# Reconciling Real-Time with Asynchronous Message Passing

M. Broy, R. Grosu, C. Klein*

Institut für Informatik, TU München, D-80290 München
email: {broy,grosu,klein}@informatik.tu-muenchen.de

**Abstract.** At first sight, real-time and asynchronous message passing like in SDL and ROOM seem to be incompatible. Indeed these languages fail to model real-time constraints accurately. In this paper, we show how to reconcile real-time with asynchronous message passing, by using an assumption which is supported by every mailing system throughout the world, namely that messages are time-stamped with their sending and arrival time. This assumption allows us to develop a formalism which is adequate to model and to specify real-time constraints. The proposed formalism is shown at work on a small real-time example.

## 1 Introduction

Asynchronous message passing has gained a lot of popularity in the industrial community. Two of the most prominent specification and description languages for real-time systems use it as their basic communication and synchronization scheme between processes: the ITU-T specification and description language SDL [OFMP+94, IT93] and the ObjecTime specification and description language ROOM [SGW94].

Basically, the behavior of a system in SDL or ROOM is given as a set of asynchronously communicating extended finite state machines (EFSM). A signal instance, i.e., a message, is created when an EFSM executes an output and ceases to exist when the receiving EFSM consumes the signal in an input. Communication channels convey the signal instance from the sender to the receiver. When the signal arrives, it is kept in the input port of the receiver, which is an unbounded FIFO queue, until the receiver consumes it. The virtue of this communication scheme is the loose coupling between system parts: a sender is never blocked because a receiver is not ready to communicate.

In order to express real-time constraints, both SDL and ROOM allow to access the global, *actual time* and to set and reset *timers*. A timer is a stopwatch which is set with an expiration time. The expiration of the timer is then signaled to

* This work is partly sponsored by the Deutsche Forschungs Gemeinschaft (DFG) project SYSLAB

the process as an ordinary input signal. When a timer is no longer needed, it can be reset before its expiration, to avoid spurious expirations.

Unfortunately, these timing facilities are not precisely defined in the SDL and ROOM semantics [Hin96, Leu95, BB91, SGW94] and therefore implementation dependent. Practical experience has shown that the SDL timing facilities behave well only if the tolerance intervals are at least 100 times the average instruction time of the CPU used for the implementation [OFMP+94]. Moreover, these timing facilities are not always expressive enough. In order to understand why this is the case, let us examine a small fragment of a telephone protocol.

> *" After the caller has lifted the phone receiver, he must receive a dial tone within 0.1s."*

Assuming that no delay occurs on the communication lines, this protocol can be expressed very intuitively by using a timed variant of message sequence charts, as shown in Figure 1, left, where `offH` is the abbreviation for off-hook, `dtB` is the abbreviation for dial-tone-begin and `a,b` are points in time (see [IT96] for the MSC standard).

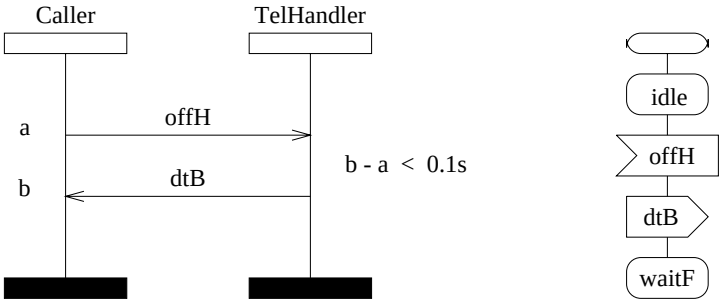



**Fig. 1.** MSC and EFSM for the protocol fragment

This requirement is neither expressible in SDL nor in ROOM. The best one can do is to write the EFSM given in SDL-notation in Figure 1, right. It consists of a start symbol, the control states `idle` and `waitF`, and a transition labeled by the input `offH` and the output `dtB`. This EFSM neither guarantees that the transition is taken synchronously with the generation of the `offH` message nor that the transition takes less than 0.1s. The transition can be delayed, even if the `dtB` message was already queued in the input port. Moreover, it is unknown how long does it take to produce the output `dtB`. Note that timers are not helpful in this case, since they can only be used to enforce a reaction if a signal does not *arrive* within a given time interval.

One of the most successful formal models for specifying and verifying real-time systems are timed automata [AD94, HNSY92]. The formalism of timed automata

generalizes finite state machines over infinite strings to generate (or accept) infinite sequences of states which are additionally constrained by timing requirements.

A timed automaton operates with finite control – a finite set of locations, a finite set of propositions and a finite set of real-valued clocks. All clocks proceed at the same rate and measure the amount of time that has elapsed since they were reset. Each edge of the automaton may reset some of the clocks and assign new values to a set of propositional variables. Each edge and each location may put certain constraints on the values of the clocks and propositions corresponding to that location.

Using a syntax similar to the one given in [HNSY92], the caller and the telephone handler automata can be expressed as shown in Figure 2. Suppose the channels `i` and `o` and the timer `x` are modeled with shared variables, and that composition is done by interleaving. If initially $\texttt{i} \neq \texttt{offH}$, then only the caller automaton can proceed by "sending" `offH` along `i` simultaneously with reseting the timer `x`. The telephone handler is then able to make its first transition. The second transition of the telephone handler can than be taken at any moment such that $\texttt{x} < \texttt{0.1s}$ with the effect of "sending" `dtB`.

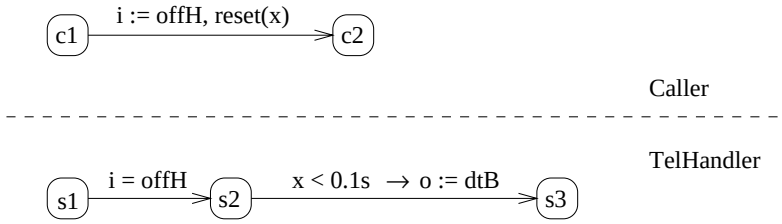



**Fig. 2.** The timed automata solution

The above timed-automata solution uses shared variables both for communication and for time synchronization. Although shared variables are appropriate for single-processor systems, they are unnatural in the context of distributed systems. Moreover, the timed automata approach is not modular, since timed automata may constrain their environment, as we will show in the next section.

