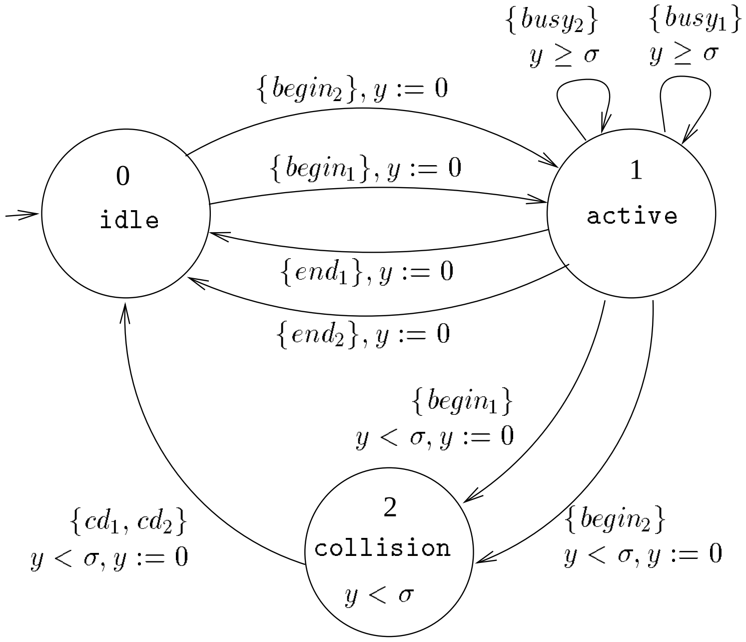

Figure 3: Bus.

events, a *guard* and an *assignment* of the clocks. Guards and invariants are omitted if they are *true*.

Initially, the sender is in location $\mathtt{wait}_i$. In the figure, the initial location is the one pointed into by the small arrow. When a message arrives, one of the following transitions is executed. If the bus is not busy, the sender starts the transmission. Otherwise, if the bus is busy because another sender is already transmitting, or because of a collision, the sender waits to re-attempt. Besides, if a collision is detected while there is no message to send, the sender remains in $\mathtt{wait}_i$. The location invariant of $\mathtt{wait}_i$ and the guards of all the outgoing transitions are *true*. This is because a message from the upper layer may arrive at any time.

In location $\mathtt{transm}_i$, the sender stays at most λ. If a collision is detected before σ, the sender goes to $\mathtt{retry}_i$. Otherwise, it terminates sending the message after exactly λ time units. In location $\mathtt{retry}_i$, the sender non-deterministically makes a new attempt to send before 2σ elapsed since the last attempt.

The textual description of the timed automaton of the sender process in the KRONOS input language is given in Figure 4.

3.3 Timed automaton of the bus

The behavior of the bus is depicted in Figure 3. The bus can be in the following three control points:

- `idle` — no station is transmitting,
- `active` — some station has started a transmission,
- `collision` — the burst noise of a collision is being propagated.

Initially, the bus is in location `idle`. When one of the stations starts sending its message, the bus moves to location `active`. After being in `active` for at least σ, the bus responds $busy_i$ to new attempts to send a message, modeling the fact that the head of the message currently being send has already propagated. If the other station begins sending before σ, the bus moves to the location `collision` where it takes a time less than σ to propagate the collision to all the senders. By labeling the transition from `collision` to `idle` with the set of events $\{cd_1, cd_2\}$, we make both senders to synchronize with the bus at the same time. This means that the collision will be simultaneously detected by all the senders. This assumption has been made in order to keep the presentation short and simple.

The textual description of the timed automaton of the bus in the Kronos input language is given in Figure 4.

4 Verification: basic notions and terminology

4.1 Logical approach

Given a system modeled as a network of synchronizing timed automata, we want to verify whether it satisfies the correctness criteria specified as formulas of the timed temporal logic called Tctl. The formal syntax and semantics of the logic are given in appendix B. Here we focus on some of the classes of properties that can be specified in the logic.

Reachability. Many interesting properties can be stated in terms of the *reachability* of a given set of states. An example are *safety* properties: a system satisfies the safety criteria if the system *never* enters into a state belonging to the set of *unsafe* states.

