Clocked FSM Synthesis considering Real Time Constraints

Wolf-Dieter Tiedemann

Universität Passau
Lehrstuhl Rechnerstrukturen (Prof. W. Grass)
D-94030 Passau, Germany
tiedeman@fmi.uni-passau.de

Abstract. *This paper presents a synthesis technique for synchronous controllers that satisfy real time constraints when realized as a clocked finite state machine that is supplied with a prespecified clock rate. The controller needs to be specified as a Timed Automaton, a representation that could for instance be derived from a timing diagram translation. Result is a Mealy machine graph that generally contains purely time-consuming states.*

1 Introduction

As microprocessor technology takes entry to more and more ranges of application, the specific requirements there lead to the identification of always new classes of systems. Accordingly, each class calls for specific design techniques that focus on the particular points of emphasis. Such classes, which came up just recently, are for instance open, distributed, embedded, or reactive systems. These systems are mostly control-oriented, ie. their crucial particularities affect the control path, not the data path, and concentrate on communication issues and the compliance with protocols. This involves quite often the need to meet real time constraints.

Traditional design techniques for controller hardware abstract from timing issues. Specifications are preponderantly given as finite state machine (FSM) representations, which do not imply an inherent notion of time. Mostly, they are interpreted as a discrete-time system, where each state transition is assumed to occur exactly at a valid point of a discrete time scale. Realizations of discrete FSMs operate synchronously to a periodical clock signal. The frequency of this signal determines the timing characteristics. Design methods commonly do not provide for observing specified real time constraints. A FSM interpretation as a continuous-time system leads to asynchronous realizations, which require a particularly careful design in order to avoid races and hazards. Design methods focus on obtaining a self-timed design that is independent from gate or wire delays, but that is not supposed to meet specified real time constraints as well.

In order to be able to specify real time constraints at all, it needs an appropriate specification language. From the designer's point of view, probably the most convenient way to do it is by using timing diagrams. From the CAD tool developer's point of view, a mathematical formalism such as Timed Process Calculi or Timed Automata would be preferable. Actually, it is possible to join both conceptions, since timing diagrams can be translated into these formalisms, as introduced in [Ti92,

Le94] and dealt with comprehensively in [Ti95]. For this paper we assume that specifications are given directly as Timed Automata [AlDi91], a recognized model for describing real time systems.

The presented synthesis technique aims at transforming such a specification into a (timed) model of a clocked Mealy-type FSM with an imprinted clock rate, which satisfies the original specification wrt. a so-called implementation relation **sat**, a preorder respecting real time. **sat** is tailored particularly for hardware issues, eg. it distinguishes between the different obligations regarding the participation in specified or implemented events, depending on whether they are input or output events. The FSM model taken as basis accepts inputs at every time except the instant of the clock event. Outputs are computed with negligible delays but changes are made observable only at those moments when the clock event takes place. The determination of the clock period as a time reference is left to the designer. An unsuited choice could mean that there does not exist an implementing model, but if it exists, a hardware solution using this period is feasible and its formal correctness is guaranteed.

The following sections describe the model of Timed Automata, the implementation relation used, the synthesis procedure, an example, and the conclusions with relation to other work.

2 Timed Automata

A timed automaton [AlDi91] is a finite automaton extended with time annotations. Time annotations refer to values of timers from a finite set T. The time domain $\mathcal{Time}$ is dense, having ∞ as its largest element. A timer can be understood as a stop watch, which always runs in accordance with the real world time but which could also be reset to value 0, from which it continues to count time. The reset appears instantaneously, without consuming any time, and does not affect the continuous flow of time. Thus, at every point in time each timer value shows exactly the time span since the timer's last reset.

Time annotations are attached to the edges of a timed automaton. They consist of

- a predicate on T, which describes a condition that has to hold in order to allow the state transition along the edge to take place and
- a subset of T, which represents the set of timers, which are supposed to be reset just at that instant when the corresponding state transition fires.

Formally, a timed automaton is defined as follows.

Definition. A *timed automaton* is a 6-tuple $A = (\sum, S, s_0, T, T_0, \rightarrow)$, where $\sum$ is a finite alphabet, S is a finite set of states, $s_0 \in S$ is an initial state, T is a finite set of timers, $T_0 \subseteq T$ is an initial resetting, and $\rightarrow$ is a transition relation. Let the set $\Phi(T)$ of permissible predicates ϕ over the set T of timers be defined inductively by

$$\phi ::= t\ op\ c \mid \phi \wedge \phi$$
$$op ::= < \mid \leq \mid \geq \mid >,$$

where t∈T and $c \in \mathit{Time}$. Then $\rightarrow \subseteq S \times \wp(\sum) \times \Phi(T) \times \wp(T) \times S$. An element $(s, a, \phi, R, s') \in \rightarrow$ represents a state transition from s to s′ on event a, when predicate $\phi(T)$ holds. At the occurrence of this transition all timers in R are reset. Instead of $(s, a, \phi, R, s') \in \rightarrow$ we also write

$$s \xrightarrow{a,\ \phi,\ R} s'.$$

Note that predicates like "t = 6" or "t∈(0, 25]" may be used as abbreviations of legal predicates.

Example. Consider the timed automaton of figure 1 as an example. On entering state s timer t_0 is reset. After at least 4 time units have elapsed, an action a will be accepted. Simultaneously, timer t_1 will be reset. Next a multi-action {b, c} may take place, but only, if t_1 has a value within the interval [1, 5]. This event resets timer t_0 again and takes the automaton back to state s.

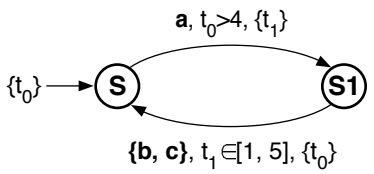


Figure 1. A sample timed automaton

Knowing the definition of two timed automata A_1 and A_2, we can define the parallel composition $A_1 \bullet A_2$ of both. This operation shall be introduced in the sense of process calculi, ie. follow a notion of synchronization on identically labeled transitions.

