T U M

I N S T I T U T F Ü R I N F O R M A T I K

The Specification of System Components by State Transition Diagrams

Manfred Broy

TUM-I9729
Mai 97

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-05-I9729-300/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

The Specification of System Components by State Transition Diagrams*)

Manfred Broy
Institut für Informatik
Technische Universität München
D-80290 München

Abstract

This technical memo provides a syntactic and semantic basis for state transition diagrams (STDs) as they are used for the description of state transition machines (STMs) with input and output. STMs serve for the specification of system components. We work with STDs with transition rules labelled by input and output patterns and pre- and postconditions. We extend our notation to support specifications that deal with the timing of input and output as well. In particular, we work out the following concepts

- the semantic model of STMs with input and output,
- the semantic model of STDs in terms of predicate logic,
- the description of STMs by STDs,
- the definition of stream processing functions by STMs,
- a syntax for STDs and their labels.

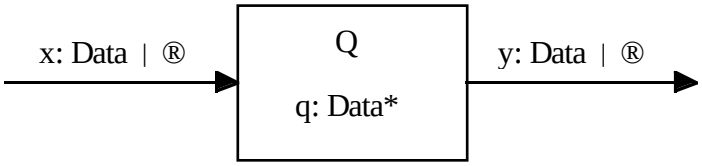
In contrast to approaches like statecharts (see [Harel 87]) we rather start from a semantic notion of a STM and then develop a tuned graphical description technique for it. We show also some methodological aspects such as the use of a partitioning of nodes in STDs as a refinement step that leads to STDs with independent transactions for input and output. We briefly discuss hierarchical STDs, time-outs, interrupts, and pre-emption.

*) This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen" and the industrial research project SysLab sponsored by Siemens Nixdorf and by the DFG under the Leibniz program.

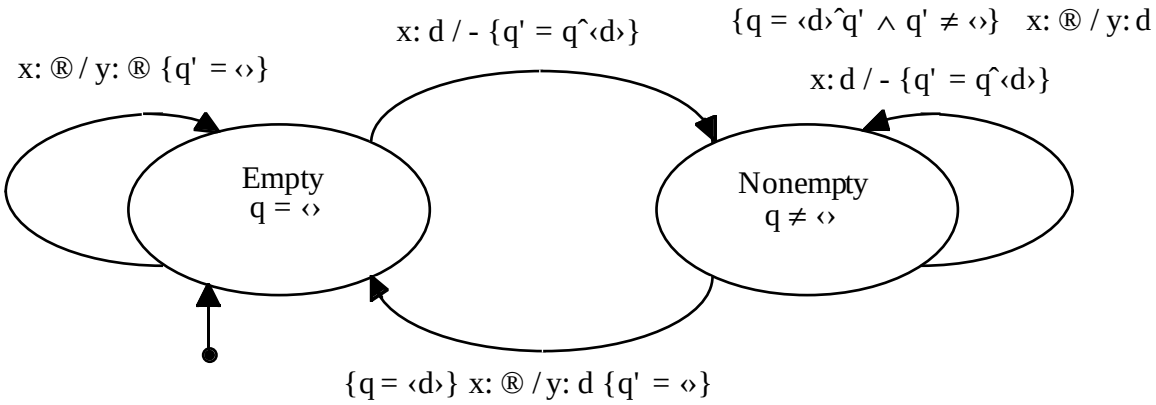
1. Introduction

The description of the behaviour of reactive systems by state transition systems as well as by state transition diagrams appears as an idea in many approaches in computing science. However, there exist a number of significant differences between the various approaches, ranging from attempts to give a visual formalism for the description of systems such as state charts (see [Harel 87]) or the system description language SDL (see [SDL 88]) to approaches with a much more logical, formal description as in the work of Lamport on TLA (see [Abadi, Lamport 88, 90]) or the I/O-automata of Lynch, Tuttle, and Stark (see [Lynch, Tuttle 87, 89] [Lynch, Stark 89], and [Lynch, Vaandrager 95]).

Typically, visual formalisms exhibit a number of unsolved problems with respect to the cleanness and preciseness of their semantics. So often, although visual formalisms are rather suggestive in a first sight, by a closer look it turns out that a number of fundamental semantic questions are not solved at all and remain unclear. On the other hand, the more logical formalisms like TLA of Leslie Lamport or the I/O-automata of Nancy Lynch use rather sophisticated logical concepts including temporal logic or tricky fairness assumptions. This makes it often unnecessary difficult for engineers, who in general prefer visualised presentations of behaviour descriptions, to work with these approaches.



(a) The Component Q as a Data Flow Node with State Attribute



(b) State Transition Diagram

Fig. 0 Description of an Interactive Queue by a Data Flow Node and a STD (Data is the sort of data elements, ® is the request signal)

In this paper we are interested in a good compromise between semantic rigor and suggestive presentation. With this goal, we develop a graphical formalism for the description of the behaviour of system components which is based on a rigorous mathematical semantics. Strictly speaking, the graphical formalisms are only syntactic sugar for writing logical formulas. This allows us to work out system descriptions using the graphical formalism and to generate from

these documents logical formulas that allow us logical manipulations and to prove properties about the specified components. Furthermore, if the description follows particular simple rules we can generate even executable prototypes as a means to experiment with the behaviours.

We work with *state transition diagrams* (STDs) as a graphical description technique for specifying *state transition machines* (STMs). As a semantic and methodological basis we use FOCUS (see [FOCUS 92]). It provides an extremely powerful, mathematical model for distributed, concurrent, interactive, real-time systems. It furthermore provides a comprehensive class of concepts for the logical specification, refinement, and verification of interactive and reactive systems. The black box behaviour (the "abstract semantics") of STMs is described in terms of the FOCUS system model.

The mathematical and logical style of the syntax of FOCUS is not always very well-suited for a practical use, however, since engineers in practice are not familiar with and not fond of large logical formulas that arise if an untuned logical formalism is used. Moreover, often properties of systems have to be written by too lengthy, unintelligible, intricate formulas. Therefore, we need a better tuned, more suggestive description formalism. In the following, we use the semantic and syntactic basis for the description of FOCUS components and composed systems as the basis of STDs.

We start with a simple motivating example of a STD. We do not explain it in detail but only use it to demonstrate our techniques. We define a simple interactive queue as shown in Fig. 0. Fig. 0 shows in part (a) the queue as a data flow node with its channel names and sorts. In part (b) it describes the behaviour by a STD. Its internal data state is given by the values of the attribute q which are sequences of data as indicated in (a). The nodes of the STD stand for sets of states. The transitions show pre- and postconditions as well as patterns of incoming and outgoing messages. A precise description of their meaning is given later.

Our paper is structured as follows: section 2 defines the concept of STMs with input and output. Section 3 defines the mathematical concepts for STDs. Section 4 introduces stream processing functions to represent the behaviour of components. Section 5 defines how to relate stream processing functions to STDs. Section 6 defines the syntax of STDs. Section 7 relates the syntax to the mathematical notion of a STD. Section 8 gives a number of examples. Section 9 introduces STDs that allow us to specify also the timing of messages. Section 10 deals with multithreaded STDs and section 11 with hierarchical STDs. This way we demonstrate how the basic semantic concept of STMs can be represented by STDs and how this notation can be extended to time, hierarchy, and parallelism. After the conclusion we treat the alternating bit protocol as an extended example in the appendix.

2. System Components as State Transition Machines with Input and Output

The mathematical system model of FOCUS is used to describe interactive and reactive system components. A component is connected to the outside world by communication channels. Its description is based on the following sets that determine its syntactic interface:

- I set of input channels (each of which has assigned an individual sort),
- O set of output channels (each of which has assigned an individual sort),
- M set of messages (in the theoretical parts of the paper we work for simplicity with the same uniform message for each channel, in the examples we work with specific message sets for each channel by providing an individual sort for each channel).

These elements define the syntactic interface of a component as shown in Fig. 1. For simplicity, we do not work with sorted channels in the theoretical part of this paper but use for every channel a universal sort M. This keeps the formulas more readable. In examples and applications we work with sorted channels, however. The theoretic parts can easily be extended to the multisorted case, however, this way the formulas get more lengthy and less comprehensible.

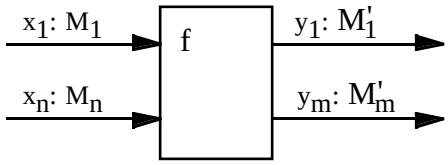


Fig. 1 A Data Flow Node Describing the Syntactic Interface of a Component

A *communication pattern* for a set of channels C is represented by a mapping p: $C \to M^*$, which assigns a finite sequence of messages to every channel in the set C. We write $C^{\vec{*}}$ for the set of valuation functions $C \to M^*$. By M* we denote the set of finite sequences over the set M.

A mathematical system model based on states uses state transitions to describe the allowed state changes. It includes in addition to the sets of sorted I/O-channels in the syntactic interface of the component listed above a set

State

which denotes the set of states (the state space) of the component. We define the behaviour of a STM by a state transition function

$$\Delta: \text{State} \times I^{\vec{*}} \to (\text{State} \times O^{\vec{*}} \to \text{Bool})$$

and a nonempty set

$$\text{State}_0 \subseteq \text{State}$$

of initial states. A STM with transition function Δ works operationally as follows. Given the input pattern i the predicate $\Delta(\sigma, i)$ characterises all pairs (σ', o) of states σ' that may be reached from state σ when the input pattern i is available issuing output as described by the output pattern o.

A STM is nondeterministic, in general. In each transition step for a state σ and it accepts a communication pattern i of its input streams and produces a successor state σ' a communication pattern, o for its output streams. These pairs (σ', o) are chosen nondeterministically from a set of pairs of states and output patterns. For this kind of STMs we represent the set of possible updated states and outputs of a transition by a predicate. Of course, sets of pairs of states and output patterns can be used here instead of predicates as well.

3. State Transition Diagrams

In this section we introduce the mathematical concept of a logical interpretation of *state transition diagram* (STD). A first example of a STD is given in Fig. 0. Without dealing with the

particular form of the syntax of a STD we introduce a number of predicates that are defined by a STD. We use a STD to describe a state transition machine (STM).