In Tctl, a safety property is expressed as follows:

$$init \Rightarrow \neg\exists\Diamond unsafe$$

where *init* is the predicate characterizing the set of initial states, and the symbol "$\exists\Diamond$" means *some state along some execution.*

Another equivalent way of expressing the same property is the following:

$$init \Rightarrow \forall\Box safe$$

```
/* Sender 1 */
#locs 3
#trans 9
#clocks X1
#sync BEGIN1 END1 BUSY1 CD1
loc: 0
prop: WAIT1
invar: TRUE
trans:
TRUE => SEND1 BEGIN1; X1:=0; goto 1
TRUE => SEND1 BUSY1; X1:=0; goto 2
TRUE => SEND1 CD1; X1:=0; goto 2
TRUE => CD1; X1:=0; goto 0
loc: 1
prop: TRANSM1
invar: X1<=808
trans:
X1=808 => END1; X1:=0; goto 0
X1<26 => CD1; X1:=0; goto 2
loc: 2
prop: RETRY1
invar: X1<=52
trans:
X1<=52 => BEGIN1; X1:=0; goto 1
X1<=52 => BUSY1; X1:=0; goto 2
X1<=52 => CD1; X1:=0; goto 2
```

```
/* Bus */
#locs 3
#trans 9
#clocks Y
#sync BEGIN1 BEGIN2 END1 END2
      BUSY1 BUSY2 CD1 CD2
loc: 0
prop: IDLE
invar: TRUE
trans:
TRUE => BEGIN1; Y:=0; goto 1
TRUE => BEGIN2; Y:=0; goto 1
loc: 1
prop: ACTIVE
invar: TRUE
trans:
Y<26 => BEGIN1; Y:=0; goto 2
Y>=26 => BUSY1; ; goto 1
TRUE => END1; Y:=0; goto 0
Y<26 => BEGIN2; Y:=0; goto 2
Y>=26 => BUSY2; ; goto 1
TRUE => END2; Y:=0; goto 0
loc: 2
prop: COLLISION
invar: Y<26
trans:
Y<26 => CD1 CD2; ; goto 0
```

Figure 4: KRONOS input files.

where the symbol "$\forall\Box$" means *every state along every execution*. In words, the above formula says that all the states reachable from the set of initial states are *safe*.

Non-Zenoness. One important property that has to be shown concerns the *divergence* of time. A *good* timed model *must* allow time to elapse without bound. Executions along which time converges are called *Zeno*. A state is called *Non-Zeno* if whatever the system does at the state, it does not prevent time to diverge. Detecting zeno behaviors is very important: it might be the case that the system meets the safety criteria by just stopping the flow of time. Clearly, such behaviors are not acceptable.

It has been shown in [16] that all the states reachable from the set of initial states are Non-Zeno if and only if the following formula evaluates to true:

$$init \Rightarrow \forall\Box\exists\Diamond_{=1} true$$

The subscript "$= 1$" means that we are only interested in states that are reachable in a time equal to one. In other words, the above formula states that *every state reachable from init can let time pass one time unit.*

Bounded response. One important property of real-time systems is that they have to respond in a bounded time to requests issued by their environment. For instance, a typical requirement is the following: every request from a client has to be served or rejected in at most 5 time units. This property is specified in TCTL as follows:

$$request \Rightarrow \forall\Diamond_{\leq 5}(served \vee rejected)$$

where *request*, *served* and *rejected* are predicates that characterize the set of states corresponding to the reception, the acceptance and the rejection of a request, respectively. The symbol "$\forall\Diamond$" means *some state along every execution*. The subscript "≤ 5" means that we are only interested in states that are reachable in a time less than or equal to 5.

4.2 Behavioral approach

Behavioral equivalences have proven useful for verifying the correctness of concurrent systems. They provide means for comparing the behavior of two systems both represented as labeled graphs. The theoretical foundations of the behavioral approach can be found in [21].

We illustrate here the basic ideas with a simple example. Suppose, for instance, that we are implementing a client process that sends a request message to the server and keeps doing so until its request is granted. We would like our client process to behave as shown in Figure 5.

The client process, however, is likely to be more complex due to implementation details such as loss of messages, timeouts, propagation delays and so forth.

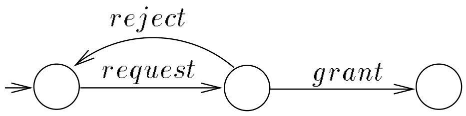

Figure 5: Abstract behavior of a client process.

Now, in order to be able to compare both systems, we must abstract away of such details.

The transitions corresponding to implementation issues can be considered to be *non-observable* for the purposes of the comparison. In the literature, non-observable transitions are traditionally labeled with the special event τ. Renaming the events of a transition by τ is called *hiding*.

Timing is an important factor we have to deal with when comparing the client process with its intended abstract behavior. Indeed, there is no mention to timing at all in the latter. One way of dealing with time consists in abstracting away of the *exact amount* of time elapsed. Here, propagation delays, idle times, and so on, cannot be precisely measured: a delay between two events appears as a τ-transition between them. However, the causality relationship between the events and their temporal ordering is preserved.

The abstraction criterion used for time leads to the definition of the so-called *time-abstracting* equivalence presented in Appendix C. This equivalence provides formal means to compare a real-time implementation of a system with a more abstract and untimed specification of it. However, it can also be used to compare the behavior of two timed processes.

Once the implementation details have been hidden, the processes can be compared with respect to many different equivalences. These equivalences differ on the way τ-transitions are treated. Several tools for verifying whether two processes are behaviorally equivalent have been developed. One such tool is Aldebaran [15].