Definition. The *parallel composition* of two timed automata $A_i = (\sum_i, S_i, P_i, T_i, T_{0,i}, \rightarrow_i)$, i = 1, 2, is described by the timed automaton

$$A_1 \bullet A_2 = (\sum_1 \cup \sum_2, S, (P_1, P_2), T_1 \dot{\cup} T_2, T_{0,1} \dot{\cup} T_{0,2}, \rightarrow),$$

where $S \subseteq S_1 \times S_2$ and $\rightarrow$ is defined by the following inference rules:

(1) $$\frac{s_1 \xrightarrow{a,\ \phi,\ R}_1 s_1'}{(s_1, s_2) \xrightarrow{a,\ \phi \wedge \mathit{intime}(s_2),\ R} (s_1', s_2)} \quad a \cap \sum_2 = \varnothing$$

(2) $$\frac{s_2 \xrightarrow{b,\ \psi,\ Q}_2 s_2'}{(s_1, s_2) \xrightarrow{b,\ \psi \wedge \mathit{intime}(s_1),\ Q} (s_1, s_2')} \quad b \cap \sum_1 = \varnothing$$

(3) $$\frac{s_1 \xrightarrow{a,\ \phi,\ R}_1 s_1' \quad s_2 \xrightarrow{b,\ \psi,\ Q}_2 s_2'}{(s_1, s_2) \xrightarrow{a \cup b,\ \phi \wedge \psi,\ R \cup Q} (s_1', s_2')} \quad a \cap \sum_2 = b \cap \sum_1$$

S contains all states that can be reached from the initial state (P_1, P_2). The predicate *intime*(s) indicates whether state s is enabled or will be enabled in the future, ie. whether there is still a chance to leave this state sometime by some transition.

In general, *intime*(s) equals the disjunction of all predicates attached to transitions emerging from s, ie.

$$\mathit{intime}(s) = \bigvee_{(s,\cdot,\phi,\cdot,\cdot) \in \rightarrow} \text{upper_bounds}(\phi),$$

where upper_bounds fades away all simple terms of the kind (t *op* c), *op*∈{≤, <}, from the denotation of ϕ. Unfortunately, this is an undesirable representation, since enabling predicates on transitions are defined to be a conjunction of simple (t *op* c) terms. In order to retain this format, we require that there is *one* predicate ϕ_{max} among all the predicates ϕ under consideration that possesses for each t_i appearing in the denotation of any upper_bounds(ϕ) = $\bigwedge_{i \in I_\phi} t_i \; op_i \; c_i$ a conjunctor ($t_i \; op \; c_i^{max}$) in its own denotation, where *op*∈{≤, <} and $c_i^{max} \geq c_i$. Note that for any timer $t_k \in T$ not appearing in the denotation of some predicate upper_bounds(ϕ), c_k equals infinity, since the predicate ϕ can be interpreted as $\phi = \phi \wedge (t_k < \infty)$ as well. Then

$$intime(s) = \bigwedge_{i \in I_{upper_bounds(\phi_{max})}} t_i < c_i.$$

The requirement of the existence of ϕ_{max} seems to be rather restrictive in the general case. Actually, it is not, since if a timed automaton is derived from a translation of timing diagrams or timed processes (what is strongly recommended for reasons of convenience) it is ensured that ϕ_{max} exists.

3 The Implementation Relation

A key premise for any formally based synthesis technique is having a verification method at disposal that allows to decide whether the synthesized result is a formally correct implementation of the given specification. This section introduces a technique that compares two timed automata in this respect.

For descriptions given by state-transition systems it is a widespread conception to consider the difference between a specification and a correct implementation just as consisting of the visibility of some additionally interspersed events in the implementation behavior. Therefore, it is consistent to perform the verification using a *restriction*-operator \L, which simply fades out all events in $L \subseteq \sum_I$ that are considered to be implementation-internal, and to check, whether or not a restricted implementation I\L satisfies a specification S, by verifying the (bisimulation) equivalence of S and I\L.

Unfortunately, this approach is too stringent in practice. The reason is that the different obligations of events in specification and implementation, depending on whether they are input or output events, are not considered. For instance a specification may allow two alternative output actions in either order. Clearly, any correct implementation chooses exactly one ordering for arbitrary reasons and leaves the other out of account. Since neither action may be restricted, the equivalence-checking approach cannot induce the expected result. Conversely, there is no reason to prevent an implementation from participating (potentially) in some actions, which according to the specification won't ever occur. As an example consider a 4-bit register, which can of course go as an implementation for a binary coded digit storage module, even if it accepts a **1** event on its 3rd bit port when also the 4th (most significant) bit is set to **1** and, hence, potentially stores a code **11****, which is impossible to occur in a well-behaving environment according to the specification.

Obviously, the relation between a specification and its implementation (even if restricted to the same alphabet) is not symmetric in general. [LaTh91, CGL93] define a *refinement* ordering ⊲ on (timed) modal transitional behaviors that characterizes a refinement I of S, written I ⊲ S, as a valid

implementation. For a simple transitional system a reflexive, transitive relation **sat** that considers I to be a valid implementation of S when I **sat** S can be found in [Ti92]. It is extended to timed transitional systems in [Ti95] and shall be adapted here to timed automata.

3.1 Intuitive Characterization of "sat"

The general idea behind the implementation relation is to consider a behavior I as a correct implementation of a specifying behavior S, when *every specified input event is also implemented*, and *every implemented output event is also specified*.