A STD is syntactically a finite directed graph that consists of a set K of nodes k and a finite number of arcs (called *transitions*) between the nodes. The nodes are labelled by identifiers and by formulas (called the *node predicates*) and the arcs are labelled by certain terms called *transition rules*.

For each node $k \in K$ of the STD a finite number of state transition rules with the node k as their source is defined that way. Each transition has a node as its source and a node as its target. A nonempty set of the nodes of the STD is marked to be initial. We use an arc without a transition rule and without a source to indicate the initial nodes. Mathematically, we work with a predicate

$$\text{initial}: K \rightarrow \text{Bool}$$

that yields true for those nodes that are initial nodes. At least one node has to be marked to be initial. In our introductory example in Fig. 0 the node called Empty is marked as initial.

In addition to the STD we describe a data state space by defining a set of attributes together with their sorts. They can be seen as the program variables. In our introductory example in Fig. 0 we use only one data state attribute denoted by q that is an identifier for a sequence of data elements as indicated in the data flow node in part (a).

The state of the STM described by the STD is decomposed into a *data state* and a *control state*. The control state space is determined by the set of nodes K. The data state space is the set of valuations for the state attributes. Let Data_State be the set of data states which are represented by the set of valuations for the attributes. We define the set of states called State as follows:

$$\text{State} = K \times \text{Data\_State}$$

We interpret a STD by a STM as follows. With each node $k \in K$ we associate a predicate on the data states (this is inspired by [Paech, Rumpe 94])

$$S_k: \text{Data\_State} \rightarrow \text{Bool}$$

that characterises the set of data states represented by the node k. The initial state set State_0 of the STM is defined by all the initial nodes of the STD and is specified by the following equation

$$\text{State}_0 = \{(k, \sigma) \in \text{State}: \text{initial}(k) \wedge S_k(\sigma)\}$$

Each node $k \in K$ has $n_k \in \mathbb{N}$ outgoing arcs that represent transition rules that all have the node k as their source. A *transition rule* j with $1 \leq j \leq n_k$ for the source node k consists of an arc to a target node t(k, j) and a *transition pattern*. The transition starts in the node k or more precisely in a state of the set that is described by the node predicate S_k. It leads to a node t(k, j) or, more precisely, to a state in the set described by the predicate $S_{t(k, j)}$ that characterises the set of states of the target node.

The transition pattern describes the

- precondition expressed in terms of the state attributes under which a transition is enabled,
- the input messages on the input channels needed for a transition,
- the output produced on the output channels in a transition,
- the postcondition, characterizing the data state after execution of the transition.

The precondition und the input messages form the *input pattern*. In the most puristic view the input pattern is the enabling condition such that the transition can be executed if and only if the

enabling condition is valid. A transition pattern consists of an input pattern and an output pattern:

Input Pattern for node $k \in K$ and transition rule j, $1 \le j \le n_k$:

$$Q_k^j : \text{Data\_State} \times \vec{I}^* \to \text{Bool}$$

Output Pattern:

$$R_k^j : \text{Data\_State} \times \vec{I}^* \to (\text{Data\_State} \times \vec{O}^* \to \text{Bool})$$

The semantics of the STD with nodes $k \in K$ with the transition rules $j = 1, ..., n_k$ is given by a state transition function:

$$\Delta: \{(k, \sigma) \in \text{State}: S_k(\sigma)\} \times \vec{I}^* \to (\text{State} \times \vec{O}^* \to \text{Bool})$$

that is specified by the equation

$$\Delta((k, \sigma), i).((k', \sigma'), o) \equiv$$
$$S_k(\sigma) \wedge \exists\, j, 1 \le j \le n_k: t(k, j) = k' \wedge Q_k^j(\sigma, i) \wedge R_k^j(\sigma, i).(\sigma', o) \wedge S_{k'}(\sigma')$$

From a methodological point of view, it is certainly advisable to insist on the puristic view that the enabledness of a transition is solely determined by the input pattern. Then whenever an input pattern is valid a output pattern and a state have to exist that fulfill the output patterns. In mathematical terms in the puristic approach we require

$$S_k(\sigma) \wedge Q_k^j(\sigma, i) \Rightarrow \exists\, \sigma', o: R_k^j(\sigma, i).(\sigma', o) \wedge S_{t(k, j)}(\sigma')$$

This condition guarantees that every state and input for which the input pattern and the precondition applies can be processed. Then transitions may be selected by looking only at the input pattern and the precondition. A STD with this property is called *input pattern enabled*.

In addition, being puristic we might require the condition

$$S_k(\sigma) \wedge Q_k^j(\sigma, i) \wedge R_k^j(\sigma, i).(\sigma', o) \Rightarrow S_{t(k, j)}(\sigma')$$

if we want to be sure that every successor state of a transition rule always fulfils the requirements of the target node. If this condition is fulfilled, then all the states σ' that can be reached according to the proposition

$$R_k^j(\sigma, i).(\sigma', o)$$

fulfil the node predicate $S_{t(k, j)}(\sigma')$ associated with the target node anyhow. Then the transition diagram is called *output pattern complete*.

We do not follow the puristic view in the examples but are rather liberal in the following, however. We do not retain the puristic requirements and do not forbid that the node predicate restricts the set of reachable states in addition to the transition predicate R_k^j . This can make the transition formulas much shorter in certain cases. Note that we also do not require that every state in the target $S_{t(k, j)}$ can actually be reached by a transition. Nevertheless for a detailed analysis of a STD it may be helpful to bring it into a form where the conditions above are fulfilled.

4. Stream Processing Functions as Models of Component Behaviours

In this section we introduce the concept of a stream and that of a stream processing function. Let M be a set of messages. By the set M^{∞} we denote the set of infinite sequences over the set of messages M. The set M^{∞} can be understood to be represented by the total mappings from the natural numbers $\mathbb{N}$ into the message set M. More precisely, we write S^{∞} for the function space $\mathbb{N}^{+} \to S$ and $\mathbb{N}^{+}$ for $\mathbb{N} \setminus \{0\}$. Formally, we define the set of timed streams (see also [Broy 83]) by

$$M^{\aleph} =_{def} (M^{*})^{\infty} .$$

For every set of channels C, every mapping $x: C \to M^{\aleph}$ provides a complete communication history. Note that the set $(C \to M^{*})^{\infty}$ and the set $C \to (M^{*})^{\infty}$ are isomorphic. Moreover, the set $M^{\aleph}$ is isomorphic to the set of streams over the set $M \cup \{\surd\}$ with an infinite number of time ticks (here $\surd$ denotes a time tick; we assume $\surd \notin M$).

We denote the set

$$C \to M^{\aleph}$$

of channel valuations by streams by

$$\vec{C}$$

For the set

$$C \to (M^{*})^{*}$$

we write

$$C^{*\vec{}}$$

We denote for every natural number $i \in \mathbb{N}$ and every stream $x \in M^{\aleph}$ by

$$x{\downarrow}i$$

the sequence of the first i sequences in the stream x. It represents the sequences of messages communicated in the first i time intervals. By

$$\overline{x} \in M^{*} \cup M^{\infty}$$

we denote the finite or infinite stream that is the result of concatenating all the finite sequences in the stream x. $\overline{x}$ is a finite sequence if and only if a finite number of sequences in x are nonempty. Going from the stream x to $\overline{x}$ corresponds to a time abstraction. In the stream x we can find out in which time interval a certain message arrives, while in $\overline{x}$ we see only the messages in their order of communication but not their timing.

For streams $z_1, z_2 \in M^{*} \cup M^{\infty}$ we denote by $z_1 \hat{} z_2$ the concatenation of the streams. If z_1 is infinite then $z_1 \hat{} z_2 = z_1$.

For streams $z_1, z_2 \in M^{*} \cup M^{\infty}$ we define a prefix ordering as follows:

$$z_1 \sqsubseteq z_2 \Leftrightarrow \exists\, z_3 \in M^{*} \cup M^{\infty}: z_1 \hat{} z_3 = z_2$$

We use both notations $x{\downarrow}$ and $\overline{x}$ introduced for streams x also for tuples, channel valuations, and sets of timed streams by applying them pointwise.

Fig. 1 describes the syntactic interface of a component with the input channels $x_1, \ldots, x_n$ of the sorts $M_1, \ldots, M_n$ and the output channels $y_1, \ldots, y_m$ of sorts $M'_1, \ldots, M'_m$. In the theoretical treatment we assume for simplicity always $M_j = M$ and $M'_j = M$.

We represent the behaviour of a possibly nondeterministic component with the set of input channels I and the set of output channels O by a function:

$$F: \vec{I} \rightarrow \wp(\vec{O})$$

This function yields the set of output histories F.x for each input history x.

However, not every function of that functionality can be seen as a proper representation of the behaviour of a component. Only if a set-valued function on streams fulfils certain properties we accept it as the representation of a behaviour. To give a precise definition of these properties we introduce a number of notions more formally. A function

$$F: \vec{I} \rightarrow \wp(\vec{O})$$

is called (note that a function $f: \vec{I} \rightarrow \vec{O}$ can be seen as a special case of a set-valued function and therefore all definitions carry over)

- *timed*, if for all $i \in \mathbb{N}$ we have

  $x{\downarrow}i = z{\downarrow}i \Rightarrow F(x){\downarrow}i = F(z){\downarrow}i$

- *time guarded*, if for all $i \in \mathbb{N}$ we have

  $x{\downarrow}i = z{\downarrow}i \Rightarrow F(x){\downarrow}i+1 = F(z){\downarrow}i+1$

- *realisable*, if there exists at least one time guarded function $f: \vec{I} \rightarrow \vec{O}$, such that for all input histories x:

  $f.x \in F.x$

  By $[\![F]\!]$ we denote the set of time guarded functions f with $f.x \in F.x$ for all input histories x. Realisability means that there is a (deterministic) strategy (not necessarily computable, however) that in a component implementation makes sure that for every input history x interaction message by message leads to an output history that is contained in F.x. Note that if a deterministic strategy does not exist, then a nondeterministic strategy does not exist, either.

- *fully realisable*, if it is realisable and for all input histories x:

  $F.x = \{f.x: f \in [\![F]\!]\}$

We assume in the following that stream processing functions that represent the behaviour of components are always time guarded and fully realisable.