5 Overview of Kronos

We briefly present in this section some of the options offered by the tool Kronos. We will use them in the next section to verify the CSMA/CD protocol specified in section 3.

Product automaton. Recall that, in Kronos, a system is specified as a set of files, say `C_1.tg, ..., C_m.tg`, each one specifying the behavior of a component. The global behavior of the system is formally defined by the product automaton, and therefore, checking whether the system satisfies a property amounts to verify whether the product automaton does satisfy it. When executing the command:

```
kronos -out S.tg C_1.tg ...  C_m.tg
```

KRONOS constructs the product automaton of the system and writes it down into the file `S.tg`. The file `S.tg` can then be used as input to KRONOS for the purpose of verification. However, for efficiency reasons, i.e., to avoid the so-called *state-explosion problem*,[3] the product automaton need not be constructed explicitly before verifying a property. KRONOS is able to construct the product automaton on-the-fly, that is, while performing the verification. In the current version,[4] this is only possible in the case of reachability properties. We are currently implementing an algorithm that allows to do on-the-fly verification for full TCTL. This algorithm, along with some experimental results, is described in [5]. In what follows, unless otherwise stated, verification is performed on the product automaton `S.tg`.

Backward analysis. Let `f` be a formula of the logic TCTL and `S` be a timed automaton. To check whether `S` satisfies `f` we can use an algorithm based on the exploration of the state-space of `S` by a *backward* traversal of its transitions. This method is called *backward analysis*. Let `f.tctl` be the file containing the formula `f`. When executing the command:

```
kronos -back S.tg f.tctl
```

KRONOS outputs the result of the verification in the file `f.eval`. This file contains a symbolic characterization of the (possibly infinite) set of states of the state-space of `S` that satisfy `f`.

Forward analysis. An alternative way of checking whether `S` satisfies `f` consists in exploring the state-space of `S` by a *forward* traversal of the transitions of `S`. This method is called *forward analysis*. In the current version, the forward analysis algorithm is only implemented for a subset of TCTL. When executing the command:

```
kronos -forw S.tg f.tctl
```

KRONOS outputs either `TRUE` or `FALSE`. In the second case, it also outputs the file `_path.reach` containing a sequence of transitions leading to a state not satisfying `f`. By default, KRONOS performs the forward analysis using a breadth-first strategy. A depth-first search algorithm can be invoked by executing KRONOS with the option `-DFS`.

Reachability. To know whether some state satisfying the formula `f` is reachable from some initial state, we can execute the command:

```
kronos -reach f.tctl S.tg
```

[3] The number of control locations of the product automaton is an exponential function of the number of components.

[4] Version 2.2

KRONOS will first invoke the backward analysis algorithm to evaluate `f` and then it will perform a forward exploration to check for reachability. By default, KRONOS will start the forward search from the control locations of `S` labeled with the keyword `init`, or from location `loc:0` if no location is labeled `init`. The default set of initial states can be changed by executing the command:

```
kronos -init i.tctl -reach f.tctl S.tg
```

where `i.tctl` describes the desired set of initial states. If the formulas specified in the files `i.tctl` and `f.tctl` are boolean combinations of atomic propositions and predicates on clocks, KRONOS can construct the product automaton on-the-fly. In order to use the on-the-fly verification algorithm, we should provide the files `C_1.tg, ... , C_m.tg` as input instead of `S.tg`.

Time-abstracting equivalence. Let `S.tg` be a timed automaton. When executing the command:

```
kronos -ta S.tg
```

KRONOS constructs a graph where time has been abstracted away as explained in section 4. To do so, KRONOS implements the algorithm presented in [26]. The output is written down into the file `S.aut` using ALDEBARAN's input format. If `S.tg` is to be compared to another timed automaton `P.tg` with respect to the time-abstracting equivalence, we execute KRONOS to construct the file `P.aut`. Checking whether `S.tg` and `P.tg` are equivalent modulo the time-abstracting equivalence amounts to checking whether `S.aut` and `P.aut` are equivalent modulo the so-called *strong* equivalence [21]. Clearly, `S.aut` and `P.aut` can also be compared with respect to other equivalences. Verifying whether `S.aut` and `P.aut` are equivalent can be done with the tool ALDEBARAN.

6 Verifying the CSMA/CD protocol

We illustrate in this section the use of the tool KRONOS to verify the system modeled in section 3. The verification of this case study is performed on the product automaton `csma.tg` which is obtained by executing the command:

```
kronos -out csma.tg bus.tg sender1.tg sender2.tg
```

where `bus.tg`, `sender1.tg`, and `sender2.tg` are the files presented in Table 4.