Note that the sharing of the timer variable `x` between the caller and the telephone handler implies that the receiver knows when the `offH` message has been sent. In the following, we use essentially the same idea, but in a modular way, and within a message-passing communication paradigm. In the solution we propose, each message is time-stamped both with the sending and the arrival time, as this is a common practice of any mailing system. To simplify the semantics (but without loss of generality because the communication medium can be modeled again as a component) we assume instantaneous delivery, i.e., the sending and

the arrival time are considered to be the same.

As in SDL and ROOM, we describe the behavior of processes with extended finite-state machines which we call *timed state transition diagrams* (TSTDs). We also assume asynchronous communication and the presence of a timer variable `now` containing at each moment the actual time. However, in contrast to SDL and ROOM we allow to access the arrival time of a message by using time-stamped input patterns. For example, the pattern `i?offH@t` matches the first `offH` message in the input port `i`, and updates the variable `t` with the arrival time of this message. This variable can be used in conjunction with `now` to guard a transition, as in `now - t < 0.1s`. When the transition is taken, it takes place instantaneously.

Using our approach, the timing requirement of the telephone protocol fragment can be expressed with only one TSTD – the telephone handler – as shown in Figure 3.

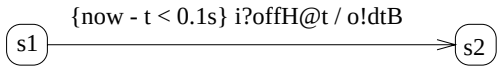



**Fig. 3.** Our solution

The input and the output patterns are separated by a slash, and written in a syntax inspired by CSP [Hoa85]. The guard (or precondition) is written within curly brackets before the input/output patterns. Assignments to the state variables can be written between curly brackets after the input/output patterns (the postcondition).

Intuitively, a component specified by this TSTD behaves as follows. If the component is in the control state `s1`, the first message on the port `i` is `offH`, and the arrival time `t` of this message is such that `now-t<0.1s`, then the component has the choices either to perform the transition immediately, or to perform it at any later moment `now'` which also satisfies the condition `now'-t<0.1s`.

To express timeouts and priorities between transitions, we also allow to test for the absence of any message in a given port. For example, the input pattern `i?`$\ominus$ is satisfied if no message is in the input port `i` at time moment `now`. The input pattern `i?a, j?`$\ominus$ is satisfied if at time point `now` the "prioritized" port `j` is empty whereas the first message in the port `i` is `a`. Hence, if a transition is labeled with `i?a, j?`$\ominus$ and another with `j?b`, and both have the same source control node, then the second transition has a higher priority.

If a transition has no pattern for an input port, then no message is discarded from that port in this transition. Similarly, if a transition has no pattern for an output port, then no message is sent on this output port.

If no transition can be taken in a given state, then the behavior of the component is completely unspecified ("chaotic"). This implies that we have to handle undesired input explicitly. This is in contrast to SDL, where input messages without a matching transition are implicitly ignored. Although at first sight the SDL approach might seem more convenient, this implicit assumption often leads to subtle errors in SDL specifications.

Note that time stamping the input can be avoided, if the input transition of the receiver is synchronous with the output transition of the sender. This assumption however, would change the communication paradigm. In contrast, our time-stamped model is fully asynchronous and contains the SDL and the ROOM models as a particular case, in which all timing constraints equal `true`. This allows for a modular specification formalism along the lines of [BDD+93, BS97, GS96], and for a stepwise development process, where timing constraints are omitted in the first step [GKRB96] and introduced gradually in the next steps. The above would not hold, however, if we would change our model such that transitions are taken as soon as they become enabled. Moreover, in this case one would also have to introduce prophecies in order to express an arbitrary delay within a given interval.

Our paper is organized as follows. In Section 2, we intuitively introduce timed state transition diagrams by specifying a simplified telephone handler. In Section 3 we introduce the abstract syntax of timed state transition diagrams. Section 4 defines the semantics of TSTDs in terms of timed input-/output relations. Section 5 is concerned with the syntax and the semantics of input- and output patterns. In section 6 we discuss the composition of TSTDs. Finally, in section 7 we draw some conclusions.

## 2 Dialing a Telephone Number

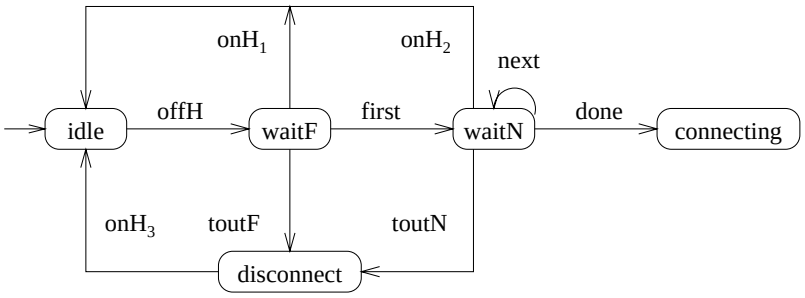
In order to get more intuition about our model and to show how we deal with timeouts, let us formalize the behavior of a telephone handler, which controls the dialing of a four-digit telephone number. The telephone handler has to satisfy the following requirements:

1. *After the caller has lifted the phone receiver, he must receive a dial tone within 0.1s*
2. *If the caller does not respect the following timing requirements, then the telephone handler should return a timeout tone and disconnect the line:*
   (a) *After receiving the dial tone, the caller must dial the first digit within 30s.*
   (b) *After a digit has been dialed, the next digit must be dialed within 20s.*
   (c) *After the caller has lifted the phone receiver, he must dial a complete number within 60s*

The specification of the telephone handler is given in Figure 4.