This simple conception needs to be modified a little for the general case, where we must expect that $\sum_S \subseteq \sum_I$, ie. the implementation automaton uses a larger alphabet containing all the implementation-internal events. When comparing I and S, these events can be ignored wrt. their effects on the qualitative behavior, but they must not be disregarded wrt. to their time effects.

For instance, consider the implementation sequence

$$\xrightarrow{\{t\}} I \xrightarrow{i,\ t=2,\ \{s\}} I' \xrightarrow{a,\ s<5,\ \cdot} I''$$

and the specification sequence

$$\xrightarrow{\{u\}} S \xrightarrow{a,\ u<6,\ \cdot} S'.$$

Let $i \notin \sum_S$ and a be an input event. To answer, whether I satisfies S or not, we have to check, if the specified input a is accepted from I, and we see it is not. Instead, I performs first an internal move and evolves by that to the subsequent behavior I′. However, it is not reasonable to consider I directly as a failing implementation, since later I′ is accepting a quite well and still in time. It would be more suitable to make the result dependent from the comparison of I′ and S. But this is still not completely the right way, since actually we have to take into account that at the time of comparison, S has already evolved in time by 2 units. What has to be compared in fact are I′ and the *time-shifted* behavior t»S (where t=2). Conceptually, the specification sequence has to be split up into

$$\xrightarrow{\{t,\ u\}} S \xrightarrow{-,\ t=2,\ \cdot} t\text{»}S \xrightarrow{a,\ u<6-t,\ \cdot} S'.$$

The comparison of I′ and t»S in the context t=2 will reveal, that I implements S, provided I′ implements S′.

In summary, we regard I as implementing S, when each of the following requirements is satisfied:

(i) If S specifies some input event at a point t in time, then this event must be accepted from I at the same point in time, *or* I must have the chance to "escape" by itself to a subsequent behavior at a point τ in time *before* t. The subsequent behavior I′ has recursively to implement the subsequent behavior S′ or the time-shifted behavior τ»S, respectively.

(ii) If I performs some output event at a point t in time, then this event must be specified in S for this point in time. Recursively, the subsequent behavior I′ has to implement S′.

(iii) If there is any output specified in S at all, then it must be sure that the implementation actually performs some output, or, at least, can "escape" by performing an event under its own control at a point in time earlier than the time limit for the latest possible specified output event.

Requirement *(iii)* shall just ensure that a result can be obtained in cases where only outputs are specified, but only inputs are implemented.

3.2 Definition of a Preorder "sat" on Timed Automata

In order to formalize the definition of the implementation relation, we have to introduce a new notion, namely the notion of *history*, that allows to keep track of the present context regarding the timer values in the course of an I-S-comparison.

At each time the present value of any timer depends on the point of its last reset. Usually, for different timers this reset point is different. It is the purpose of history to map the differently named timers and the various possibilities of time distances between their resettings into a common scale. A history is basically a list

$$\Gamma = \langle v_0, e_1, v_1, e_2, \ldots, v_N \rangle$$

where each v_i represents a set of timers that have been reset simultaneously and e_{i+1} denotes an interval that represents the time distance between the resettings of v_i and v_{i+1}. In order to model time dependency, the representation of such intervals may use references to previous intervals. A reference in e_i to the interval e_j is written as π_{j-i}.

Having a history, it is possible to evaluate, which corresponding interval is decribed by a given predicate. Instead of going too much into detail, we will just give an example.

Example. Let $\Gamma = \langle \{t_2\}, 20\text{-}\pi_1, \{t_1\}, (0,20), \{t_0\} \rangle$ be a history and $\phi = t_0{>}40 \wedge t_1 \in (30, 40) \wedge t_2 \leq 35$ be a predicate. This situation could arise from a timed graph that has been traversed up to state s along the path shown in figure 2. We are interested in finding out whether ϕ is an empty interval or not.

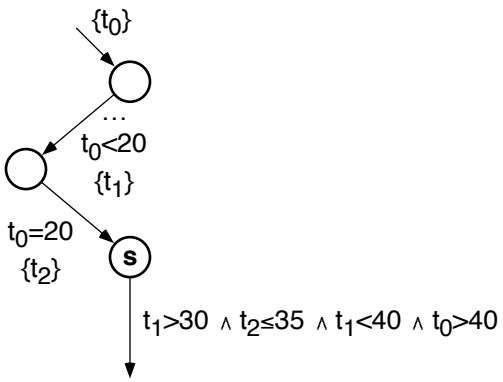


Figure 2. Path in a Timed Graph

In a first step, we can map the predicate onto the scale given by Γ. If we call this representation $\phi|_\Gamma$, then

$$\phi|_\Gamma = \pi \leq 35 \wedge \pi > 30\text{-}\pi_1 \wedge \pi < 40\text{-}\pi_1 \wedge \pi > 40\text{-}\pi_1\text{-}\pi_2.$$

Correspondingly, the interval that is described by predicate ϕ under history Γ, call it $Int_\Gamma(\phi)$, is given as

$$Int_\Gamma(\phi) = (0, 35] \cap (30, 40)\text{-}\pi_1 \cap (40, \infty)\text{-}\pi_1\text{-}\pi_2.$$

In order to evaluate this interval, we resolve

$(30, 40)\text{-}\pi_1 = (30, 40)$ - *Int*(*smallest-lb**$(1, \Gamma)$, *largest-ub**$(1, \Gamma)$) and

$(40, \infty)\text{-}\pi_1\text{-}\pi_2 = (40, \infty)$ - *Int*(*smallest-lb**$(2, \Gamma)$, *largest-ub**$(2, \Gamma)$).

*smallest-lb**(n, Γ) and *largest-ub**(n, Γ) denote the smallest possible value and largest possible value, respectively, for the time span that has elapsed since the last but n-1-th event in history Γ. In this example we obtain for instance

*smallest-lb**$(1, \Gamma) = 20 - (20 - \varepsilon) = \varepsilon$ and

*largest-ub**$(2, \Gamma) = 20$.