6.1 Non-Zenoness

We start by checking whether the system satisfy the Non-Zenoness property. In KRONOS syntax, this property is written as follows:

```
init impl ab ( ed{=1} true )
```

Let `nz.tctl` be the file containing the formula above. To check for Non-Zenoness, we execute the command:

```
kronos -v -forw csma.tg nz.tctl
```
[5]

By default, `init` characterizes the control location `loc:0` of `csma.tg` with all the clocks set to zero, that is:

$$\mathtt{wait}_1 \wedge x_1 = 0 \wedge \mathtt{wait}_2 \wedge x_2 = 0 \wedge \mathtt{idle} \wedge y = 0$$

Kronos finds that the property is indeed *not* verified by the system and outputs a *counter-example*, that is, an example of finite execution that violates the property. The counter-example is written down into the file `_path.reach` and looks as follows:

```
location: 0
propositions: WAIT2 WAIT1 IDLE
constraint: TRUE
transition: TRUE => SEND1 BEGIN1 ; RESET{ Y X1 } ; goto 1

location: 1
propositions: WAIT2 TRANSM1 ACTIVE
constraint: X1<=808 and Y=X1 and Y<=X2
transition: Y<26 and X1<=808 => SEND2 BEGIN2 ;
            RESET{ Y X2 } ; goto 3

location: 3
propositions: TRANSM1 TRANSM2 COLLISION
constraint: Y<=26 and Y<=X1 and X1<Y+26 and Y=X2
```

This counter-example shows that some states reachable from the initial state after executing the transition labeled $\{send_1, begin_1\}$ followed by the transition labeled $\{send_2, begin_2\}$ are zeno. How it might be? It is easy to see that the system may reach this global control location with the value of the clock x_1 being 26. Now, at this state, process $Sender_1$ *cannot* take the transition labeled cd_1 into the global control location ($\mathtt{retry}_1, \mathtt{retry}_2, \mathtt{idle}$). Therefore, it misses the synchronization with $Sender_2$ and Bus on the set of events $\{cd_1, cd_2\}$. This behavior is zeno because at that global location no transition may ever be taken while the location invariant prevents time to diverge.

Now, observe that the inequality associated with the final global control location of the counter example satisfies $x_2 = y \leq x_1 < y + 26$. This inequality expresses the fact that $Sender_2$ *started sending at most 26 time units after* $Sender_1$. Recall that, in the worst case the collision may take 26 time units to be propagated. Now, since the propagation delay is measured by the clock y, the value of x_1 may be such that $26 \leq x_1 < 52$ when the collision is detected. However, $Sender_1$ is *not* waiting for a collision to occur after x_1 becomes 26.

Thus, according to the information extracted from the counter-example, we should modify our original model such as sender i waits 52 time units instead of 26 to prevent taking the transition cd_i. If we re-evaluate the Non-Zenoness property on this new model we obtain that the system is now well-timed.

[5] `-v` is the verbose option.

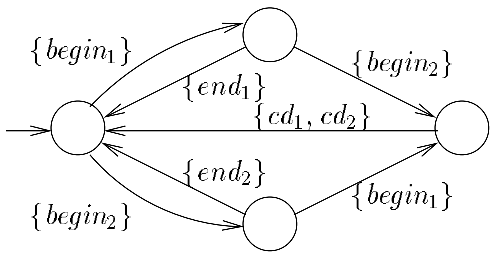

Figure 6: Abstract specification of the CSMA/CD.

6.2 Transmission and collision detection

A very abstract description of the CSMA/CD protocol is the following. When a station starts sending a message, it either terminates normally or it aborts the transmission when a collision occurs. In the latter case, both senders detect the collision, and re-start all over again. Figure 6 illustrates this desired abstract behavior.

We would like to prove now that `csma.tg` is equivalent to the automaton depicted in Figure 6. Suppose that `csma_spec.aut` is the file containing the textual description of this automaton in ALDEBARAN input format. As explained in section 5, we start by constructing the automaton `csma.aut` where time has been abstracted away. This is done by executing the command:

```
kronos -ta csma.tg
```

The file `csma.aut` generated by KRONOS has 481 locations and 875 transitions. We then have to hide all the non-observable events, that is, we have to replace all the events not appearing in `csma_spec.aut` by τ.[6] Suppose that `csma_hide.aut` is the resulting automaton. In order to prove that `csma_hide.aut` is equivalent to `csma_spec.aut`, we use ALDEBARAN as follows:

```
aldebaran -sequ csma_hide.aut csma_spec.aut
```

The option `-sequ` corresponds to the *safety* equivalence defined in [4]. ALDEBARAN answers that the systems are indeed equivalent.