**tstd** `TelHandler` = {

**input** `i : TelI`
**output** `o : TelO`
**attributes** `n, c, dt, nt : Nat`
**transitions**




| transition | precondition | input | output | postcondition |
|---|---|---|---|---|
| offH | now − t < 0.1s | i?offH @ t | o!dtB | dt' = nt' = now |
| first | t − dt < 30s | i?d(a) @ t | o!dtE | n' = a ∧<br>c' = 1 ∧<br>dt' = now |
| next | t − dt < 20s ∧<br>t − nt < 60s ∧<br>c < 3 | i?d(a) @ t | | n' = n ∗ 10 + a ∧<br>c' = c + 1 ∧<br>dt' = now |
| done | t − dt < 20s ∧<br>t − nt < 60s ∧<br>c = 3 | i?d(a) @ t | | n' = n ∗ 10 + a |
| onH1 | now − t < 0.1s | i?onH @ t | o!dtE | |
| onH2 | t − dt < 20s ∧<br>t − nt < 60s | i?onH @ t | | |
| onH3 | | i?onH | o!dcE | |
| toutF | now − dt = 30s | i?⊖ | o!dcB | |
| toutN | now − dt = 20s ∨<br>now − nt = 60s | i?⊖ | o!dcB | |

}

**Fig. 4.** The telephone handler

The specification consists of an interface declaration part, an attribute declaration part and a state transition diagram. The interface declaration lists the names of the input and the output ports together with their types. The attributes are defined by their name, type, and (optionally) an initial value. The telephone handler has an input port `i : TelI` and an output port `o : TelO`, where the message sets `TelI` and `TelO` are defined as follows:

```
data TelI = offH | onH | d(Digit)
data TelO = dtB  | dtE | dcB | dcE
```

The above types define the set of messages allowed to flow between the caller and the telephone handler. Each message consists of a message name and optional data. For example, `offH` contains no data while `d(8)` contains both the message name `d`(igit) and the data value $8 \in$ `Digit`, where `Digit` is assumed to range between 0 to 9. The bar notation is used to separate alternatives. It is similar to data-type declarations as they are used in functional languages like ML (see [Pau91]).

The abbreviations `onH, dtE, dcB` and `dcE` are used for on-hook, dial-tone-end, disconnect-begin and disconnect-end respectively.

The attribute `n` is used to store the telephone number, the attribute `c` is used to count the number of digits already received, the attribute `nt` is used to save the arrival time of the first digit and the attribute `dt` is used to save the arrival time of the previous digit.

To enhance the readability of the state transition diagrams we allow to define the transitions separately in tabular form and to refer them by their name. A good practice is to use for the transition's name the name (or a common attribute) of the transition's input message(s). For example, the transition `offH` is defined as follows: if the component is in the control state `idle` and the first message in its input port `i` is `offH`, then it sends the message `dtB` along its output port `o` provided that `offH` was received at a time `t` such that `now - t` $<$ `0.1s`. Additionally, the current time is saved in `dt` and `nt`. This is needed to raise a timeout if the first digit is not dialed within 30s or if the number is not dialed within 60s. The other transitions are defined analogously.

Note that if no message has arrived on the port `i`, then a timeout transition takes place exactly at the moment when the time limit has expired. Contrast this with SDL and ROOM, where a timeout message is queued as an ordinary message after the timeout has expired. As a consequence, it can be only guaranteed that timeout processing does not happen before the time limit has expired. Contrast this also with timed automata, where the input is not inspected in the timeout transitions. This is not necessary in that approach, because the correct traces of the environment *and* the automaton are generated together.

Since we use transition diagrams primarily for specification purpose, the preconditions and the postconditions are allowed to be arbitrary predicates. However, often one might be interested in a specification technique which can be used for

validation purposes or for direct code generation. In such cases, one is free to use some restricted form of predicates, such as equations or executable statements of some programming language. Let us now give the formal definition of the state transition diagrams introduced so far.

## 3 Timed State Transition Diagrams

A real-time component described by a timed state transition diagram communicates with its environment along typed input and output ports. They define the interface of the component and are given formally by a *port signature*.

**Definition 1.** (Port signature) Let $I$ and $O$ be disjoint sets of *input* and respectively *output port names*. Let $D$ be a mapping assigning to each port $c \in I \cup O$ a type $D_c$. A *port signature* is a tuple $\Sigma = (I, O, D)$.

Let $\varphi$ denote the time-stamped sequence of message which is expected to arrive along the input interface before a transition is taken. Since the input ports are named and typed, $\varphi$ is a record (or named tuple), and for each $i \in I$, $\varphi.i$ represents the sequence of messages which is expected to arrive along the port $i$. Formally[2], $\varphi$ is an element of the named product $\Pi_{i \in I}(D_i \times \mathbb{N})^*$, where $\mathbb{N}$ denotes the domain of time stamps. In the following, we denote this set of *input actions* by $Act_\Sigma^I$. Similary, let $\psi$ denote the sequence of messages which are sent along the output interface when a transition is taken. Since the output ports are named and typed and the output sequences are sent on all ports in the same time unit, $\psi$ is an element of the set of *output actions* $Act_\Sigma^O = \Pi_{o \in O} D_o^*$. This powerful concept of input and output actions often allows us, in contrast to SDL, to eliminate trivial intermediate states and transitions.

**Definition 2.** (State Transition Diagram) A state transition diagram is a tuple $\mathcal{D} = (\Sigma, S, G, \nu, \eta)$ where:

$\Sigma = (I, O, D)$ is a port signature.

$S = \Pi_{a \in A} D_a$ defines the data state space of $\mathcal{D}$. Each $a \in A$ is an *attribute* $a$ with associated type $D_a$. One special, read only attribute $\texttt{now} \in \mathbb{N}$ contains at each moment the current time.

$G = (N, E \subseteq N \times N, n_0)$ is the *control graph* of $\mathcal{D}$. Each node $n \in N$ defines a *control state* and each edge $(n_1, n_2) \in E$ defines a *control transition*. $n_0 \in N$ is the *initial control node*.

$\nu \in N \to (S \to \mathbb{B})$ is the *node labeling* of $\mathcal{D}$. It marks each control state $n \in N$ with a predicate $\nu_n$ giving the associated data states.