5. Translation of State Transition Machines to Stream Processing Functions

In this section we define an abstract semantics for STMs using stream processing functions. We consider only the time independent case to start with. We associate a stream processing function with a STM that is given by the transition function Δ using the following definition. More precisely, we associate a time guarded function F_σ with every state $\sigma \in$ State as defined by the following equation:

$$F_\sigma(x) = \{y \in \vec{O}:$$
$$(\exists\, i \in \vec{I}^*,\ o \in \vec{O}^*,\ \sigma' \in \text{State},\ x' \in \vec{I},\ y' \in \vec{O}:$$

$$\overline{y} = o\hat{}\overline{y'} \wedge \overline{x} = i\hat{}\overline{x'} \wedge \Delta(\sigma, i).(\sigma', o) \wedge y' \in F_{\sigma'}(x')) \vee$$
$$(\forall i \in \vec{I^*}: i \sqsubseteq \overline{x} \Rightarrow \forall o \in \vec{O^*}, \sigma' \in \text{State}: \neg\Delta(\sigma, i).(\sigma', o))\}$$

The first (existentially quantified) part of the right-hand side of this defining equation treats the case where at least one of the transition patterns applies. The second part treats the case where for the input stream x none of the transition patterns applies. This case is mapped onto arbitrary output and is called *chaos*. This technique guarantees that the function F_σ is always realisable.

If we do not want to associate a chaotic, but a more specific behaviour to input situations where no input pattern applies, we can work with default transitions (for instance time ticks) or simply drop the second clause in the definition. Working with chaos, however, has the advantage that adding input patterns in a transition diagram for input for which no pattern applied so far is a correct refinement step in the development process (see [Rumpe 96]).

A situation for which a less liberal treatment than chaos is suggestive is the case where an input pattern is still possible but never realised. This is the case if for some input pattern i for which $\Delta(\sigma, i).(\sigma', o)$ holds for some state σ' and some output pattern o and for the input history x we have

$$\overline{x} \sqsubseteq i$$

but $(\neg i \sqsubseteq \overline{x})$. This means that, speaking in operational terms, we can never be sure that we get further input that either enables the rule or definitively shows that the rule cannot be applied. However, in such a case only time ticks may be generated as output, since to generate other output may turn out to be incorrect (with respect to the transition relation) by input arriving later. If only time ticks are generated, in the end the STM generates an infinite stream of empty sequences[1)].

If the STD contains cycles then the definition of F_σ is recursive. In this case, we cannot be sure that by the defining equation above the behaviour F_σ is uniquely specified. Therefore we define F_σ to be the largest (in the sense of elementwise set inclusion) time guarded function that fulfils this equation.

Our model of the behaviour of a component works with timed input and output streams. Since the input and output patterns as introduced above do not refer to timing, in the definition we work with the time abstractions of the input and output streams. Of course, we may also work with input patterns that refer to time. We will come back to this issue in section 9. The explicit time concept in the behaviour model of a component allows us also to deal with priorities in transitions and with a reaction due to lack of input.

To avoid the problems with unguarded recursion that may lead to defining equations the fixpoints of which are not unique, we may assume for convenience that for every input pattern i with the property

$$\Delta(\sigma, i).(\sigma', o)$$

every sequence in the output pattern o for each of the channels contains at least one time tick or one message. Formally this requirement is expressed by replacing the equation

$$\overline{y} = o\hat{}\overline{y'}$$

in the definition above for the function F_σ by the formula

[1)] In fact, this is not a deep problem since these pathological behaviors are ruled out by the realizability closure RC which is a mapping that maps a set valued function F onto the largest fully realizable component contained in F. It is defined by $RC[F].x = \{f.x: f \in [\![F]\!]\}$.

$$\exists\ o' \in O^{*\vec{*}}: y = o' \hat{}\ y' \wedge o = \overline{o'} \wedge \forall\ c \in O: o'.c \neq \langle\rangle$$

Then the equation is guarded. Every message that occurs in an output stream is then caused explicitly by a transition of the STM or by chaos. In terms of timing, we may even refine the assumption above and assume that every transition requires at least one time tick on all its input and output channels.

We even permit transitions with empty input patterns. In this case we speak of *spontaneous transitions*. If we work with our assumption that for a transition on every input channel there is at least one time tick, then a spontaneous transition is only enabled if there are no messages but only time ticks on the input channels.

The assumption that transition rules are always guarded, at least by time ticks, leads to a very robust model of time with a proper notion of causality. However, when specifying components where timing properties are not significant, it may be to get rid of the timing requirements for transitions. A component behavior

$$F: \vec{I} \rightarrow \wp(\vec{O})$$

is called *timing independent*, if for all $x, z \in \vec{I}$ we have

$$\overline{x} = \overline{z} \Rightarrow \overline{F.x} = \overline{F.z}$$

The we may use the operator

$$\text{retime} : (\vec{I} \rightarrow \wp(\vec{O})) \rightarrow (\vec{I} \rightarrow \wp(\vec{O}))$$

to get rid of ubconvenient time resrtiction. We assume

$$\overline{\text{retime}(F).x} = \overline{F.x}$$

The function retime rearranges all the time ticks in the input and output streams. We may choose in the implication ordering the largest time guarded function for retime(F) that fulfils the equation above. Then all timings that are possible are allowed. Of course, we require that retime(F) is again time guarded to guarantee a proper time flow.

Note: Priorities and Spontaneous Transitions
Since our basic model of system behaviour is timed we may introduce priorities for the transition rules, too. Let us assume that the transition rule R_1 has a higher priority than the transition rule R_2. Such priorities may be indicated for instance by special graphical means in STDs.

In a state transition system that is correct with respect to this priorities we require: whenever for an input stream x for a time point j the input pattern of rule R_1 applies to a prefix

$$x{\downarrow}j$$

then the transition rule R_2 may only be chosen if there exists a time point j' such that for the initial input

$$x{\downarrow}j$$

the pattern of the transition rule R_2 with the lower priority applies but not the pattern of the transition rule R_1 with the higher priority.

Spontaneous transitions are transitions with an empty input pattern. In other words, the transition can be selected without getting any specific input messages. Spontaneous transitions are not a problem in our setting. However, if we have cycles in STDs all transitions of which are spontaneous, then a behaviour is included that never takes any input into account again.

This perhaps unintended behaviour can be avoided by requiring that a spontaneous transition is only allowed if a time tick arrives on each of the channels. Then a transition is only allowed if there is no input within a time interval. This allows us to support transitions with "negative" input patterns. Such a transition is only possible if there is no input on the set of indicated input channels within some time bound. The idea of "negative" input conditions can also be introduced to our version of STDs. A negative input condition expresses that at certain channels there is not input. In our case this is easily translated in a positive condition: on the respective channel we receive a time tick. We will come back to this issue in section 9. □

Note that we can give along these lines a precise treatment for sophisticated concepts like priorities and spontaneous transitions due to our carefully chosen semantic model that includes time. Without an explicit notion (at least on the semantic level) of time a proper semantical treatment of priorities or of spontaneous reactions is difficult (or even impossible).

Note the impossibility to talk about priorities without an explicit notion of time. If two transition rules R1 and R2 are enabled at the same time and R1 has a higher priority than R2, then of course always R1 should be executed. However, if only R2 is enabled then we have to admit that R2 may be executed till the time point where R1 gets enabled. If we do not have any information about the timepoints at which R1 and R2 get enabled then any temporal relation may be given and we may freely (nondeterministically) choose between R1 and R2. This nondeterminism is, of course, not a proper model of priorities. Only if we have an explicit notion of time in our model we can express formally what the idea of a priority means.

6. Syntax of State Transition Systems

A pragmatic and practically applicable description technique of system behaviours has to provide concrete representations in the form of a specific syntax for all parts of a STD description. In this section we introduce such a concrete syntax (see also [Grosu et al. 96b]).

We describe the interface of a component by the following sets of channels:

I set of input channels (with assigned sorts of messages for each channel),
O set of output channels (with assigned sorts of messages for each channel).

We describe this interface graphically by a data flow node as shown schematically in Fig 1. Let for this purpose the following sets be given:

M set of messages (could be one specific for each channel, if channels are assigned sorts),
State set of states (the data state space).

The set State is described syntactically by a number of declarations of *variables* (also called the *attributes* of the state) together with their sorts:

$$v_1 : D_1, ..., v_n : D_n.$$

Here D_1, ..., D_n are the sorts of the variables. They can be described by sort declarations or simply assumed to be given. The description of the state space can be given by graphical means, too, such as for instance by an extended E/R-diagram.

The state predicates for the nodes of the diagram can easily be written by Boolean expressions (formulas of predicate logic).

A transition rule is described by a label in the STD consisting of an input pattern and an output pattern (separated by "/") of the form:

$$\{ A \}\ x_1{:}E_1, \ldots / y_1{:}B_1, \ldots \{ C \}$$

Here A and C are Boolean expressions, x_1, ... and y_1, ... are input and output channels. E_1, … and B_1, ... are expressions denoting messages or sequences of messages of the sort of the channel. The expressions may contain the attributes v_1, ..., v_n and the primed attributes v_1', ..., v_n' as well as free identifiers. We use the common convention that by v' we denote the value of the state attribute v after the transition, while v denotes its value before the transition.

It may improve the readability of STDs if we abbreviate every formula and every transition rule in a diagram by an identifier and collect their meaning in a table. As a notational convention, we may leave out the names of channels if the described components have only one input channel (or one output channel) or if by the sort information the channel can be determined uniquely.

7. Relating the Syntax to the Semantics

How to associate with a syntactic input/output pattern the predicates introduced in section 3 may seem quite obvious. So we do not give this quite straightforward formalisation in full detail. What may deserve some attention, however, is the treatment of the free variables in the input and output patterns.

Formally the data state space Data_State of a STD which includes the attributes v_1, ..., v_n of the sorts D_1, ..., D_n is represented by the set of valuations of the attributes:

$$\text{State} = \{\sigma: \{v_1, ..., v_n\} \rightarrow D: \forall\ i, 1 \leq i \leq n: \sigma(v_i) \in D_i\}$$

where the D_i are the sets of data elements associated with the sorts D_i. D denotes the universe of all data values:

$$D = \bigcup_{i=1}^{n} D_i$$

A state is, by this definition, a valuation of the state attributes v_1, ..., v_n that is consistent with the sort restrictions. Let V denote the set $\{v_1, ..., v_n\}$ of identifiers for attributes and V' denote the set $\{v_1', ..., v_n'\}$ of primed attribute identifiers. Let X be a set of further identifiers.