6.3 Bounded delay for collision detection

The previous result shows that a collision is detected whenever the two senders are simultaneously transmitting. However, it provides no information about the time it takes to the senders to become awared of the collision. One important property of the CSMA/CD protocol is that it must ensure a collision to be

[6] Actually, events $send_1$ and $send_2$ are simply deleted instead of being replaced by τ. This is because these events only appear in transitions also labeled with events $begin_1$ and $begin_2$.

detected at most $\sigma = 26$ time units after the senders started sending. We can express this property in TCTL as follows:

$$(\texttt{transm}_1 \wedge \texttt{transm}_2) \Rightarrow \forall\Diamond_{\leq 26}(\texttt{retry}_1 \wedge \texttt{retry}_2)$$

In KRONOS syntax:

```
TRANSM1 and TRANSM2
        impl ad{<= 26} ( RETRY1 and RETRY2 ).
```

Let `cd.tctl` be the file containing this formula. We verify this property with the *forward* analysis algorithm of KRONOS:

```
kronos -v -forw csma.tg cd.tctl
```

KRONOS answers that the system satisfies the formula.

6.4 Successful transmission

Clearly, the CSMA/CD protocol does not ensures that each message will be eventually transmitted. It might be the case that a collision occurs each time a sender tries to transmit its message. However, when one of the senders begins transmitting, there *must exist* a behavior leading to a successful transmission. We can formally state this property in TCTL as follows. For $i = 1, 2$:

$$\texttt{transm}_i \wedge x_i = 0 \wedge \neg\texttt{collision} \Rightarrow \exists\Diamond_{=26}\forall\Diamond_{=782}\texttt{wait}_i$$

This formula says that from the states satisfying $\texttt{transm}_i$, there exists an execution that in $\sigma = 26$ time units leads to a state where no collision happens and the sender will eventually terminate in $\lambda - \sigma = 782$ time units. For $i = 1$, the formula above is written in KRONOS syntax as follows:

```
TRANSM1 and X1=0 and not COLLISION
                impl ed{=26} ad{=782} WAIT1.
```

Let `transm1.tctl` be the file containing this formula. To evaluate this property with use the *backward* analysis algorithm of KRONOS:

```
kronos -v -back csma.tg transm.tctl
```

KRONOS answers that the system satisfies the formula.

7 Conclusions

We have briefly presented KRONOS, a tool for the verification of real-time systems. This tool is freely distributed through the web for academic non-profit use. To down-load the executable code (for several architectures), the (draft) user guide, an on-line reference manual, and several examples check the Kronos home page.

The main purpose of the paper has been to illustrate the use of KRONOS with a very simple case study. This example allowed us to present some of the features provided by KRONOS, such as: backward analysis, forward analysis, generation of counter-examples and time-abstracting bisimulation checking in connection with the tool ALDEBARAN.

The verification of more complex systems should require the use of other features and options provided by KRONOS. Such features, whose explanation is out of the scope of this paper, include the optimization of the number of clocks used in the model [11], the use of on-the-fly [5] and partial-order techniques [24], and binary decision diagrams [6]. All these features have an important aspect in common: they try to reduce the size of the state-space explored during the verification. They have been implemented in order to improve both the amount of memory and time required by the verification algorithms.

Acknowledgments. C. Daws, A. Olivero, F. Pagani and S. Tripakis have contributed to the development of the version of KRONOS presented in this paper.

References

[1] A. Gollu A. Deshpande and P. Varaiya. Shift: A formalism and a programming language for dynamic networks of hybrid automata. In *Proc. Hybrid Systems VI*. To appear in LNCS, Springer-Verlag, 1997.

[2] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] A. Bouajjani, J. C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *Proc. 18th ICALP*, July 1991.

[5] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model-checking for real-time systems. In *Proc. 1997 IEEE Real-Time Systems Symposium, RTSS'97*, San Francisco, USA, December 1997. IEEE Computer Society Press.

[6] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proc. 1997 Computer-Aided Verification, CAV'97*, Israel, June 1997. LNCS, Springer-Verlag.

[7] S. Bradley, D. Kendall, W. D. Henderson, and A. P. Robson. Validation, verification and implementation of timed protocols using AORTA. In *Protocol Specification, Testing and Verification XV (PSTV '95), Warsaw*, pages 193–208. IFIP, North Holland, June 1995.

[8] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, pages 208–219. LNCS 1066, Springer-Verlag, 1996.

[9] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proc. 7th. IFIP WG G.1 International Conference of Formal Description Techniques, FORTE'94*, pages 227–242, Bern, Switzerland, October 1994. Formal Description Techniques VII, Chapman & Hall.

[10] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. 1995 IEEE Real-Time Systems Symposium, RTSS'95*, Pisa, Italy, December 1995. IEEE Computer Society Press.

[11] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proc. 1996 IEEE Real-Time Systems Symposium, RTSS'96*, Washington, DC, USA, December 1996. IEEE Computer Society Press.

[12] A. Deshpande and S. Yovine. The diadem-kronos connection: Bridging the gap between implementation and verification of hybrid systems. In *Hybrid Systems V*, Notre Dame, Indiana, USA, September 1997.

[13] E. A. Emerson and E. Clarke. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Workshop on Logic of Programs*. LNCS 131, Springer-Verlag, 1981.