[2] Given an arbitrary set $M$, we denote by $M^*$ the set of finite sequences over $M$. Given a set of names $I$ and a mapping $D$ assigning to each name $i \in I$ a type $D_i$, we denote by $\Pi_{i \in I} D_i$ the set of named tuples $\{f : I \to \bigcup_{i \in I} D_i | f.i \in D_i\}$

$\eta \in E \to \wp((S \times Act^I_\Sigma \to \mathbb{B}) \times (S \times Act^I_\Sigma \times Act^O_\Sigma \times S \to \mathbb{B}))$ is the *edges labeling* of $\mathcal{D}$. It marks each edge $e \in E$ with a set of precondition/postcondition pairs $(pre, post)$. The *precondition* acts as a guard on the transition's source state and input. The transition is taken only if the guard is true. In that case the *postcondition* defines the next state and the output, possibly by referring to the current state and the input.

Although state predicates $\nu_n$ were not used in our telephone protocol, they are convenient to impose for example an upper bound timing constraint for a given control node $n$. If all transitions leaving node $n$ have only lower bounds, then the upper bound constraint for $n$ enforces that one of the transitions is taken before the upper bound is reached.

From a methodological point of view, it might be appropriate to require certain well-formedness conditions for the predicates involved. For example, the enabledness of a transition should depend only on the precondition, i.e., the postcondition should not constrain the current state and the input. Formally, for all data states $s \in S$, inputs $i \in Act^I_\Sigma$, edges $e \in E$ and precondition/postcondition pairs $(pre, post) \in \eta_e$:

$$pre(s, i) \quad \Rightarrow \quad \exists s', o.\ post(s, i, o, s')$$

A state transition diagram satisfying this property is called *precondition controlled*. Similarily, the destination data state $s'$ should be a valid state of the destination control node. We call such a node *postcondition complete*. Formally, for all states $s, s' \in S$, inputs $i \in Act^I_\Sigma$, outputs $o \in Act^O_\Sigma$, edges $(n, n') \in E$ and precondition/postcondition pairs $(pre, post) \in \eta_{(n,n')}$:

$$pre(s, i) \wedge post(s, i, o, s') \quad \Rightarrow \quad \nu_{n'}(s')$$

However, we do not enforce these conditions in the semantics. They should be treated on the methodological level, either by restricting the language or by automatically generating proof obligations.

## 4 The Semantics of TSTDs

The semantics of a timed state transition diagram is given as a relation between the communication histories along its input and its output ports. For simplicity, the communication histories are defined over a discrete time domain which is taken to be the set of natural numbers $\mathbb{N}$. However, this semantics could easily be extended to a dense time domain [MS96].

As usual in weakly monotonic discrete time semantics [AH92], each complete communication history is an infinite sequence of finite sequences of messages. Each finite sequence contains the messages occuring within the same time unit. Given a set of messages $D$, the set $(D^*)^*$ is the set of *partial* communication histories over $D$ and the set $(D^*)^\infty$ is the set of *complete* communication histories over $D$. In the following we abbreviate $(D^*)^\infty$ by $D^\aleph$.

The ports of a component in our semantic model are named and typed. As a consequence the communication histories over its input and respectively output ports are named products of communication histories. We call them *named communication histories*. Formally, if the port signature of the timed state transition diagram is $\Sigma = (I, O, D)$, then the set of complete named communication histories over the input and the output ports are given by $\prod_{i \in I} D_i^{\aleph}$ and $\prod_{o \in O} D_o^{\aleph}$, respectively. The set of partial named communication histories is given by $\prod_{i \in I} (D_i^*)^*$ and $\prod_{o \in O} (D_o^*)^*$. The named communication sequences within a time unit are denoted by $\prod_{i \in I} D_i^*$ and $\prod_{o \in O} D_o^*$. In the following we also refer to named communication histories as communication histories when no confusion arises.

Given $\alpha \in D^{\aleph}$ and $i \in \mathbb{N}$, then $\alpha{\downarrow_i} \in (D^*)^*$ denotes the partial communication history consisting of the first $i$ finite sequences in the complete communication history $\alpha$. This operation is overloaded to named communication histories and to sets of named communication histories in a point-wise and an element-wise style, respectively.

The input/output relation corresponding to a TSTD is a *set valued function*

$$F : \prod_{i \in I} D_i^{\aleph} \to \wp(\prod_{o \in O} D_o^{\aleph})$$

mapping complete input histories to sets of complete output histories. However, not every relation with this functionality is adequate to give the semantics of a TSTD: *In reality, TSTDs can not predict the future.* The output produced by a TSTD until some point in time must not depend on input the TSTD will receive in the future. This condition is formally captured by the following definition:

**Definition 3.** (Timed relations) We call an input/output relation $F \in \prod_{i \in I} D_i^{\aleph} \to \wp(\prod_{o \in O} D_o^{\aleph})$ *weakly time guarded,* if for all $\alpha, \beta : \prod_{i \in I} D_i^{\aleph}$ and $i \in \mathbb{N}$

$$\alpha{\downarrow_i} = \beta{\downarrow_i} \quad \Rightarrow \quad F(\alpha){\downarrow_i} = F(\beta){\downarrow_i}$$

We call an input/output relation *strongly time guarded*, if the following stronger condition holds

$$\alpha{\downarrow_i} = \beta{\downarrow_i} \quad \Rightarrow \quad F(\alpha){\downarrow_{i+1}} = F(\beta){\downarrow_{i+1}}$$

Before we give the formal definition of the input output relation generated by a TSTD, let us introduce some operators on sequences which are used in this definition.

For any finite sequences $s, s_1, s_2$ and element $a$, $a :: s$ is the sequence $s$ with $a$ appended in front of $s$, $s_1 \frown s_2$ is the concatenation of $s_1$ and $s_2$ and $\#s$ is the length of $s$. $[]$ is the empty sequence and $[a_1, \ldots, a_n]$ is the sequence $a_1 :: (a_2 :: \ldots (a_n :: []))$. If $a_1 = a_2 = \ldots = a_n$ we also write $[a^n]$. The operations $::$ and $\frown$ are overloaded for the case in which the second argument is an infinite sequence.