In summary, we have

Int(*smallest-lb**$(1, \Gamma)$, *largest-ub**$(1, \Gamma)) = (0, 20)$ and

Int(*smallest-lb**$(2, \Gamma)$, *largest-ub**$(2, \Gamma)) = [20, 20]$,

so that

$(30, 40)\text{-}\pi_1 = (30, 40) - (0, 20) = (10, 40)$ and

$(40, \infty)\text{-}\pi_1\text{-}\pi_2 = (40, \infty) - [20, 20] = (20, \infty)$.

Hence, $Int_\Gamma(\phi) = (0, 35] \cap (10, 40) \cap (20, \infty) = (20, 35] \neq \varnothing$ denotes an existing interval.

The decision, whether some predicate corresponds to an empty interval, or two predicates are equal or are in a subset relation can be done efficiently by mapping the predicate *and* the history into a matrix representation like the one used in [Di89] and solving the all-pair shortest path problem on these matrices.

Now we are prepared to define **sat**. We do it with the help of a binary relation $\mathcal{S}$ on *deterministic* automata. We denote the input alphabet of a timed automaton A by **i**A and the output alphabet by **o**A. It holds **i**A $\cup$ **o**A $= \sum_A$ and **i**A $\cap$ **o**A $= \varnothing$. Further, we use the list operators $\circ$ to denote a push ($l_0 \circ \langle l_1, \ldots\rangle = \langle l_0, l_1, \ldots\rangle$) and \R to denote a collective set-subtraction on odd-numbered list entries ($\langle l_1, l_2, l_3, \ldots\rangle\backslash R = \langle l_1\backslash R, l_2, l_3\backslash R, \ldots\rangle$). If I, J are disjoint intervals, and the upper bound of I is less than the lower bound of J, we denote this by I<J. An interval $I^{\leq}$ denotes a left-open interval with lower bound 0 and the same upper bound as I, and $I^{<}$ is the same as $I^{\leq}$ except that is made right-open if $I^{\leq}$ is right-closed. Moreover, (and only for the next definition) we allow an interval to be even a non-contiguous set.

Definition. Let S, I be deterministic timed automata and **i**S $\subseteq$ **i**I, **o**S $\subseteq$ **o**I. A relation $\mathcal{S}$ is called *satisfaction* wrt. history Γ, if

$(I, S, \Gamma) \in \mathcal{S} \Rightarrow$

(i) $S \xrightarrow{a,\ \phi,\ R} S' \wedge a \subseteq \mathbf{i}S \wedge Int_\Gamma(\phi) \neq \varnothing \Rightarrow$

(1) $not\text{-}covered = \varnothing \vee (\exists I', b \cap \mathbf{i}I = \varnothing.\ I \xrightarrow{b,\ \psi,\ Q} I' \wedge Int_\Gamma(\psi) < not\text{-}covered)$,

where *not-covered* := $Int_\Gamma(\phi) \setminus \bigcup_{I \xrightarrow[b\cap \mathbf{s}S=a]{b,\psi,\cdot} \cdot,} Int_\Gamma(\psi)$,

(2) $\forall I', b\cap\sum_S=a.\ I \xrightarrow{b,\psi,Q} I' \wedge Int_\Gamma(\psi) \cap Int_\Gamma(\phi) \neq \emptyset$

$\Rightarrow (I', S', R\cup Q \circ (\phi|_\Gamma \wedge \psi|_\Gamma) \circ \Gamma\backslash R\backslash Q)\in\mathcal{S}$,

(ii) $I \xrightarrow{b,\psi,Q} I' \wedge b\cap\sum_S \subseteq \mathbf{o}S \wedge Int_\Gamma(\psi)\neq\emptyset \Rightarrow$

(1) $Int_\Gamma(\psi) \setminus \bigcup_{S \xrightarrow[a=b\cap \mathbf{s}S\neq\emptyset]{a,\phi,\cdot} \cdot,} Int_\Gamma(\phi) = \emptyset$,

(2) *(a)* $b\cap\sum_S=\emptyset \Rightarrow (I', S, Q \circ \psi|_\Gamma \circ \Gamma\backslash Q)\in\mathcal{S}$,

(b) $\forall S', a=b\cap \mathbf{s}S\neq\emptyset.\ S \xrightarrow{a,\phi,R} S' \wedge Int_\Gamma(\phi) \cap Int_\Gamma(\psi) \neq \emptyset$

$\Rightarrow (I', S', R\cup Q \circ (\phi|_\Gamma \wedge \psi|_\Gamma) \circ \Gamma\backslash R\backslash Q)\in\mathcal{S}$,

(iii) *out-expected* $\neq \emptyset \Rightarrow$

$$\exists I', b\cap \mathbf{i}I=\emptyset.\ I \xrightarrow{b,\psi,Q} I' \wedge \begin{cases} Int_\Gamma(\psi) \cap out\text{-}expected^{<} \neq \emptyset & \text{if } b\cap \mathbf{o}S=\emptyset \\ Int_\Gamma(\psi) \cap out\text{-}expected^{\leq} \neq \emptyset & \text{else,} \end{cases}$$

where *out-expected* := $\bigcup_{S \xrightarrow[a\subseteq \mathbf{o}S]{a,\phi,\cdot} \cdot,} Int_\Gamma(\phi)$.

Now we define **sat** to be the *largest* satisfaction relation $\mathcal{S}$, ie.

I **sat** $S \Leftrightarrow \exists\mathcal{S}.\ \mathcal{S}$ is a satisfaction $\wedge$ $(I, S, \Gamma_0)\in\mathcal{S}$,

where Γ_0 is the initial history consisting of just one entry consisting of the set of all timers of I and S that are initially reset, ie. $\Gamma_0 = \langle T_0(I)\cup T_0(S)\rangle$.