For simplicity we consider only transition patterns of the form

$$\{A\}\ x{:}E / y{:}B\ \{C\}$$

for defining the semantics. A generalisation of our semantic definitions to general patterns is rather straightforward and should be obvious. For our transition pattern we assume that

- E is an expression of the sort of the channel x,
- A is a Boolean expression,
- B is an expression of the sort of the channel y,
- C is a Boolean expression.

Each of these expressions contains free identifiers from the set $V \cup V' \cup X$. In a more restricted approach we would not admit occurrences of the primed identifiers from V' in the

expressions A, E, and B. However, often it is notationally convenient to use v' in these expressions, too. This avoids the use of additional free identifiers.

We define the pattern predicates as follows (let Val_α be the valuation function that associates a value with each expression or formula, by α we denote valuations for all identifiers in V, V' and X)[1] :

$$Q(\sigma, i) \equiv \exists\, \alpha: \quad Val_\alpha[A] \wedge$$
$$\forall\, c \in I: i.c = \textbf{if } c = x \textbf{ then } \langle Val_\alpha[E] \rangle \textbf{ else } \langle\rangle \textbf{ fi} \wedge$$
$$\forall\, a \in V: \sigma(a) = \alpha(a)$$

Note that the last line makes sure that α and σ coincide for all attributes of the state.

$$R(\sigma, i).(\sigma', o) \equiv \exists\, \alpha: \quad Val_\alpha[C] \wedge Val_\alpha[A] \wedge$$
$$\forall\, c \in I: i.c = \textbf{if } c = x \textbf{ then } \langle Val_\alpha[E] \rangle \textbf{ else } \langle\rangle \textbf{ fi} \wedge$$
$$\forall\, c \in O: o.c = \textbf{if } c = y \textbf{ then } \langle Val_\alpha[B] \rangle \textbf{ else } \langle\rangle \textbf{ fi} \wedge$$
$$\forall\, a \in V: \sigma(a) = \alpha(a) \wedge$$
$$\forall\, a \in V': \sigma'(a) = \alpha(a)$$

This fixes the semantics of our syntax. Note that we have made sure that

$$R(\sigma, i).(\sigma', o) \Rightarrow Q(\sigma, i)$$

For notational convenience we admit to replace the postcondition also by statements in the form of an assignment. So we write

$$v := v + 1$$

to express v' = v + 1 and that all other attributes remain unchanged.

8. Examples

In this section we give a number of small and medium size examples of specifications by STDs. We start with a very simple one.

Example: The Sorting Cell
The sorting cell can be used to build a network for sorting a sequence of natural numbers. Each cell stores at most one natural number. First we describe the data model which, in this simple case, consists only of a sort declaration:

sort M = Nat | {®}

The element ® is used as a dummy in the place of a number. The sort Nat denotes the set of all natural numbers. The syntactic interface of the sorting cell and its state space are represented by a data flow node as it is shown in Fig. 2.

[1] Note that we repeat the condition $Val_\alpha[A]$ in the definition of R to make sure that the binding of the variables in X is consistent in E, B, C, and A.

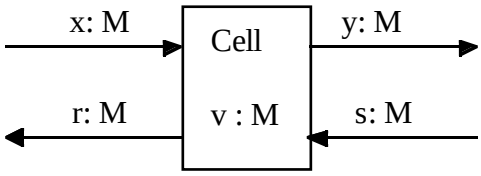


Fig. 2 Data Flow Node Giving the Syntactic Interface and the Local Data State Space of the Sorting Cell

The data state space is given by the declaration of the attribute

v: M

We describe the behaviour of the data flow node by a STD. The initial state is labelled by ⟶.

The sets of data states associated with the nodes representing the control states are specified by predicate logic as follows:

Empty: $v = ®$
Waiting: $v = ®$
Full: $v \neq ®$

We see that in this case, as often, the state sets associated with the individual nodes by the data state predicates are not disjoint. □

The fact that the data state sets of the nodes are not disjoint is not a problem. The actual state of a component is represented by a pair, consisting of an element $\sigma \in$ State from the data state space and a node k of the STD representing the control state such that $S_k(\sigma)$ holds. The node k can be seen as the control state.

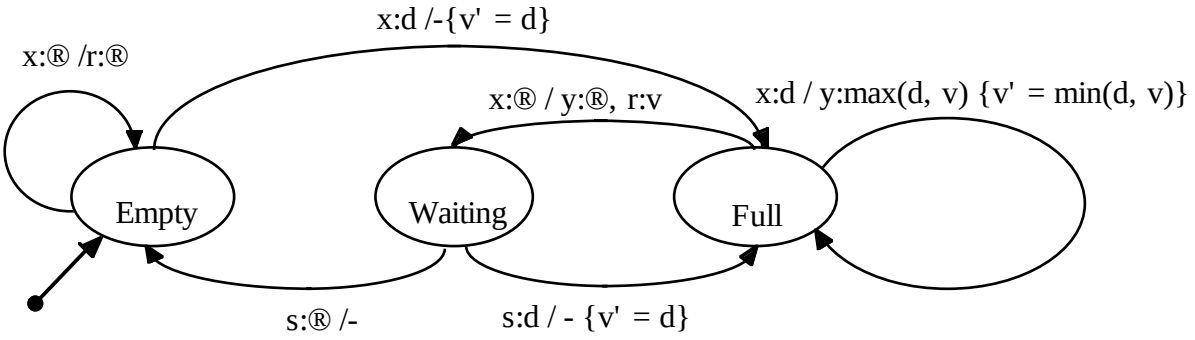


Fig. 3 STD Defining the Behaviour of the Sort Cell (let d be of sort Nat)

Example: A simple store
We follow again the scheme we have used in the previous example. We describe the syntactic interface of the component called store by the data flow node given in Fig. 4.

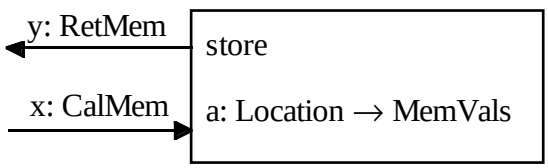


Fig. 4 Data Flow Node Giving the Syntactic Interface of the Store

Let Location be the sort of memory locations and MemVals the sort of values that can be stored in the memory. The data model of the component store consists of the sorts of messages that are declared as follows[2)]:

sort CalMem = put (i: Location, d: MemVals) | get (i: Location)

RetMem = ret (c: CalMem, rv: RetVals)

RetVals = MemVals | {MemFail, Ack}

The set State is declared by the attribute

a: Location $\rightarrow$ MemVals

Initially all memory cells have the initial value initial_val which is a distinguished element of the set MemVals

Initial: a(i) = initial_val

In the STD for the component store we work with only one node. In such a case, we better do not use a STD at all, but rather work with a set of transitions very similar to TLA (see [Abadi, Lamport 90]).

x:put(i, d) / y:ret(put(i, d), Ack) {a'(i) = d $\wedge$ $\forall$ j, j $\neq$ i: a'(j) = a(j)}
x:put(i, d) / y:ret(put(i, d), MemFail) {a' = a}
x:get(i) / y:ret(get(i), a(i)) {a' = a}
x:get(i) / y:ret(get(i), MemFail) {a' = a}

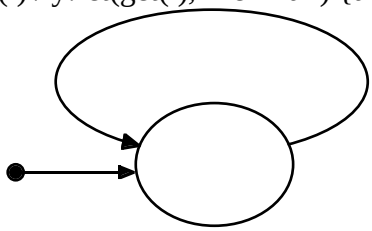


Fig. 5 STD with only One Node

The transitions can be gathered into a table. This way we can avoid to write long formulas in the STDs.

Tab. 1 Transition Table

name	input	output	postcondition
Write	x:put(i, d)	y:ret(put(i, d), Ack)	a'(i) = d $\wedge$ $\forall$ j, j $\neq$ i : a'(j) = a(j)
WriteFail	x:put(i, d)	y: ret(put(i, d), MemFail)	a' = a
Read	x:get(i)	y:ret(get(i), a(i))	a' = a
ReadFail	x:get(i)	y:ret(get(i), MemFail)	a' = a

[2)] Throughout this paper we use a notation for the declaration of sorts with an obvious meaning.

We can use the table together with the STD given in Fig. 6, which only refers to labels included in the table.

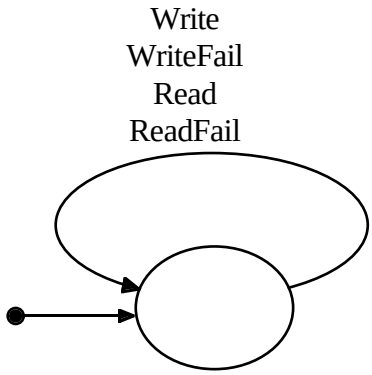


Fig. 6 STD with Abbreviations for Transitions

The STD in Fig. 6 contains only one node. In such a case, as pointed out, it is an overkill to work with a STD at all. However, we may replace the STD by one that contains two nodes such that each transition is split into two, one that accepts the input and the next one which produces the output. Then we need a more detailed state space to store the location (and the value) for which reading or writing is required. So the state space is given by the attribute declaration

a: Location $\rightarrow$ MemVals, c: CalMem

The table Tab. 2 lists all the transitions occurring in the STD given in Fig. 7.