On sequences of sequences, an operation related to $\frown$ is the paste operation $.\!\smile\!. : (D^*)^* \times D^{\aleph} \to D^{\aleph}$. It only differs from the conventional concatenation $\frown$

in that the last sequence of $s_1$ is pasted with the first sequence of $s_2$.

$$\forall a, b \in D^*, s_1 \in (D^*)^*, s_2 \in D^{\aleph}.\ \ (s_1 \frown [a])^{\smile}([b] \frown s_2) = s_1 \frown [a \frown b] \frown s_2$$

This operation allows us to take a prefix of a communication history, by cutting the history in the middle of a sequence of messages occurring in the same time unit. Formally, we say that $\varphi$ is a prefix of $\alpha$, written as $\varphi \sqsubseteq \alpha$ if the following holds:

$$\varphi \sqsubseteq \alpha \quad \Leftrightarrow \quad \exists \beta.\ \ \varphi^{\smile} \beta = \alpha$$

Given a partial communication history $\varphi \in (D^*)^*$ and a time value $t \in \mathbb{N}$, we denote by $\varphi@t \in (D \times \mathbb{N})^*$ the time-stamped communication history, i.e., the communication history with time information made explicit and adjusted to the time point $t$. Formally, for every $a \in D, u \in D^*$ and $\varphi \in (D^*)^*$ the time-stamped sequence $((a :: u) :: \varphi)@t$ is defined as below. Remember that $a$ and all the messages in $u$ occurr in the same time unit.

$$\begin{aligned}((a :: u) :: \varphi)@t &= \langle a, t\rangle :: ((u :: \varphi)@t)\\ ([] :: \varphi)@t &= \varphi@(t+1)\\ []@t &= [\langle \ominus, t\rangle]\end{aligned}$$

If no confusion can arise, a tuple $\langle a, t\rangle$ from a time stamped sequence is also written as $a@t$. In this definition and in the following, we tacitly assume that the symbol $\ominus$ is also an element of $D$. The last element $\langle \ominus, t\rangle$ of a time stamped sequence is needed to distinguish between timed sequences ending with a different number of trailing empty sequences. As we shall see in Section 5, this allows us to give a different semantics to empty- and to negative patterns, respectively. Empty patterns like `i?[]` are satisfied when no message in the input port is consumed, whereas negative patterns like `i?`$\ominus$ are satisfied only if no message has arrived on the input port until `now`.

All operators introduced in this section are overloaded to named communication histories and to sets of communication histories in a point-wise and an element-wise style, respectively[3]. If $\gamma, \delta \in \prod_{i \in I} \mathbb{N}$ and $k \in \mathbb{N}$ then we write $\gamma + \delta$ and $\gamma + k$ for the named product which is obtained by summing $\gamma$ and $\delta$ componentwise and by adding $k$ to each component of $\gamma$ respectively. Moreover, for a state $s$ of the TSTD we write $s + k$ for the state in which `now` is incremented by $k$.

The operational intuition is as follows: The state space of the component consists of a control part (node), a data part (attributes), a time part (`now` variable), and a tuple $f \in \Pi_{i \in I} \mathbb{N}$. The value $f.i$ indicates the arrival time of the first message in the input port $i$ which has not yet been processed by the component. The component starts in an initial data state $s_0$ satisfying the predicate $\nu_{n_0}$ of the start node, and $f.i = (s_0).\mathrm{now}$ for all $i \in I$. In each state, the component can wait for some amount of time until a transition is taken. However, waiting is

[3] Note that $\prod_{i \in I} D_i^{\aleph} \cong (\prod_{i \in I} D_i^*)^{\infty}$. For $\varphi \in \prod_{i \in I} D_i^*$ and $\alpha \in \prod_{i \in I} D_i^{\aleph}$ we therefore consider $\varphi :: \alpha$ to be also an element of $\prod_{i \in I} D_i^{\aleph}$.

only allowed either if no transition was already enabled or if waiting does not lead to a chaotic behavior. The transition consumes part of the input, updates $f$ accordingly, and sends some output in the current time interval. If the component performs a transition, then the transition is instantaneous, i.e., the value of the timer `now` is the same in the next state. In the definition below, we will assume that `now` cannot be modified, i.e., it does not occur primed in the postcondition of the TSTD.

**Definition 4.** (History Semantics of TSTDs) Given a state transition diagram $\mathcal{D} = (\Sigma, S, G, \nu, \eta)$. Then the semantics of $\mathcal{D}$ is the input/output relation $\mathcal{F}$ which is defined as follows:

$$\mathcal{F} \in \prod_{i \in I} D_i^{\aleph} \to \wp(\prod_{o \in O} D_o^{\aleph})$$
$$\mathcal{F}(\alpha) = \bigcup_{s \in \nu(n_0)} F(s, n_0, \lambda i \in I.(s.\texttt{now}))(\alpha)$$

$F$ is the greatest weakly time guarded input/output relation parametric with respect to the current data state, current control state and arrival time of the first messages in the named input communication history that satisfies the following equation:

$$F \in S \times N \times \prod_{i \in I} \mathbb{N} \to \prod_{i \in I} D_i^{\aleph} \to \wp(\prod_{o \in O} D_o^{\aleph})$$