It can be shown that **sat** is reflexive and transitive, ie. **sat** is a preorder.

4 Transformation into a FSM Representation

The aim of this section is to present the synthesis technique for transforming a specification given as a timed automaton into a (timed) model of a clocked Mealy-type FSM with a user-determined clock rate, which satisfies the original specification wrt. **sat**. First, we review the behavior of a FSM circuit model.

4.1 Circuit model

A typical Mealy machine realization is depicted in figure 3. It consists of a combinational logic block and a clocked register to store the present state of the machine. The timing behavior of such a realization is described by the propagation delay Δ_C of the combinational block as well as the delay

Δ_S of the register and depends on the clock events *clk*. Whenever an input event occurs (let this be at time t) the corresponding output event will occur at time $t+\Delta_C$. At this same time the new state code appears at the input port of the register. It will be put through to the register's output port at the time of the next clock event, which occurs later in time than $t+\Delta_C$. Let next(*e*)(*t*) denote the earliest point in time later or equal than *t*, at which event *e* occurs, then the new state code appears at time $\text{next}(clk)(t+\Delta_C)+\Delta_S$. In order to avoid metastability problems, the following timing constraint has to be satisfied

$$t+\Delta_C \notin [\text{next}(clk)(t+\Delta_C) - \Delta_{setup}, \text{next}(clk)(t+\Delta_C) + \Delta_{hold}] \qquad (1)$$

where Δ_{setup} and Δ_{hold} represent the setup- and hold time of the register.

A second problem will arise, if a further input change occurs between t and $\text{next}(clk)(t+\Delta_C)+\Delta_S$, because it catches the circuit in a transient condition, where the present state code is not yet fed back to the combinational block.

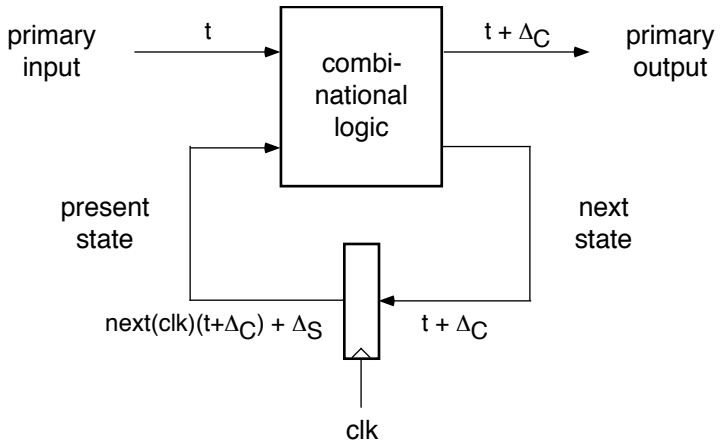


Figure 3. Mealy machine realization

To justify such a realization nevertheless, designers commonly assume a circuit operation in *fundamental mode*, ie. inputs change only one-at-a-time and only when the circuit is in stable state. Obviously, this is a very restrictive assumption and cannot be drawn eg. for reactive systems that have to accept arbitrary inputs at every time.

Fortunately, in many cases it is not necessary to have an explicit state transition (of the hardware realization) between two successive input events: if the (specified) behavior is confluent, it does not make any difference in which order the input changes occur: the reaction will finally be the same. The hazard problem is a good instance. In order to avoid hazards, a (cheap) approach is just to "wait a little time until the affairs are settled" and then to communicate the accumulated changes at once. This could be done by leading the outputs through a clocked register.

We are going to adopt this idea and pass the outputs through an additional register. (One could argue that we now have a Moore-type machine. Although this is not completely right – there is no output decoder, output and next state code are computed concurrently – it cannot be completely denied, also. We prefer to call it Mealy-type, nevertheless, since the kernel is realized following the mathematical model of a Mealy machine.) This requires of course that (1) there are pathes in the specifying automaton that allow multiple input changes before some output needs to occur, and (2)

that it is possible to move every output event in a way that it can be performed in association to the clock event of the output register. It will be the essence of the synthesis technique proposed in the sequel to verify these requirements.

The most important implication of this clocked output register is that by its effective clock period it establishes a real time behavior of the FSM model that can be related to the specified real time constraints.

For the following considerations let us assume delays that are “nearly zero”, or ε, compared to the clock period. Taking advantage of this simplification allows a reformulation of timing constraint (1)

$$t+\varepsilon \notin [\text{next}(clk)(t+\varepsilon) - \varepsilon, \text{next}(clk)(t+\varepsilon) + \varepsilon] \qquad (2)$$

which implies

$$t \neq \text{next}(clk)(t), \qquad (3)$$

ie. no input event must occur at the same time as a clock event (a fundamental problem when interfacing a synchronous module to an asynchronous world). We comply with this requirement by passing the input through a synchronization element that delays only those input events by ε, which otherwise would occur together with the clock event. (Synchronization elements cannot avoid the metastability problem totally, however, there are special analog realizations which minimize the risk of malfunction [Se80].)

The modified circuit model is shown in figure 4.

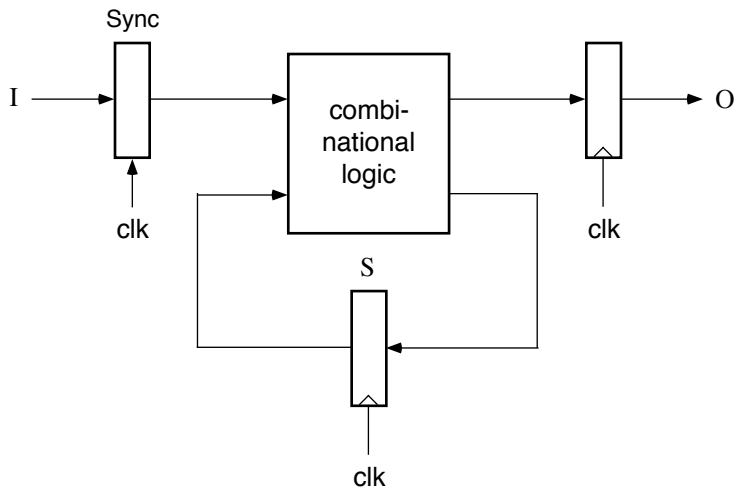


Figure 4. Circuit model

In the next section we look at a way to transform an arbitrary timed automaton into a model that describes a behavior like that of such a circuit model.