[14] J. C. Fernandez and L. Mounier. On the fly verification of behavioural equivalences and preorders. In *Proc. CAV'91*. LNCS 757, Springer-Verlag, 1991.

[15] J. C. Fernandez, H. Garavel, L.Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A tool box for the verification of lotos programs. In *14th Int. Conf. on Software Engineering*, 1992.

[16] T. A. Henzinger, X. Nicollin, J.Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[17] R. Huuck, Y. Lakhnech, L. Urbina, S. Engell, S. Kowalewski, and J. Preußig. Comparing timed condition/event systems and timed automata. In *Proc. HART'97*, pages 81–86, Grenoble, 1997. LNCS 1201, Springer-Verlag.

[18] IEEE. ANSI/IEEE 802.3, ISO/DIS 8802/3. IEEE Computer Society Press, 1985.

[19] M. Jourdan, F. Maraninchi, and A. Olivero. Verifying quantitative real-time properties of synchronous programs. In *Fifth Int. Workshop on Computer-Aided Verification, Elounda (Crete)*. LNCS 697, Springer Verlag, 1993.

[20] O. Maler and S. Yovine. Hardware timing verification using KRONOS. In *Proc. 7th Israeli Conference on Computer Systems and Software Engineering*, Herzliya, Israel, June 1996.

[21] R. Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[22] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE TSE Special Issue on Real-Time Systems*, 18(9):794–804, September 1992.

[23] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In *Proc. 6th Computer-Aided Verification*, pages 81–94, California, June 1994. LNCS 818, Springer-Verlag.

[24] F. Pagani. Partial orders for real-time systems. In *Proc. 4th Intl. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'96*, Uppsala, Sweden, September 1996.

[25] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, second edition, 1989.

[26] S. Tripakis and S. Yovine. Analysis of timed systems based on time–abstracting bisimulations. In *Proc. 8th Conference Computer-Aided Verification, CAV'96*, pages 232–243, Rutgers, NJ, July 1996. LNCS 1102, Springer-Verlag.

A Timed automata

In this paper we follow the definitions given in [22, 10].

A.1 Syntax

Clocks. Let $\mathcal{X}$ be a finite set of *clocks*. A *valuation* v is a function that assigns a non-negative real-value $v(x) \in \mathsf{R}^+$ to each clock $x \in \mathcal{X}$. The set of valuations is denoted $\mathcal{V}_{\mathcal{X}}$. For $\delta \in \mathsf{R}^+$, $v + \delta$ denotes the valuation v' such that $v'(x) = v(x) + \delta$ for all $x \in \mathcal{X}$.

Constraints. Let $\Psi_{\mathcal{X}}$ be the set of predicates over $\mathcal{X}$ defined as a conjunction of atoms of the form $x \# c$ or $x - y \# c$, where $x, y \in \mathcal{X}$, $\# \in \{<, \leq, >, \geq, =\}$ and c is an integer constant. For $\psi \in \Psi_{\mathcal{X}}$ we write $\mathsf{clk}(\psi)$ to denote the set of clocks that appear in ψ.

Assignments. Let $\mathcal{X}^*$ be the set $\mathcal{X} \cup \{0\}$. An *assignment* ρ is a function from $\mathcal{X}$ to $\mathcal{X}^*$. For $X \subseteq \mathcal{X}$, $\rho(X) \subseteq \mathcal{X}$ denotes the set of clocks $\{x \in \mathcal{X} \mid \exists y \in X.\ x = \rho(y)\}$. We denote $v[\rho]$ the valuation v' such that for all $x \in \mathcal{X}$, $v'(x) = v(\rho(x))$ if $\rho(x) \in \mathcal{X}$, otherwise $v'(x) = 0$.

Timed automaton. A *timed automaton* $\mathcal{M}$ is a tuple $\langle \mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{T}, \mathcal{I}, \mathcal{P} \rangle$.

1. $\mathcal{S}$ is a finite set of *locations*. We denote by s_{init} the initial location.
2. $\mathcal{X}$ is a finite set of *clocks*.
3. $\mathcal{L}$ is a finite set of *synchronization events*.
4. $\mathcal{T}$ is a finite set of *edges*.
5. Each edge t is a tuple (s, L, ψ, ρ, s') where
   (a) $s \in \mathcal{S}$ is the source,
   (b) $s' \in \mathcal{S}$ is the target,
   (c) L is a set of events,
   (d) $\psi \in \Psi_{\mathcal{X}}$ is the enabling condition, and
   (e) $\rho : \mathcal{X} \to \mathcal{X}^*$ is the assignment.
6. $\mathcal{I}$ is a function that associates a condition $\mathcal{I}(s) \in \Psi_{\mathcal{X}}$ to every control location $s \in \mathcal{S}$ called the invariant of s.
7. $\mathcal{P}$ associates with each control location a set of atomic propositions.