$$F(s, n, f)(\alpha) = \{\ \beta \in \prod_{o \in O} D_o^{\aleph} \quad |$$

| | | |
|---|---|---|
| $\forall k \in \mathbb{N}.$ | $--$ | the delay measured in ticks |
| $\forall m \in N.$ | $--$ | the next control node |
| $\forall \varphi \in \prod_{i \in I} (D_i^*)^*.$ | $--$ | the timed input of the TSTD |
| $\forall \psi \in \prod_{o \in O} D_o^*.$ | $--$ | the output of the TSTD |
| $\forall s' \in S.$ | $--$ | the next state of the TSTD |
| $\forall pre \in S \times Act_\Sigma^I \to \mathbb{B}.$ | $--$ | the transition's precondition |
| $\forall post \in S \times Act_\Sigma^I \times Act_\Sigma^O \times S \to \mathbb{B}.$ | $--$ | the transitions's postcondition |
| | | |
| $(\varphi \sqsubseteq \alpha \wedge$ | $--$ | $\varphi$ is a prefix of $\alpha$ |
| $(pre, post) \in \eta_{(n,m)} \wedge$ | $--$ | (pre,post) is in the TSTD |
| $\nu_n(s) \wedge \nu_m(s') \wedge$ | $--$ | src/dest predicates are satisfied |
| $pre(s{+}k, \varphi@f) \wedge post(s{+}k, \varphi@f, \psi, s')$ | $--$ | pre/post predicates are satisfied |
| $\Rightarrow$ | $--$ | chaos completion |
| $\exists \alpha' \in \prod_{i \in I} D_i^{\aleph}, \beta' \in \prod_{o \in O} D_o^{\aleph}.$ | $--$ | the suffixes of $\alpha$ and $\beta$ |
| $\alpha = \varphi^\frown \alpha' \wedge$ | $--$ | $\alpha'$ is indeed a suffix of $\alpha$ |
| $\beta = [[]^{k-1}, \psi]^\frown \beta' \wedge$ | $--$ | $\beta'$ is indeed a suffix of $\beta$ |
| $\beta' \in F(s', m, f{+}\#\varphi)(\alpha')$ | $--$ | $\beta'$ is in the continuation of $F$ |
| ) | | |
| $\wedge$ | $--$ | no chaos for time: |

$$
\begin{array}{lll}
(\forall k' \leq k. & -- & \text{an enabled transition} \\
\nu_n(s) \wedge \nu_m(s') \wedge & -- & \text{has to be taken} \\
pre(s{+}k', \varphi@f) \wedge post(s{+}k', \varphi@f, \psi, s') & -- & \text{before} \\
\Rightarrow & -- & \text{it becomes disabled} \\
\nu_n(s) \wedge \nu_m(s') \wedge & -- & \text{because of} \\
pre(s{+}k, \varphi@f) \wedge post(s{+}k, \varphi@f, \psi, s') & -- & \text{time progression} \\
)
\end{array}
$$
}

As usual we define that $F_1 \subseteq F_2$ holds if $\forall \alpha. F_1(\alpha) \subseteq F_2(\alpha)$. Since the $F$ occurs only positively on the right-hand side of the equation, the corresponding functional is monotone w.r.t. this ordering, which implies the existence of a greatest solution. By the greatest weakly time guarded input-/output relation we mean the relation which is obtained by removing all behaviors which are not weakly time guarded in this solution.

## 5 Pattern Syntax of Transitions

The concrete syntax for state transition diagrams, and for the underlying predicate logic may depend on the concrete objectives for which timed state transition diagrams are used, and on the available tool support. One possible syntax has been given in the telephone protocol example.

However, we found it convenient to use transition rules of the following form:

$$\{\, Pre \,\}\ ip \ /\ op\ \{\, Post \,\}$$

where *Pre* and *Post* are *predicate expressions* and *ip* and *op* are *input* and *output patterns* (see the next two sections). The precondition *Pre* may contain as free variable the current state $s \in \Pi_{a \in A} D_a$ and the variables occuring free in the input pattern. The postcondition *Post* may contain as free variables $s$ and the next state $s'$, as well as the variables occuring free in the input and output patterns. Since attributes not mentioned explicitly primed in the postcondition *Post* are assumed to remain unchanged, we denote by $\overline{Post}$ the conjunction of *Post* with equations $s'.a = s.a$ for all these attributes.

Suppose that $\rho$ and $\eta$ are environments for the free variables contained in the input and respectively the output patterns, that $\varphi$ denote the current input action, and that $\psi$ denotes the current output action. Then the predicates *pre* and *post* with respect to *Pre*, *Post*, *ip* and *op* are defined as follows:

$$
\begin{array}{lll}
pre(s, \varphi) & = \exists \rho : & [\![Pre]\!]_{s,\rho} \wedge [\![ip]\!]_{s,\rho,\varphi} \\
post(s, \varphi, \psi, s') & = \exists \rho, \eta : & [\![ip]\!]_{s,\rho,\varphi} \wedge [\![\overline{Post}]\!]_{s,s',\rho,\eta} \wedge [\![op]\!]_{s,s',\rho,\eta,\psi}
\end{array}
$$

Here, $[\![PredExp]\!]_{env}$ denotes the interpretation (i.e. a truth value) of the predicate expression *PredExp* with respect to some environment *env*. We assume the

interpretation function to be given as usual for predicate expressions. The definition of the input and the output patterns together with their interpretations is the subject of the next two sections.

### 5.1 The Input Patterns

The input patterns are used to simplify the definition of the precondition/postcondition. An *input pattern* has the following form:

$$ip ::= i_1?p_1, \ldots, i_n?p_n$$
$$p \ ::= [m_1[@t_1], \ldots, m_k[@t_k]] \mid \ominus$$

where the port names $i_k$ are all distinct. It associates each input port name $i_k$, $1 \leq k \leq n$, with an *input port pattern* $p_k$. The port pattern $[m_1[@t_1], \ldots, m_k[@t_k]]$ tests for the presence of the message sequence $[m_1, \ldots, m_k]$ on the associated port. Each message $m_k$ may be optionally *time stamped* with the time of the message arrival. The empty pattern [] indicates that input is ignored on the associated port. To simplify the diagrams, input port patterns with empty patterns are not explicitly written, and one-element patterns like $i?[m]$ are written as $i?m$. The port pattern $\ominus$ tests for the absence of any message on the associated port. It allows us to specify priorities and to model timeouts and interrupts. Note again the subtle difference between the empty pattern [] and the negative pattern $\ominus$. [] expresses that input is not required for the transition, while $\ominus$ expresses that there is no input.