4.2 Synthesis Steps

The key idea for the transformation is to extend a given timed automaton A by a clock signal and then to modify this representation stepwise towards a behavior that corresponds to the (timed) model of a clocked Mealy-type FSM as it has been given in section 4.1. Such a model consists of subgraphs like the one depicted in figure 5. It depicts the alternating sequence of concurrent input events (a and

b) and simultaneous output events (y and z), which, moreover, are bound to occur at the same time as the clock event (clk).

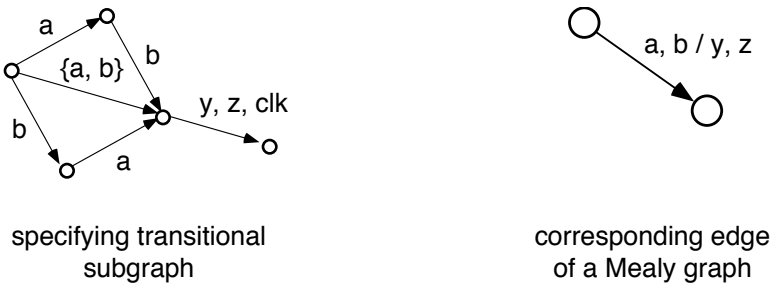
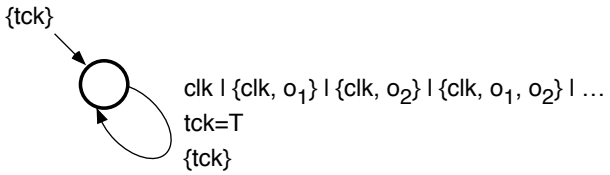


Figure 5. Resolution of a a, b/y, z-labeled Mealy transition

Step 1. Introduction of the clock period. This is simply performed by composing A in parallel with a timed automaton **syn**, which forces each output event o_1, o_2, … of A to occur simultaneously with the clock event. The graph of **syn** looks as follows



The clock period T has to be proposed by the designer. The timer tck allows transitions of **syn** only when it has reached the value T. In performing a transition, tck is reset to 0.

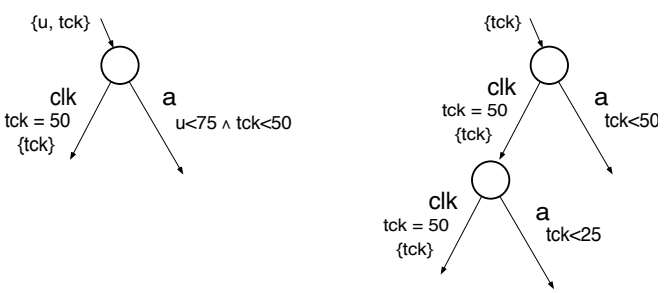
Binding all output events into a global equidistant time scheme is a crucial manipulation that could mean in case of an unsuited selection of T that there does not exist an implementing model. This is why we have to verify the suitability of T in step 2.

Step 2. Verification, whether

A • **syn sat** A,

where **sat** is the implementation relation of section 3.2. If actually A • **syn** satisfies A, the proposed clock period T is admissible. Otherwise, the synthesis should go back to step 1 and ask for a better suited clock rate.

Step 3. Transformation into the model of a clock-synchronous machine. The automaton A • **syn** is "unwound" such that each enabling predicate assigned to any transition uses just one single timer, eg.



This appears to be a complicated transformation. Actually, it is a by-product of the verification run from step 2: by constructing a satisfaction $\mathcal{S}$ that contains (A • **syn**, A, $\langle T_0(A) \cup \{\mathsf{tck}\}\rangle$) and keeping track of the transitions that cause the recursive descent to further elements of $\mathcal{S}$, we obtain the desired automaton "for free". Clearly, this automaton satisfies A.

In a postprocessing step we can overwrite every input predicate with "tck<T" (T=50 in the example above), independent of its actual value. This amounts at the very most to an extension of the represented time interval and conforms to **sat**, since the (clocked) implementation is allowed to (unnecessarily) accept inputs also at points in time, where the specification already tells that no inputs need to be expected. The result is called A_M. It is still guaranteed that A_M **sat** A.

Step 4. Transformation into a symbolic Mealy machine description. First, any timing annotation in A_M can be stripped off, since now timing is obvious and needs no longer to be explicitly annotated (every transition labeled ia. with the clk action has an enabling predicate "tck=T" and a reset set {tck}, all others have a predicate "tck<T" and no timers to be reset). The resulting representation A_M^{untimed} is a simple transitional graph without any time annotation. On A_M^{untimed} we can perform a conventional state reduction, leading to a representation $A_M^{\text{untimed, reduced}}$. Up to this stage it still holds

$$A_M^{\text{untimed, reduced}} \bullet \textbf{syn sat } A.$$

With the next (irreversible) manipulation, we introduce the Mealy machine typical strict ordering "input ι before output ω" by deleting those transitions, which express a simultaneity of ι and ω. The transformation rule says

Every transition $s \xrightarrow{\iota \cup \omega \cup \{\mathsf{clk}\}} s'$,where $\iota \subseteq \mathbf{i}A$, $\iota \neq \varnothing$, $\omega \subseteq \mathbf{o}A$, will be

- deleted, if there is a path $s \xrightarrow{\omega \cup \{\mathsf{clk}\}} s'' \xrightarrow{\iota} s'$, otherwise it will be
- replaced by a path $s \xrightarrow{\omega \cup \{\mathsf{clk}\}} \sigma \xrightarrow{\iota} s'$, where σ is a new state.