A.2 Semantics

State. A *state* of $\mathcal{M}$ is a pair $(s,v) \in \mathcal{S} \times \mathcal{V}_{\mathcal{X}}$ such that v satisfies $\mathcal{I}(s)$. The initial state is the pair (s_{init},v_{init}) such that $v_{init}(x) = 0$ for all $x \in \mathcal{X}$. Let $\mathcal{Q}$ denote the set of states of $\mathcal{M}$.

Transitions. At any state, $\mathcal{M}$ can evolve either by moving through an edge that changes the location and the value of some of the clocks (*discrete transition*), or by letting time pass without changing the location (*time transition*).

Discrete transitions. Let $t = (s, L, \psi, \rho, s') \in \mathcal{T}$ be an edge. The state (s, v) has a discrete transition to (s', v'), denoted $(s, v) \to_0^L (s', v')$, if v satisfies ψ and $v' = v[\rho]$.

Time transitions. Let $\delta \in \mathsf{R}^+$. The state (s, v) has a time transition to $(s, v + \delta)$, denoted $(s, v) \to_\delta^\emptyset (s, v + \delta)$, if for all $\delta' \leq \delta$, $v + \delta'$ satisfies the invariant $\mathcal{I}(s)$.

We denote $(\mathcal{Q}, \to)$ the transition system of $\mathcal{M}$.

Runs. A *run* r of $\mathcal{M}$ is an infinite sequence $q_0 \to_{\delta_0}^{L_0} q_1 \to_{\delta_1}^{L_1} \dots$ of states and transitions. We denote $\mathcal{R}$ the set of runs of $\mathcal{M}$. A run is *divergent* if $\sum_{i=0}^{+\infty} \delta_i$ diverges. We denote $\mathcal{R}_\infty$ the set of divergent runs of $\mathcal{M}$. An *initialized* run is one starting at the initial state. We say that a state q *reachable* from q_0 if there exists a run such that $q = q_i$ for some $i \geq 0$. We write $\mathsf{Reach}(q_0)$ for the set of states reachable from q_0, and $\mathsf{Reach}_\infty(q_0)$ for the set of states reachable from q_0 by divergent runs.

A state $q = (s, v) \in \mathcal{Q}$ *satisfies* the formula φ, denoted $q \models \varphi$, if		
$q \models x \# c$	iff	$v(x) \# c$
$q \models x - y \# c$	iff	$v(x) - v(y) \# c$
$q \models b$	iff	$b \in \mathcal{P}(s)$
$q \models enable(l)$	iff	$\exists (s, L, \psi, X, s') \in \mathcal{T}.\ l \in L \wedge v \models \psi$
$q \models \neg\varphi$	iff	$q \not\models \varphi$
$q \models \varphi_1 \vee \varphi_2$	iff	$q \models \varphi_1$ or $q \models \varphi_2$
$q \models \exists\Diamond_{\# c}\varphi$	iff	$\exists r \in \mathcal{R}_\infty(q).\exists i \geq 0, \delta \leq \delta_i.\ q_i + \delta \models \varphi \wedge \delta + \sum_{j<i} \delta_j \# c$
$q \models \forall\Diamond_{\# c}\varphi$	iff	$\forall r \in \mathcal{R}_\infty(q).\exists i \geq 0, \delta \leq \delta_i.\ q_i + \delta \models \varphi \wedge \delta + \sum_{j<i} \delta_j \# c$
where $q + \delta$ is the state $(s, v + \delta)$.		

Table 1: Semantics of TCTL.

A.3 Product automaton

Let $\mathcal{M}_i$ be $\langle \mathcal{S}_i, \mathcal{X}_i, \mathcal{L}_i, \mathcal{T}_i, \mathcal{I}_i, \mathcal{P}_i \rangle$, for $i = 1, 2$. We assume that $\mathcal{S}_1 \cap \mathcal{S}_2 = \mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$. We denote by $sync(\cdot)$ the set of synchronization labels of an edge, that is, for all $i = 1, 2$, for all $t_i = (s_i, L_i, \psi_i, X_i, s_i') \in \mathcal{T}_i$, $sync(t_i) = L_i \cap \mathcal{L}_i$.

The product automaton $\mathcal{M} = \langle \mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{T}, \mathcal{I}, \mathcal{P} \rangle$ is such that:

1. The set $\mathcal{S}$ of locations is $\mathcal{S}_1 \times \mathcal{S}_2$.
2. The set $\mathcal{X}$ of clocks is $\mathcal{X}_1 \cup \mathcal{S}_2$.
3. The set $\mathcal{L}$ of synchronization labels is $\mathcal{L}_1 \cup \mathcal{L}_2$
4. The set $\mathcal{T}$ of edges is obtained as follows. Let $t_i \in \mathcal{T}_i$ of the form $(s_i, L_i, \psi_i, X_i, s_i')$, for $i = 1, 2$.
   (a) Let $t = ((s_1, s_2), L_1 \cup L_2, \psi_1 \wedge \psi_2, X_1 \cup X_2, (s_1', s_2'))$. Then $t \in \mathcal{T}$ iff $sync(t_1) \cap \mathcal{L}_2 = sync(t_2) \cap \mathcal{L}_1$.
   (b) Let $t = ((s_1, s_2), L_1, \psi_1, X_1, (s_1', s_2))$. Then $t \in \mathcal{T}$ iff $sync(t_1) \cap \mathcal{L}_2 = \emptyset$.
   (c) Let $t = ((s_1, s_2), L_2, \psi_2, X_2, (s_1, s_2'))$. Then $t \in \mathcal{T}$ iff $sync(t_2) \cap \mathcal{L}_1 = \emptyset$.
5. The invariant $\mathcal{I}(s_1,s_2)$ is $\mathcal{I}_1(s_1) \wedge \mathcal{I}_2(s_2)$.
6. The labeling $\mathcal{P}(s_1,s_2)$ is $\mathcal{P}_1(s_1) \cup \mathcal{P}_2(s_2)$.

When initial locations $s_{1,init}$ and $s_{2,init}$ are given we will consider only the set of locations of the product reachable from the location $(s_{1,init}, s_{2,init})$.

B The logic TCTL

TCTL is an extenzion of CTL [13]. For simplicity we consider here a simple version of TCTL. The syntax and semantics of full TCTL is given in [16].

Syntax. Let $\mathcal{X}$ be a set of clocks, $\mathcal{B}$ be a set of boolean propositions, and $\mathcal{L}$ be a set of events. The formulas of TCTL are defined by the following grammar:

$$\begin{array}{rcl} \varphi & ::= & x \# c \mid x - y \# c \mid b \mid enable(l) \mid \neg\varphi \mid \\ & & \varphi_1 \vee \varphi_2 \mid \exists\Diamond_{\# c}\varphi \mid \forall\Diamond_{\# c}\varphi \end{array}$$

where $x, y \in \mathcal{X}$, $\# \in \{<, \leq, >, \geq, =\}$, c is an integer constant, $b \in \mathcal{B}$, and $l \in \mathcal{L}$.

Typical abbreviations are $\exists\Diamond\varphi$ for $\exists\Diamond_{\geq 0}\varphi$, $\forall\Box\varphi$ for $\neg\exists\Diamond\neg\varphi$, $\forall\Diamond\varphi$ for $\forall\Diamond_{\geq 0}\varphi$, and $\exists\Box\varphi$ for $\neg\forall\Diamond\neg\varphi$.

Semantics. The formulas of TCTL are interpreted over the set of states of a timed automaton $\mathcal{M}$. The satisfaction relation $\models$ is given in Table 1. The set of states that satisfy the formula φ is called the *characteristic set* of φ, and it is denoted $[\![\varphi]\!]$. $\mathcal{M}$ satisfies φ, denoted $\mathcal{M} \models \varphi$, if all the states of $\mathcal{M}$ satisfy φ.

C Time-abstracting equivalence

C.1 Bisimulation

Let $(\mathcal{Q}, \rightarrow)$ be a labeled graph. A relation $B \subseteq \mathcal{Q} \times \mathcal{Q}$ is a *bisimulation* iff for all $(q_1, q_2) \in B$, $\delta \in \mathsf{R}^+$ and $L \subseteq \mathcal{L}$,

(1) whenever $q_1 \rightarrow^L_\delta q'_1$, then $q_2 \rightarrow^L_\delta q'_2$ and $(q'_1, q'_2) \in B$ for some $q'_2 \in \mathcal{Q}_2$, and

(2) whenever $q_2 \rightarrow^L_\delta q'_2$, then $q_1 \rightarrow^L_\delta q'_1$ and $(q'_1, q'_2) \in B$ for some $q'_1 \in \mathcal{Q}_1$.

Two states q_1 and q_2 are *bisimilar*, denoted $q_1 \sim q_2$, if there exists a bisimulation B such that $(q_1, q_2) \in B$.

C.2 Time-abstracting bisimulation

Let $\mathcal{M}$ be a timed automaton and $(\mathcal{Q}, \rightarrow)$ the model of $\mathcal{M}$. Let $\mapsto \subseteq \mathcal{Q} \times \mathcal{Q}$ be the relation defined by the following rules:

$$\frac{q \rightarrow^L_0 q'}{q \mapsto^L q'} \qquad \frac{q \rightarrow^\emptyset_\delta q'}{q \mapsto^\tau q'}$$

That is, $\mapsto$ is obtained from $\rightarrow$ by abstracting away the exact amount of time elapsed in a time transition. This is done by replacing all labels $\delta \in \mathsf{R}^+$ by the label τ. The *time-abstracting* bisimulation is the greatest bisimulation defined on $(\mathcal{Q}, \mapsto)$.