Given the current state $s$, an environment $\rho$ for the free variables in $ip$ and an input sequence $\varphi$ we define the interpretation of an input pattern

$$i_1?p_1, \ldots, i_m?p_m$$

as below. Without loss of generality, we assume that $I = \{i_1, \ldots, i_m, \ldots\}$.

$$[\![i_1?p_1, \ldots, i_m?p_m]\!]_{s,\rho,\varphi} \stackrel{\text{def}}{=} \forall_{k \leq m} : [\![i_k?p_k]\!]_{s,\rho,\varphi} \wedge \forall_{k>m} : [\![i_k?[]]\!]_{s,\rho,\varphi}$$

$$[\![i?[m_1[@t_1], \ldots, m_k[@t_k]]]\!]_{s,\rho,\varphi} \stackrel{\text{def}}{=} [[\![m_1@t_1]\!]_{s,\rho}, \ldots, [\![m_k@t_k]\!]_{s,\rho}, [\![\ominus@t]\!]_{s,\rho}] = \varphi.i$$

$$[\![i?\ominus]\!]_{s,\rho,\varphi} \stackrel{\text{def}}{=} [\ominus@s.\text{now}] = \varphi.i$$

For messages not explicitly time stamped in the input patterns as well as for the negative pattern $\ominus$, we assume the existence of fresh, anonymous time variables in the environment $\rho$.

For simplicity, all patterns in the example in Section 2 have used only one-element lists. However, it is often convenient to use the more general form of patterns introduced in this section. For instance, the requirement that the time between the dialing of two consecutive digits is limited to 20s can be expressed as follows:

$\{\forall \mathtt{i}.1 \leq \mathtt{i} \leq 3.\, \mathtt{t}_{i+1}\text{-}\mathtt{t}_i < 20\mathtt{s}\ \}\ \mathtt{i?[d(a_1)@t_1,\ d(a_2)@t_2,\ d(a_3)@t_3,\ d(a_4)@t_4]}$

### 5.2 The Output Patterns

The output patterns are used to simplify the definition of the postcondition. An *output pattern* has the following form:

$$op ::= o_1!p_1, \ldots, o_n!p_n$$
$$p \ ::= [m_1, \ldots, m_k]$$

where the port names $o_k$ are all distinct. It associates each output port name $o_k$, $1 \leq k \leq n$, with an *output port pattern* $p_k$. The port pattern $[m_1, \ldots, m_k]$ defines the message sequence on the associated output port.

Given two states $s$ and $s'$, environments $\rho, \eta$ for the free variables, and an output sequence $\psi$, we define the interpretation of an output pattern

$$o_1!p_1, \ldots, o_m!p_m$$

as below. Without loss of generality, we assume that $O = \{o_1, \ldots, o_m, \ldots\}$.

$$[\![o_1!p_1, \ldots, o_m!p_m]\!]_{s,s',\rho,\eta,\psi} \stackrel{\text{def}}{=} \forall_{k \leq m} : [\![o_k!p_k]\!]_{s,s',\rho,\eta,\psi} \ \wedge \ \forall_{k>m} : \psi.o_k = []$$

$$[\![o![m_1, \ldots, m_k]]\!]_{s,s',\rho,\eta,\psi} \stackrel{\text{def}}{=} [[\![m_1]\!]_{s,s',\rho,\eta}, \ldots, [\![m_k]\!]_{s,s',\rho,\eta}] = \psi.o$$

## 6 Composition

Given two timed state transition diagrams $tstd_1$ and $tstd_2$ with port signatures $\Sigma_1 = (I_1, O_1, D_1)$ and $\Sigma_2 = (I_2, O_2, D_2)$, respectively. If $O_1 \cap O_2 = \emptyset$ and for all $i \in (I_1 \cap O_2) \cup (I_2 \cap O_1)$ it holds that $(D_1)_i = (D_2)_i$, then we call the two port signatures *compatible*. For compatible signatures $\Sigma_1$ and $\Sigma_2$, the composition $\Sigma_1 \otimes \Sigma_2$ is defined as follows: each output port of $\Sigma_1$ is connected with an input port of the same name of $\Sigma_2$, and similarily, each output port of $\Sigma_2$ is connected to an input port of $\Sigma_1$. The *feedback channels* introduced this way are hidden in $\Sigma_1 \otimes \Sigma_2$. Formally, $\Sigma_1 \otimes \Sigma_2 = (I, O, D)$, where[4]

$$I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1) \quad O = (O_1 \setminus I_2) \cup (O_2 \setminus I_1) \quad D = D_1 + D_2$$

The resulting network is graphically depicted in Figure 5.

[4] Given two type mappings $(D_i)_{i \in I}$ and $(D'_j)_{j \in J}$, their *sum* is defined such that $(D + D').i = D.i$ if $i \in I$, and $(D + D').i = D'.i$ if $i \in J$. Given two named tuples $\varphi \in \Pi_{i \in I} D_i$ and $\psi \in \Pi_{j \in J} D'_j$, and $K \subseteq I$, their *sum* $\varphi + \psi$ is defined such that $(\varphi + \psi).i = \varphi.i$ if $i \in I$, and $(\varphi + \psi).i = \psi.i$ if $i \in J$. The projection $\varphi|_K$ is defined such that $(\varphi|_K).i = \varphi.i$ for all $i \in K$.