The inverse transformation cannot re-build the original representation in each case. However, this loss in granularity is unavoidable, since the operational Mealy model is a coarser model than the transitional graph model (cf. figure 5).

From this intermediate graph we derive the symbolic Mealy graph M_{symb} just by performing the two steps

- states of M_{symb} are those having at least one in-going output transition, ie. a transition labelled with ω∪{clk};
- transitions of M_{symb} are generated for every path between two states of M_{symb} that does not visit a third state of M_{symb}; the label of the newly generated transition is determined by the union of all labels along the path.

Step 5. Transformation into Mealy format. In order to get the final format of a Mealy machine description as it is expected from logic synthesis tools, state preserving edges have to be explicitly included (complete specification) and symbolic event sets have to be replaced by bitvectors.

5 An Example

Figure 6 shows the timing diagram specification and the timed automaton of a simple interface circuit that converts between a full interlocked and a half interlocked handshake within two write cycles (see [Ti92] for an untimed version of this example).

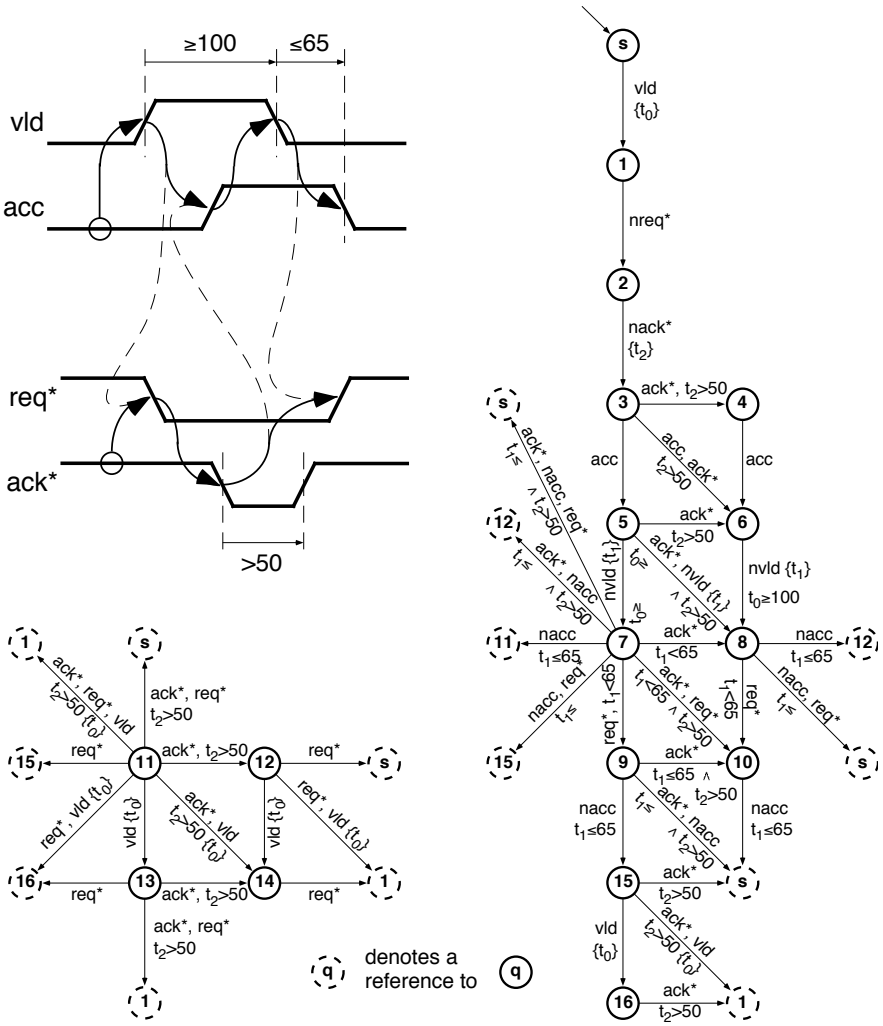


Figure 6. Specification of an interface circuit

Now, an imaginary designer is asked to propose a clock period length. If he selects eg. 100 ns in this example, the verification returns a failure, since it is impossible to generate an output event nacc (ie. acc = 0) within a time span of 65 ns when using a 100 ns clock period. If the designer selects a period length of 60 ns, he should receive a warning, since in this case it cannot be excluded that a narrow pulse nack*–ack* is lost. This is not a "hard" violation, since according to the specification this has the same effects like waiting on nack* till infinity – what is implied in the specified time interval (50, ∞) between both events nack* and ack*. Such a situation can be detected automatically: it applies whenever some event and its inverse appear in the same label.

When proposing a clock period length of 50 ns a solution will be generated that does not need any time counting states. It is the same result that is obtained for the untimed case [Ti92]. Finally, figure 7

shows the solution that is generated for an assumed clock period length of 30 ns. Note that states 2 and 3 don't show state preserving edges, ie. they are taken only for 1 clock period. They can be considered as being *time counting states*. The Mealy graph is incomplete, ie. it does not show transitions for unspecified input events. These transitions can be added on user demand. Optionally, such transitions take the Mealy machine to either a distinguished error state or to the initial state.