Tab. 2 Transition Table for the Decoupled STD

name	condition	input	output	postcondition
Com		x:e	-	c' = e $\wedge$ a' = a
Write	c = put(i, d)	-	y:ret(put(i, d), Ack)	a'(i) = d $\wedge$ $\forall$ j, j $\neq$ i : a'(j) = a(j)
WriteFail	c = put(i, d)	-	y: ret(put(i, d), MemFail)	a' = a
Read	c = get(i)	-	y:ret(get(i), a(i))	a' = a
ReadFail	c = get(i)	-	y:ret(get(i), MemFail)	a' = a

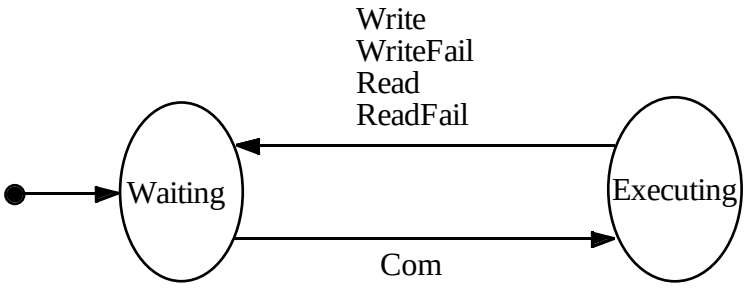


Fig. 7 STD for the Store with Transition Rules Separated into Input and Output

This refined version of the store has (abstracting from timing) the same behaviour as the store described by the diagram given in Fig. 6. □

The example above shows a technique for detailing and refining a STD by splitting its nodes and its transitions (see also [Rumpe 96] for refinement steps on STDs). If all nodes have only transition rules where either the input pattern or the output pattern is empty, the STD is called *decoupled*. Such STDs correspond closely to the I/O-automata of [Lynch, Stark 89] where in each transition exactly one input or one output message is processed.

So far, we have shown examples of STDs only for the description of components and a first example of a refinement. Now we show how to put together components into a network such that they cooperate. Since STDs describe set-valued functions on streams, for which a composition within data flow nets is well-defined (see [Broy, Stølen 94]), we can use this concept of composition immediately for state transition descriptions, too. We demonstrate this with the help of a simple component called driver that co-operates with the memory of the previous example.

Example: Store Driver
A driver is a component that controls the access to the memory encapsulated in the component store. It receives calls and forwards them to the memory. It may retry a call for the memory if a memory call fails. However, it may stop trying at any time, even before it tried at all. We use our general specification scheme again to describe the driver.

Tab. 3 Table of Sorts

MemLocs		memory locations,
MemVals		memory values,
PrIds		identifiers for processes,
CalMem	= put (i: Location, d: MemVals) | get (i: Location)	memory calls,
RetVals	= MemVals | {MemFail, Ack}	return values,
RetMem	= ret(c: CalMem, rv: RetVals)	return messages of the memory,
Returns	= ren(c: Calls, m: RetVals)	return messages of the driver,
Calls	= ca(pi: PrIds, mc: CalMem)	calls for the driver.

Tab. 3 shows all the sorts involved. It defines the data model. The syntactic interface of the component driver is described by a data flow node.

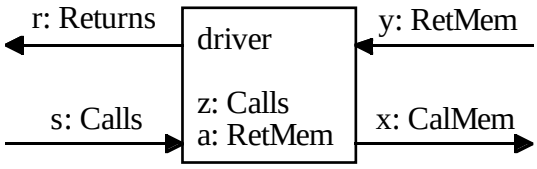


Fig. 8 Data Flow Node Giving the Syntactic Interface of the Driver

The local data state space of the component driver is described by the following attributes:

z: Calls, a: Returns

The STD given in Fig. 9 fixes the behaviour of the component driver.

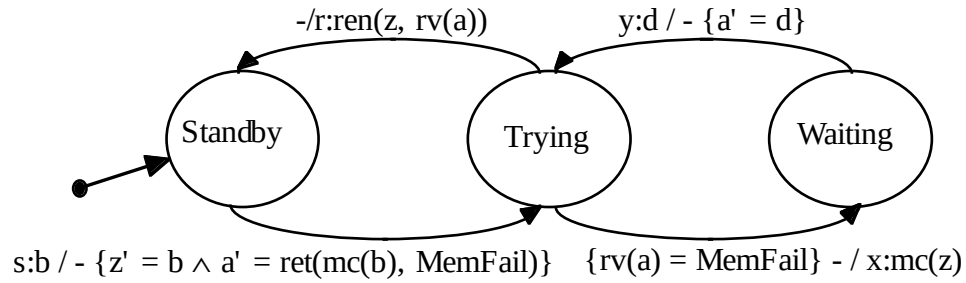


Fig. 9 STD for the Driver and its State Attributes

The driver is designed to cooperate with the store. The driver may try forever in the case of memory failures, if we do not make any fairness assumptions. To get rid of this "unfair" behaviour, we may add the liveness condition

$$\# \bar{r} = \# \bar{s}$$

that expresses that every input leads to an output under the assumption

$$\# \bar{x} = \# \bar{y}$$

that expresses that all calls sent on channel x to the store get eventually served. It is very helpful that the state transition description can be combined with logical equations to express liveness properties. We may introduce more refined fairness properties for the driver that way. For instance, the driver may guarantee to return never MemFail provided a message different to MemFail is guaranteed by the store if the driver retries long enough. □

The driver can be composed with the component store specified in the previous example. This composition is asynchronous as it is defined in FOCUS (see [FOCUS 92]). By combining the driver and the store, we obtain a component which again can be described by a STM. Its state space consists of the product of all the state spaces of the subcomponents. In addition, we need buffers in the state space to store the messages sent on internal channels but not yet received.

Such buffers can be avoided, however, if we work with special decoupled STDs defining STMs that do all their transitions with input from internal channels in two steps. In the first step the input is received and in a following step without input the output is produced. We require here that in every control state for every nonempty input an input pattern can be selected without producing output. By such a refinement the buffers have to become part of the state space.

Example: Composing the Driver and the Store
The interface of the driver and the store fit together such that we may compose them as shown in Fig. 10.

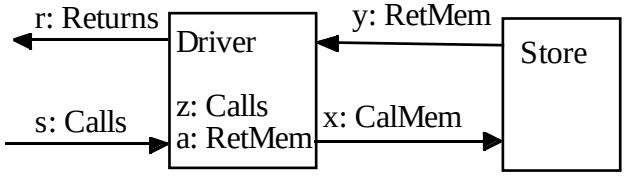


Fig. 10 Composition of the Driver and the Store

This composition can also be described by a textual syntax (see [Andl 95]). As the diagram suggests, the channels x and y are hidden and therefore called internal channels.

The driver and the second version of the memory component are already in a form that allows us to work without additional buffers for the channels when combining their state spaces into a state space for the state transition description of the composed system. □

Of course, the introduction of the buffers can only be avoided completely, in general, if the decoupling is broken down onto the level of individual steps and in every node every input can be accepted. Only then every message can be processed without immediately generating further messages and thus no additional buffers are needed to store messages on input channels, provided the system does the corresponding scheduling and runs fast enough. For a state view on composed systems see [Grosu et al. 96a].

9. Describing Time Dependent Components by STDs

So far we have only studied STDs for the specification of the behaviour of time independent components. Now we extend our approach to time dependent components. For them we have to use input patterns that refer explicitly to time.

We define a timed STM that works with timed input and output patterns by a state transition function

$$\Delta: \text{State} \times I^{**\vec{}} \to (\text{State} \times O^{**\vec{}} \to \text{Bool})$$

A timed input segment $i \in I^{**\vec{}}$ is concatenated to a timed input stream $x \in \vec{I}$ by the operator $\check{}$. This operation is defined as follows (let $a, b \in I^{*\vec{}}$):

$$(i\hat{}\langle a\rangle)\check{}(\langle b\rangle\hat{}x) = i\hat{}(\langle a\hat{}b\rangle\hat{}x)$$

The operator $\check{}$ concatenates two sequences x, y of sequences by concatenating the last sequence in x with the first one in y.

An input segment $i \in I^{**\vec{}}$ fits to a stream $x \in \vec{I}$ if it is a "generalised" prefix, abbreviated by $i \sqsubseteq_\tau x$. This relation is defined by:

$$i \sqsubseteq_\tau x \Leftrightarrow \exists\, z \in \vec{I} : i\check{}z = x$$

We associate a stream processing function F_σ with a STM with the transition function Δ by the following definition that associates a function F_σ with every state $\sigma \in$ State by the equation

$$F_\sigma(x) = \{y \in \vec{O}: \quad (\exists\, i \in I^{**\vec{}}, o \in O^{**\vec{}}, \sigma' \in \text{State}, x' \in \vec{I}, y' \in \vec{O}:$$
$$y = o\check{}y' \wedge x = i\check{}x' \wedge \Delta(\sigma, i).(\sigma', o) \wedge y' \in F_{\sigma'}(x')) \vee$$
$$(\forall\, i \in I^{**\vec{}}: i \sqsubseteq_\tau x \Rightarrow \forall\, o \in O^{**\vec{}}, \sigma' \in \text{State}: \neg\Delta(\sigma, i).(\sigma', o))\}$$

Again we choose for the set-valued function F_σ the largest (with respect to set inclusion) function for which the equation above holds. This definition of the semantics of timed state transition systems by stream processing functions is quite straightforward. More sophisticated is the syntax of STDs for timed components.

We start with an explanation of the time model. Each transition is executed at a time point. Thus each node in a transition diagram is reached at a point in time. Then it may take a while until an input pattern may become enabled. The time distance (the number of time ticks)

between the reaching of a node by a transition and the pattern i becoming enabled for the next transition is denoted by δ_i. It is formally defined by

$$\delta_i = \min \{j \in \mathbb{N}: i \sqsubseteq_\tau x{\downarrow}j \}$$

Informally δ_i defines the number of time ticks that take place after the current control state has been reached by a transition until the input pattern i becomes enabled. We may refer to δ in the specifying formulas. According to our definition we can be sure that in the timed case a transition is carried out as soon as possible. As soon as enough input is available to enable an input pattern the transition is carried out and the target state is assumed immediately[3)].

We want to work with relative time and with absolute time in applications. Absolute time always refers to the time distance between the initialisation of the system and the time at which a certain event happens. Absolute time allows us also to speak about dates (days, hours, years) provided, we specify the absolute time of the initialisation of the system (system start-up time) and also indicate which time unit (frequency) the system works with for its time intervals such as for instance microseconds, milliseconds, seconds, minutes or days.

Basically we are interested to express the following timing properties when specifying component behaviour:

- delay by some time span with lower and upper bounds,
- time-out, watchdog (guaranteed reaction within time bounds, captured by upper bounds on the delay),
- pre-emption, interrupt (immediate reaction on timed events),
- clocks, timers.

We need a special notation to express time events in STDs.