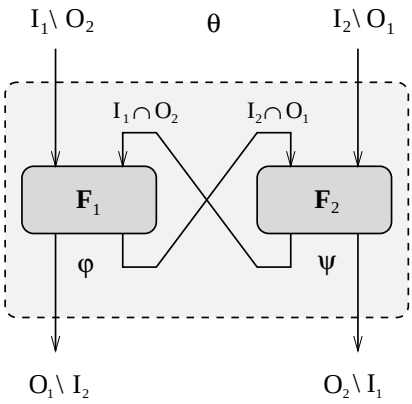



**Fig. 5.** Composition

Let the semantics of $tstd_1$ and $tstd_2$ with compatible port signatures be given by

$$[\![tstd_1]\!] : \prod_{i\in I_1} (D_1)_i^{\aleph} \to \wp(\prod_{i\in O_1} (D_1)_i^{\aleph}),$$
$$[\![tstd_2]\!] : \prod_{i\in I_2} (D_2)_i^{\aleph} \to \wp(\prod_{i\in O_2} (D_2)_i^{\aleph}).$$

The semantics $[\![tstd_1 \otimes tstd_2]\!]$ of their composition is equal to the *parallel composition with feedback* $[\![tstd_1]\!] \otimes [\![tstd_2]\!]$ of their input-/output-relations. The operator $\otimes$ is defined for relations as follows:

$$[\![tstd_1]\!] \otimes [\![tstd_2]\!] : \prod_{i\in I} D_i^{\aleph} \to \wp(\prod_{i\in O} D_i^{\aleph})$$

$$([\![tstd_1]\!] \otimes [\![tstd_2]\!])(\alpha) =$$

$$\{\beta|_O \in \prod_{i\in O} D_i^{\aleph} |\ \beta|_{O_1} \in [\![tstd_1]\!](\alpha|_{I_1} + \beta|_{I_1}) \wedge \beta|_{O_2} \in [\![tstd_2]\!](\alpha|_{I_2} + \beta|_{I_2})\}$$

For the above equation a unique solution exists if both input-/output relations are time guarded on the feedback channels [GKR96, BS97].

## 7 Conclusion

We have presented timed state transition diagrams, a new formalism for the specification of real-time aspects of reactive systems. The formalism is based on a semantic model where components communicate asynchronously via unbounded FIFO-channels. We have shown that existing approaches using this communication paradigm, like SDL and ROOM, fail to model real-time aspects accurately. The basic idea of our formalism is to time-stamp messages with their arrival time and to allow the specifier to label transitions with timing constraints. By using negative patterns, we can also easily deal with priorities. Therefore, timed

state diagrams can be seen as a powerful extension of SDL and ROOM. In particular, it allows for a stepwise development process, where timing constraints are omitted in the first step and introduced gradually in the next steps. Our approach is fully modular, since it is based on FOCUS, a formal approach for the specification and refinement of reactive systems [BDD+93, BS97, GS96].

Among other formalisms for real time systems, timed input/output automata [LV96, LSVW95] seem to be closest to our approach. However, whereas the main goal of timed input/output automata is to provide a semantical model for real time systems, our main concern was to provide a pragmatic and tractable specification formalism for real time systems.

Timed state transition diagrams can be extended in several ways. A first extension would be to use a dense time model. We believe that such an extension can easily be accomplished along the lines of [MS96]. Dense time models are gaining more and more attention in theory and practice for the modeling of real-time systems. Of great importance is also the development of a tractable refinement calculus, as it has been presented for a similar, time-independent notation in [RK96]. Future extensions will also include an extension to hierarchical state transition diagrams à la statecharts [Har87]. The semantic foundations needed for hierarchical AND-states have already been presented in this paper. In constrast to the various approaches under way to define a semantics for statecharts [Von94], we start from a semantic model and define an appropriate notation for specifying components.

## Acknowledgments

We thank Jan Philipps, Ursula Hinkel and Christian Prehofer for reading a draft version of the paper.

## References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 2(126):183–235, April 1994.

[AH92] R. Alur and T.A. Henzinger. Logics and models of real time: a survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 74–106. Springer-Verlag, 1992.

[BB91] F. Bause and P. Buchholz. Protocol analysis using a timed version of SDL. In J. Quemada, J. Mañas, and E. Vazquez, editors, *Formal Description Techniques*. North Holland, 1991.

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber. The Design of Distributed Systems - An Introduction to

FOCUS. Technical Report SFB 342/2/92 A, Technische Universität München, Institut für Informatik, 1993.

[BS97] M. Broy and K. Stølen. *Interactive System Design*. To appear, 1997.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SYSLAB system model with state. TUM-I 9631, Technische Universität München, 1996.

[GKRB96] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State transition diagrams. TUM-I 9630, Technische Universität München, 1996.

[GS96] Radu Grosu and Ketil Stoelen. A Model for Mobile Point-to-Point Dataflow Networks without Channel Sharing . In Martin Wirsing and Maurice Nivat, editors, *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology, AMAST'96, Munich, Germany*, pages 505–519. Lecture Notes in Computer Science 1101, 1996.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[Hin96] Ursula Hinkel. SDL and Time – A Mysterious Relationship. 1996. submitted to SDL Forum 97.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science*, pages 394–406. IEEE Computer Society Press, 1992.

[Hoa85] C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall International series in computer science. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[IT93] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL)*. ITU-T, Geneva, 1993.

[IT96] ITU-T. *Z.120 – Message Sequence Chart (MSC)*. ITU-T, Geneva, 1996.

[Leu95] S. Leue. Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach. In *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification PSTV'95*. Chapmann & Hall, 1995.

[LSVW95] N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. Technical Report CS-R9578, CWI, Computer Science Department, Amsterdam, 1995. Also appeared in: Hybrid Systems III, Lecture Notes in Computer Science. Available under `http://www.cs.kun.nl/~fvaan/`.

[LV96] N.A. Lynch and F. Vaandrager. Forward and backward simulations – part II: Timed systems. *Information and Computation*, 128(1):1–25, 1996.

[MS96] Olaf Müller and Peter Scholz. Specification of real-time and hybrid systems in FOCUS. TUM-I 9627, Technische Universität München, 1996.

[OFMP+94] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92.* Elsevier Science, North-Holland, 1994.

[Pau91] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[RK96] B. Rumpe and C. Klein. Automata describing object behavior. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286, Norwell, Massachusetts, 1996. Kluwer Academic Publishers.

[SGW94] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley and Sons, Inc., 1994.

[Von94] M. Von der Beeck. A Comparison of Statecharts Variants. *Lecture Notes in Computer Science*, 863:128–148, September 1994.

This article was processed using the LaTeX macro package with LLNCS style