The statistics for this example are shown in the following table (entries represent state numbers):

	50 ns period	30 ns period
A_M	99	182
$A_M^{\text{untimed, reduced}}$	10	16
M_{symb}	6	8

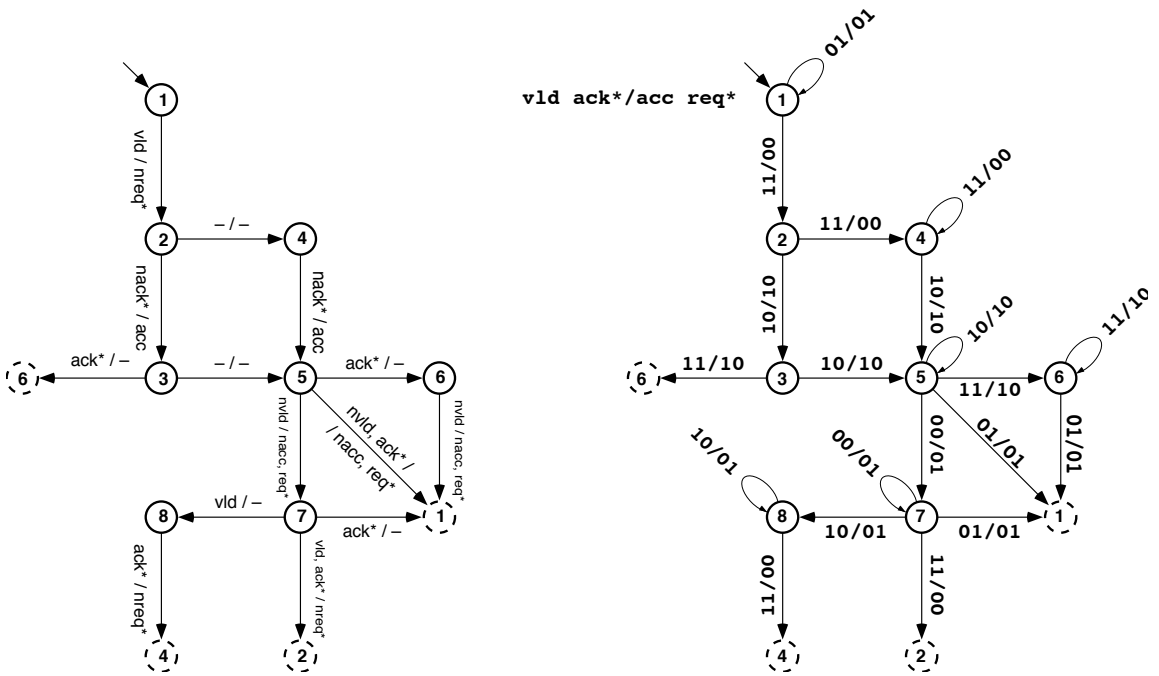


Figure 6. Symbolic FSM graph and Mealy graph

6 Conclusions, Prospects and Related Work

The focus of this paper has been on the formal synthesis of clocked FSMs from timed automata. The resulting FSM considers real time constraints. The design correctness is guaranteed with respect to an implementation preorder **sat**. A necessary user input is the clock period length. For an unsuited clock rate there could exist no solution. Although for "some" clock rate there is always a solution (just make the rate high enough), this solution might not be technically feasible or desirable (rate too fast, too many counting states). Currently, we are working on an automatic derivation of reasonable clock rates by applying Linear Programming techniques.

A synthesis tool has been implemented and integrated into the SABINA system for communication processor synthesis [Ti95] as a distinguished synthesis option. By that a specification can also be done graphically in terms of timing diagrams, without needing timed automata as a user interface. Of course, they still serve as an internal representation.

The presented proposal for solving the design problem of synchronous controllers that comply with real time specifications automatically is probably one of the first. To the author's knowledge there is merely one other publication on this topic, namely [MPS95], that presents a semi-decision procedure to prove, whether for a given system (specified as a timed automaton) there exists a realizable controller, such that their interaction will produce a "good" behavior. "Good" behavior corresponds to the set of all those traces that satisfy a specific criterion, which can be formulated in terms of Temporal Logic. The proof is constructive, hence, the controller description itself is a by-product. Concerning the constant time distance t between always two control steps it can just be stated that some $t>0$ exists.

References

[AlDi91] R.Alur, D.Dill: The Theory of Timed Automata; Proc. REX'91 Workshop on Real Time: Theory in Practice, LNCS 600, Springer (1992), 45-73.

[CGL93] K.Cerans, J.C.Godskesen, K.G.Larsen: Timed Modal Specification - Theory and Tools; Proc. Int. Workshop on Computer Aided Verification CAV'93, LNCS 697, Springer (1993), 253-267.

[Di89] D.L.Dill: Timing Assumptions and Verification of Finite-State Concurrent Systems; Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407, Springer (1989), 197-212.

[LaTh91] K.G.Larsen, B.Thomsen: Partial specifications and compositional verification; Theoretical Computer Science 88 (1991), 15-32.

[Le94] S.Lenk: Extended Timing Diagrams as a Specification Language; Proc. 3rd European Design Automation Conference EURO-DAC'94 (1994), 28-33.

[MPS95] O.Maler, A.Pnueli, J.Sifakis: On the Synthesis of Discrete Controllers for Timed Systems; Proc. 12th Annual Symp. on Theoretical Aspects of Comp. Science STACS'95, LNCS 900, Springer (1995), 229-242.

[Se80] C.L.Seitz: System Timing; chapter 7 in C.Mead, L.Conway (eds.): Introduction to VLSI Design; Addison-Wesley (1980).

[Ti92] W.D.Tiedemann: An Approach to Multi-paradigm Controller Synthesis from Timing Diagram Specifications; Proc. 1st European Design Automation Conference EURO-DAC'92 (1992), 522-527.

[Ti95] W.D.Tiedemann: Synthese von Kommunikationsprozessoren aus Spezifikationen in Form von Impulsdiagrammen; Universität Passau, Dissertation (1995).