A simple syntax for timed input patterns is obtained if we allow time ticks $\surd$ as pseudo signals in the input patterns and refer to the time when a node was reached. Therefore all we need is a timing convention. By τ we denote the absolute time at which a node has been reached.

Example: Timer
The timer is a component that works with the following input and output sorts:

sort Tin = set (t: Nat)

sort Tout = {time-out}

Its interface is described by Fig. 11.

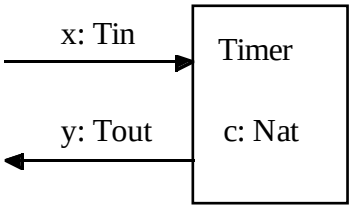


Fig. 11 Timer Interface

The state space of the timer consists only of the time attribute:

c: Nat

[3)] This implies that an input pattern is always avoided if another one gets always enabled sooner.

that stores the number of time ticks until the time-out occurs. If c = 0 the timer is not set. The behaviour of the timer is described by the STD given by Fig. 12. There, by √ we denote the time tick signal which according to our semantic model indicates the end of a time interval.

The control state predicates are defined by the following formulas

Not set: c = 0
Set: c > 0

For this example we do not have to refer to absolute time but only to the relative time captured by the time concept that is part of our system model. □

We typically refer to absolute time in business applications. There we express that an event can take place at absolute time γ by a transition with the precondition

$\{\tau + \delta = \gamma \ldots\} \ldots / \ldots$

Here $\tau + \delta$ denotes the absolute time.

Explicit time allows us to be very explicit about sophisticated notions like interrupts, and time-outs. A special case are soft time-outs where sharp time bounds are not required but a reaction is requested eventually if there is a lack of input. A typical example is the repetition of sending a message for the sender in the alternating bit protocol if an acknowledgement signal is missing (see appendix).

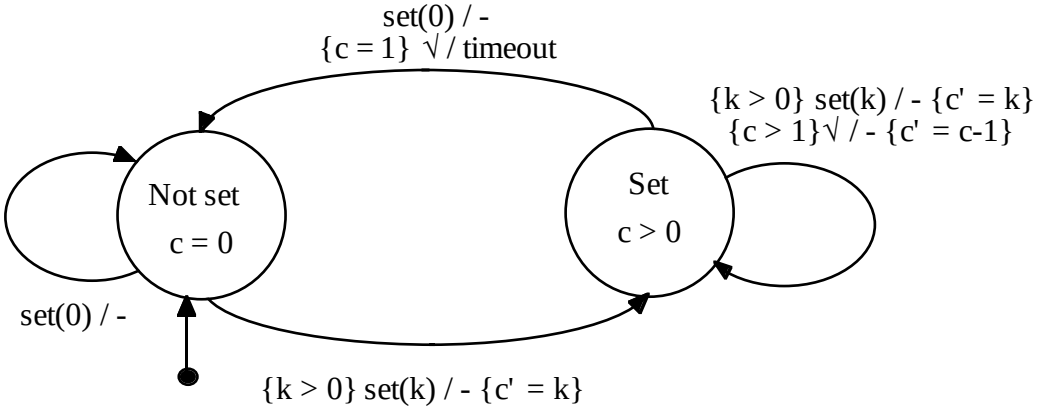


Fig. 12 STD of the Timer

Negative input conditions:

In many applications we are interested to react eventually due to missing input. More precisely, the component is required to react if on a specific channel a message is expected but does not arrive within a certain interval of time (reaction on time out). Of course, this informal description is not precise enough. A more precise description has to indicate how long the component waits until it responds.

If the waiting time is bounded then we simply may work with timed STDs. A special case is that no specific time bound is given. In other words, the component has to react eventually if there is no input but it should wait at least say one time unit after it may react a finite number of times spontaneously. To express such a situation we write for channel x

x: -

in the input pattern. This means that if there is no input on the channel x after some finite nonempty amount of time the transition rule fires. We call this a *soft time-out*. □

Interrupts:
The idea of an interrupt is that by some external event the current activity of a system is stopped and another activity is started. The critical issue here is at which points of an activity an interrupt is accepted. We call this the *interrupt granularity*. An important requirement is that an interrupt is executed at the earliest point in time at which it can be executed. In other words, an interrupt is described by a transition rule with a high priority. This is easily expressible in our formalisms as long as we have individual interrupt channels. Otherwise we have to work with messages/channels with priorities. Then certain input patterns are processed as soon as they get valid. Earlier unprocessed input that arrived before the interrupt happened is dropped or saved. □

Pre-emption:
Pre-emption is very similar to interrupt. The only difference is that the activity that is interrupted is not continued at the point it was interrupted, but, if at all, at a fresh point that is independent of the state in which the component was interrupted. This corresponds to some specified re-initialisation. □

With our notation, pre-emption can be easily expressed in STDs by transitions with a higher priority and a state concept that forgets about the control state in which the interrupt was received. To model interrupts we either have to save the control state explicitly as well as relevant parts of the state space or we have to introduce a tuned notation along the ideas histories as found of statecharts. We come back to this issue under the heading of hierarchical STDs.

Of course, depending on the application, special notations are useful to support a flexible notation of interrupts and reaction on lacking input on specific channels (for more technical details on timed STDs see [Müller, Scholz 96] and [Broy et al. 97].

10. Multithreaded STDs

So far we have only worked with single-threaded STDs. At every point in time exactly one node is enabled. It may be interesting to extend our concept to multithreaded STDs. In a multithreaded STD the control state is not only characterised by one node but by a set or even a multiset of nodes. The number of threads may of course change dynamically.

To give semantics to multithreaded diagrams we have to extend our definition of F_σ for a single state $\sigma \in$ State to F_Σ where $\Sigma \subseteq$ State is a set (or even a multiset) of states. We only treat the nontimed case. For simplicity we assume that for each node in the transition diagram at most one thread is active[1)] . Otherwise we have to work with multisets of threads. We define

$$F_\Sigma(x) = \{y \in \vec{O}:$$
$$(\exists\, i \in \vec{I}^*, o \in \vec{O}^*, \sigma \in \Sigma, x' \in \vec{I}, y' \in \vec{O}, \sigma' \in \text{State}:$$
$$\overline{y} = o\hat{}\overline{y'} \wedge \overline{x} = i\hat{}\overline{x'} \wedge \Delta(\sigma, i).(\sigma', o) \wedge y' \in F_{(\Sigma\setminus\{\sigma\})\cup\{\sigma'\}}(x')) \vee$$
$$(\forall\, i \in \vec{I}^*, \sigma \in \Sigma: i \sqsubseteq \overline{x} \Rightarrow \forall\, o \in \vec{O}^*, \sigma' \in \text{State}: \neg\Delta(\sigma, i).(\sigma', o))\}$$

[1)] This applies, in particular, for a number of state machines with single threads composed in parallel.

This definition shows that in a multithreaded automaton the transitions work interleaved. If transitions work on disjoint sets of input and output channels, however, they can be carried out in parallel. For certain applications additional fairness assumptions about the relative speed of the different threads might be useful.

Example: RPC
Our driver in the example in section 8 can handle only one request at a time. To be able to handle more than one request at a time we introduce a set PrIds of processors and define the state space as follows:

v : PrIds $\rightarrow$ state (z: Calls, a: Returns)

We work with a multithreaded diagram (or more precisely with a family of single threaded diagrams that are executed in parallel). For all process identifiers p $\in$ PrIds we give one transition diagram. For an expression R that denotes a record of sort

cons(sel_1: M_1, ..., sel_n: M_n)

we use the convention that for an expression E

with R: E

stands for the expression

E[sel_1(R)/sel_1, ..., sel_n(R)/sel_n]

Fig. 13 shows the STD of the multithreaded driver.

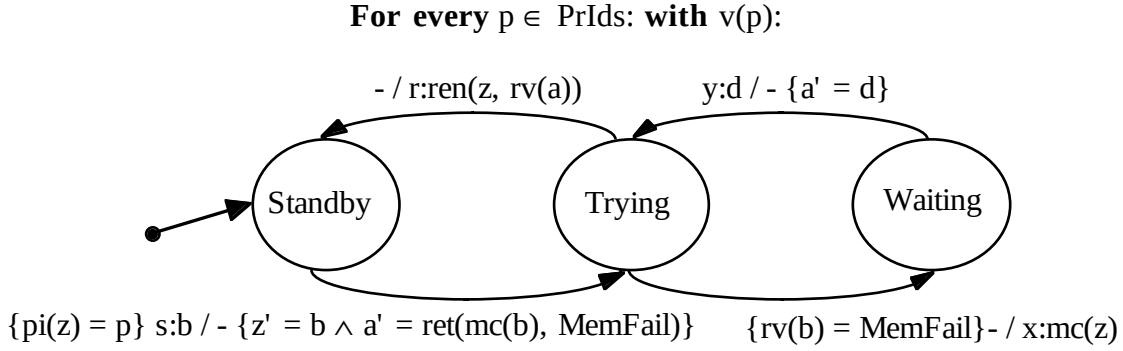


Fig. 13 STD for the Multithreaded Driver

Initially, all default outputs are memory failures:

Initial: a(v(p)) = MemFail

Note that the syntactic interface of the multithreaded Driver coincides with that of the single-threaded driver. □

In a multithreaded STD a set of nodes is active. Each transition moves the control of one of these nodes to another one. Chaotic default transitions are only allowed if for none of the active nodes a transition is enabled.

Note: If we use a set of nonconnected STDs each of which works with disjoint sets of channels, we can decompose the data flow node in a set of data flow nodes that work in

parallel. Vice versa, a data flow diagram where we have a transition diagram for each node is a special case of multithreaded state transition system. □

Multithreaded STDs correspond to the concept of parallel composition of state machines as it is found in statecharts (see [Harel 87]).

11. Hierarchically Structured STDs

STDs may be hierarchically structured. Then we give for certain nodes not just a state predicate, but a complete STD again. So the node has its own local control state space and its local state space predicates and its own local state attributes. The local STD associated with a hierarchical node is activated whenever we enter the node and deactivated, whenever we leave the node. We either may initialise the diagram again every time we enter it (in the case of pre-emption) or we may freeze the control and data state it is in when we leave it (in the case of interrupt) and reactivate it as soon as we enter it again. The idea of hierarchical STDs leads to particular notions of refinement of STMs.

For hierarchical STDs we may distinguish two cases:

(1) The channels in all the STDs in the nodes deal with a disjoint set of input and output channels compared with the transitions at the upper level.
(2) The channels to which the STDs in the nodes refer are the same as the channels in the upper level of transitions.

In the case (1) the levels are decoupled. In the case (2) the hierarchy seems less significant and can just be seen as a notational abbreviation. By hierarchical STDs we may nicely describe interrupts if we give priority to upper level transitions.

Local states may be persistent or non-persistent. For a persistent hierarchical node its state (the actual node and the values of the local attributes) are stored until the diagram is activated again.

Example: Video Remote Control
We specify a video remote control component VRC. Its syntactic interface is described by a data flow node as given in Fig. 14.

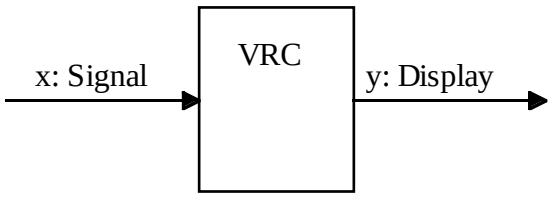


Fig. 14 Syntactic Interface of the VCR

We define a very simple version only. We use the following sorts that define the signals sent to the VCR and the observations that are caused by them:

sort Signal = {on, off} ∪ {ch[i]: i ∈ [1:16]} ∪ {c↑, c↓, v↑, v↓, voff, von}

sort Display = {ch[i]: i ∈ [1:16]} ∪ {vol[i]: i ∈ [0:7]}

By the signals c↑, ... we control the channels of the video and by the signals v↑, ... we control its volume. The state space is defined by the following attributes

a: {on, off}, c: [1:16], v: [0:7]

The behaviour of the VCR is defined by the STDs in Fig. 15 and Fig. 16. Double circles mark hierarchical nodes.

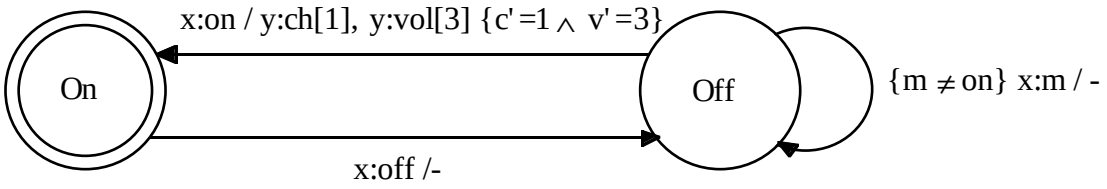


Fig 15 Top Level STD of the VCR

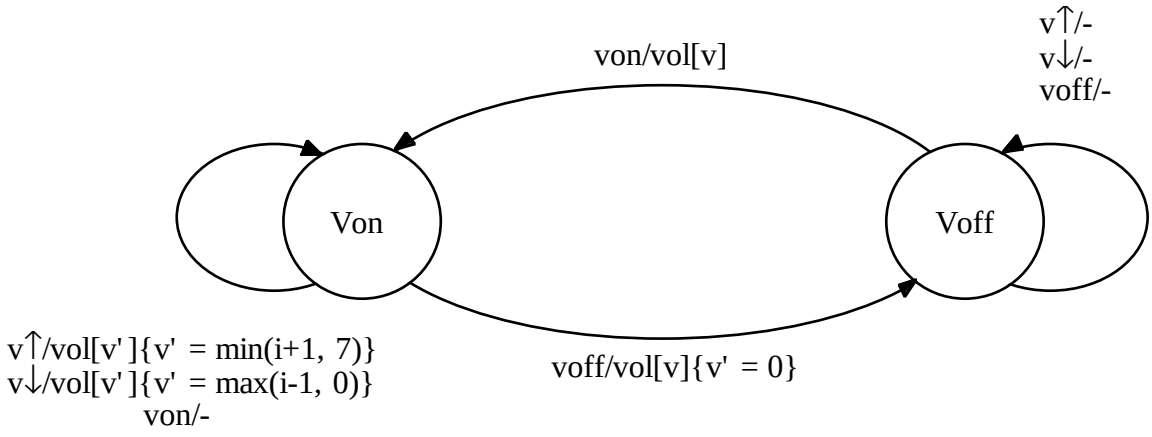


Fig. 16 STD for State Predicate Vol of the VRC

Here we use the convention that, if no postcondition is stated, in the successor state the values of the attributes do not change. More formally expressed, a = a' holds for all attributes a.

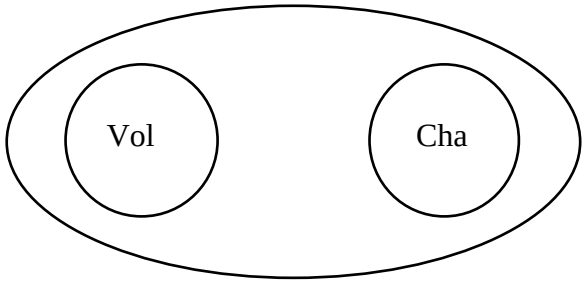


Fig. 17 Decomposition of the Node Von of the VCR

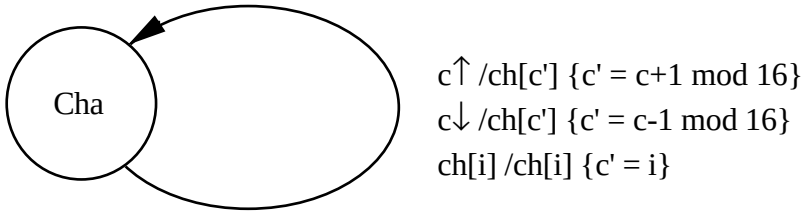


Fig. 18 STD for the Node Cha of the VRC

The predicates that define the state sets associated with the nodes are defined as follows:

Off: $a = off \wedge c = 1 \wedge v = 3$
On: $a = on$
Von: true
Voff: $v = 0$

The hierarchical decomposition is very helpful to keep the STDs small and manageable. □

Of course, we may combine timed input patterns also with hierarchical STDs. Hierarchical STDs can be explained as a rather straightforward notational extension of flat STDs. For every hierarchical node with k subnodes we introduce an attribute h that may assume one of the values {0, ..., n}. Every outgoing arc from the hierarchical node is replaced by n arcs from each of the nodes inside the hierarchical node. All these arcs are labelled by the same transition pattern as the transition at the higher level. We only add in the postcondition h := j for the arcs coming from node j if the hierarchical node is persistent (saves its history and thus models an interrupt). Otherwise we assign an initial node to h. Every ingoing arc we replace by n arcs one for each of the nodes j and add the precondition h = j to each of the transition rules. Of course, h is initialised by one of the internal nodes that are initial.

According to our concept that a node in a transition diagram denotes a set of states, a hierarchical node also denotes a set of states obtained by the union of the state sets of the internal nodes

12. Conclusion

Clearly our idea of a STD has much in common with the concepts of statecharts. However, in contrast to statecharts and the supporting tool STATEMATE, which were developed very much aiming at a practically helpful tool for a graphical system description technique, we are rather interested in a clean and simple semantic model and a tailored graphical description technique for it. We believe that the often confusing and painful discussions about the semantics of statecharts are an indication of not only a theoretical problem with the semantics of statecharts but also are a sign of a deep practical problem of statecharts. Obviously, the meaning of the chosen notation is less intuitive and self-explaining than anticipated. This may lead to serious methodological problems when using statecharts.

Also the explanation of the semantics as given in the recent paper [Harel, Naamed 96] contains a number of design decisions that are not clearly justified (for the formal semantics of a clean version of a subset of statecharts, see [Nazareth et al. 96]; for ideas how to clean up statecharts, see [Philipps, Scholz 97a], [Philipps, Scholz 97b], [Philipps, Scholz 97c], [Scholz 96a], [Scholz 96b] and [Scholz et al. 96]). Moreover, the many gimmicks of statecharts hide and obscure the basic semantic concepts. We believe that consequent encapsulation of states, a

strictly logical treatment of state spaces and transition rules, a simple time model, and proper notion of interfaces lead to a better manageable description medium.

There are many description formalisms based on STDs (such as extended finite STMs; for object oriented STDs see [Rumpe, Klein 96]). Prominent examples are, among others, SDL, statecharts, or ARGOS. All these formalisms suffer under the unpreciseness of their semantics. This is why we decided not to introduce a graphical formalism in the first place and then to define its semantics but rather to start from a semantic model and develop a graphical description formalism for it.

We consider our graphical specification method just as a notational sugar for the description of systems in a firm logical and mathematical framework. Everything written in the graphical style can also be presented by a table and translated directly into a set of logical formulas. The formulas, however, are not necessarily easier to read. In general, they are more lengthy and contain some syntactic noise. The particular choice of the names in a diagram, the structuring, and the layout of the diagram may in addition help to understand the behaviour of systems in more detail.

Since diagrams are, in our case, nothing than syntactic sugar, all the techniques from mathematical models and logical techniques for describing systems developed for FOCUS carry over immediately. So we immediately get a formal notion of property refinement, a formal notion of interaction refinement, and we can compose systems as demonstrated (see also [Broy, Stølen 94]). Therefore we consider what we have presented as an attempt to bring closer together the techniques used in practice today and the theoretical work towards the formal foundations that are more in the target of the research carried out today.

Acknowledgement

This work was heavily influenced by discussions in the KORSYS team, the SysLab team and the Focus team, in particular by Bernhard Rumpe, Barbara Paech, Radu Grosu, Cornel Klein, Bernhard Schätz, Alex Schmidt, and Katharina Spies. It is partly based on work done in a cooperation with Siemens ÖN. I thank Olaf Müller and Oscar Slotosch for helpful comments on the manuscript.

Appendix:

In this appendix we give a further simple example for the usage of STDs in system specification.

Example: Alternating Bit Protocol
As an example we consider the alternating bit protocol. It has the structure as given in Fig. 19.

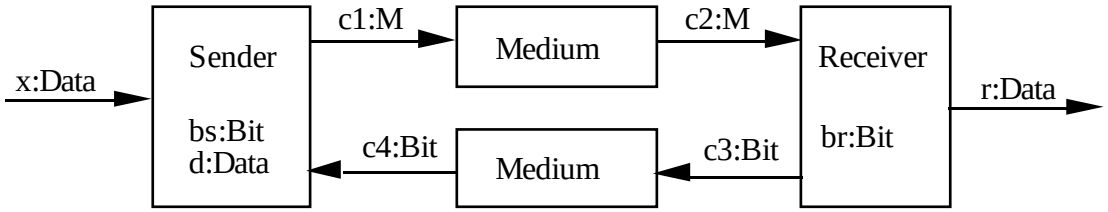


Fig. 19 Data Flow Net of the Alternating Bit Protocol

We use the following sorts:

sort Data

sort M = m(d:Data, b:Bit)

The STD of the component Receiver is given by Fig. 20.

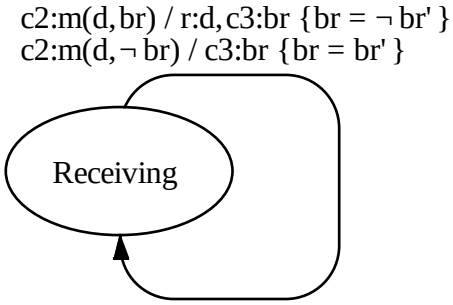


Fig. 20 STD of the Receiver

The state space of the receiver is indicated by the data flow node given in Fig. 21.

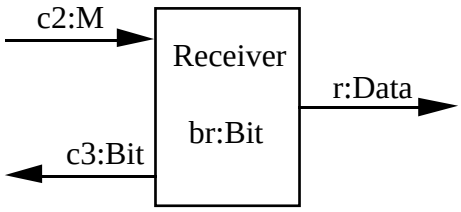


Fig. 21 Receiver as Data Flow Node

The syntactic interface and the attributes of the data space of the sender are given by the data flow node of Fig. 22.

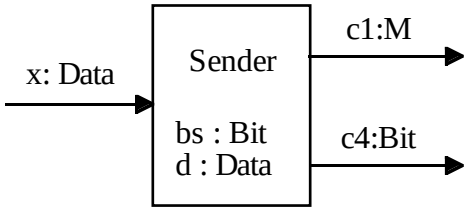


Fig. 22 Sender as Data Flow Node

The sender has a more sophisticated STD. It is shown in Fig. 23.

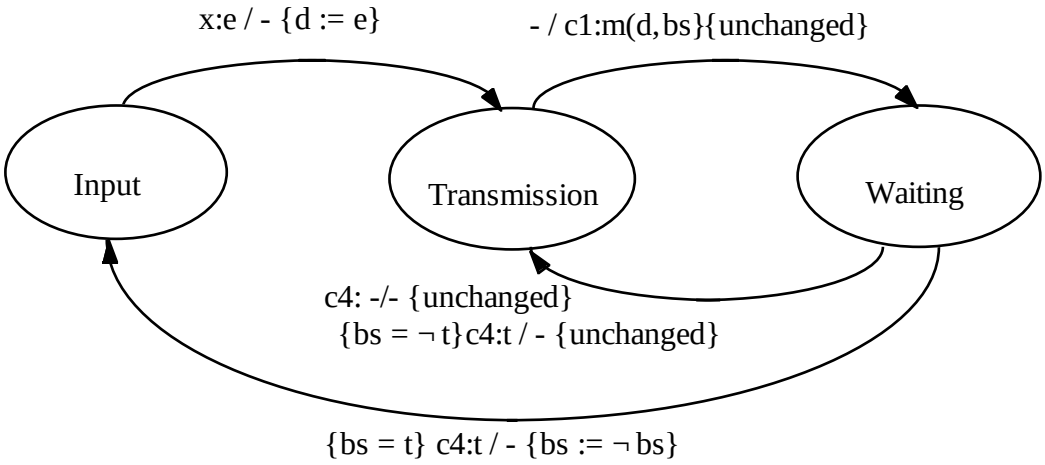


Fig. 23 STD of the Sender

In this STD we have a spontaneous transition modelling a soft time-out. It is needed for the case where messages are lost. Note, however, that here we have a cycle where all transitions are spontaneous. Therefore we better use the convention that spontaneous transitions consume one time tick or replace the transition -/- by c4:√/- requiring that a transmission is done only on empty input (more precisely no input over a certain time interval) on channel c4.

The component Transmitter is very simple. Fig. 24 shows it as a data flow node.

Fig. 24 Transmitter as Data Flow Node

Fig. 25 shows the STD of the transmitter. It does not include any fairness assumptions. Fairness assumptions can be included by prophecies in the state space or by additional equations for the transmitter.

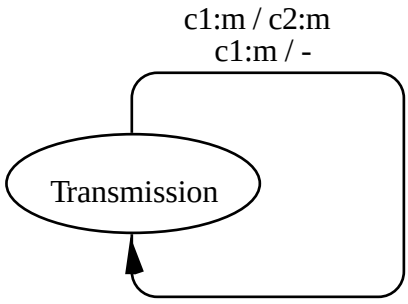


Fig. 25 STD of the Transmitter

References

[Abadi, Lamport 88]
M. Abadi, L. Lamport: The Existence of Refinement Mappings. Digital Systems Research Center, SRC Report 29, August 1988

[Abadi, Lamport 90]
M. Abadi, L. Lamport: Composing Specifications. Digital Systems Research Center, SRC Report 66, October 1990

[Andl 95]
B. Schätz, K. Spies: Formale Syntax zur logischen Kernsprachen der FOCUS-Entwicklungsmethodik, Technische Universität München, Institut für Informatik, SFB-Bericht Nr. 342/16/95 A, TUM-I9529, October 1995

[Broy 83]
M. Broy: Applicative Real Time Programming. In: Information Processing 83, IFIP World Congress, Paris 1983, North Holland Publ. Company 1983, 259-264

[Broy et al. 97]
M. Broy, R. Grosu, C. Klein: Reconciling Real-Time with Asynchronous Message Passing. To appear 1997

[Broy, Stølen 94]
M. Broy, K. Stølen: Specification and Refinement of Finite Dataflow Networks – a Relational Approach. In: Langmaack, H. and de Roever, W.-P. and Vytopil, J. (eds): Proc. FTRTFT'94, Lecture Notes in Computer Science 863, 1994, 247-267

[FOCUS 92]
M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, R. Weber: The Design of Distributed Systems - an Introduction to FOCUS. Technische Universität München, Institut für Informatik, TUM-I9203, Januar 1992

[Grosu et al. 96a]
R. Grosu, C. Klein, B. Rumpe: Enhancing the SysLab System Model with State. Technische Universität München, Institut für Informatik, TUM-I9631, 1996

[Grosu et al. 96b]
R. Grosu, C. Klein, B. Rumpe, M. Broy: State Transition Diagrams. Technische Universität München, Institut für Informatik, TUM-I9630, 1996

[Grosu, Rumpe 95]
R. Grosu, B. Rumpe: Concurrent Timed Port Automata. Technische Universität München, Institut für Informatik, TUM-I9533, 1995

[Harel 87]
D. Harel: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 1987, 231 -274

[Harel, Naamed 96]
D. Harel, A. Naamed: The STATEMATE Semantics of Statecharts. Unpublished manuscript 1996

[Lynch, Tuttle 87]
N. A. Lynch, M. R. Tuttle: Hierarchical Correctness Proofs for Distributed Algorithms. In: Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing, 1987

[Lynch, Stark 89]
N. Lynch, E. Stark: A Proof of the Kahn Principle for Input/Output Automata. Information and Computation 82, 1989, 81-92

[Lynch, Tuttle 89]
N. A. Lynch, M. R. Tuttle: Qn introduction to Input/Output automata. CWI Quaterly, 2(3), 1989, 219-246

[Lynch, Vaandrager 95]
N. A. Lynch, F. Vaandrager: Forward and Backward Simulations - Part I: Untimed Systems. Information and Computation 121(2), 1995, 214-233

[Müller, Scholz 96]
O. Müller, P. Scholz: Specification of Real-Time and Hybrid Systems in FOCUS. Technische Universität München, Institut für Informatik, TUM-I9627, 1996

[Nazareth et al. 96]
D. Nazareth, F. Regensburger, P Scholz: Mini-Statecharts: A Lean Version of Statecharts. Technische Universität München, Institut für Informatik, TUM-I9610, 1996

[Paech, Rumpe 94]
B. Paech, B. Rumpe: A new Concept of Refinement used for Behaviour Modelling with Automata. In: {M. Naftalin, T. Denvir, M. Bertran (eds.): FME'94, Formal Methods Europe, Symposium '94, Springer-Verlag: Berlin, LNCS 873, 1994

[Philipps, Scholz 97a]
J. Philipps, P. Scholz: System-Level Hardware Design with Mu-Charts. Accepted to: CHDL'97

[Philipps, Scholz 97b]
J. Philipps, P. Scholz: Compositional Specification of Embedded Systems with Statecharts. Accepted to: TAPSOFT'97

[Philipps, Scholz 97c]
J. Philipps, P. Scholz: Formal Verification of Statecharts With Instantaneous Chain Reactions. Accepted to: TACAS'97

[Rumpe 96]
B. Rumpe: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Ph.D. Thesis Technische Universität München, Fakultät für Informatik 1996. Published by Herbert Utz Verlag

[Rumpe, Klein 96]
B. Rumpe, C. Klein: Automata Describing Object Behavior. In: H. Kilov, W. Harvey (eds.): Specification of Behavioral Semantics in Object-Oriented Information Modeling. Kluwer Academic Publishers 1996, 265-286

[Scholz 96a]
P. Scholz: An Extended Version of Mini-Statecharts. Technische Universität München, Institut für Informatik, TUM-I9628, 1996

[Scholz 96b]

P. Scholz: A Light-Weight Formalism for the Specification of Reactive Systems. In: SOFSEM'96, Milovy, Czech Republic, 23-30 November 1996

[Scholz et al. 96]
P. Scholz, D. Nazareth, F. Regensburger: Mini-Statecharts: A Compositional Way to Model Parallel Systems. In: 9th International Conference on Parallel and Distributed Computing Systems, Dijon, France, 25-27 September 1996

[SDL 88]
Specification and Description Language (SDL), Recommendation Z. 100. Technical Report, CCITT 1988