T U M

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

State Transition Diagrams

Radu Grosu
Cornel Klein
Bernhard Rumpe
Manfred Broy

TUM-I9630
Juli 1996

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-07-1996-I9630-250/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

State Transition Diagrams *

Radu Grosu, Cornel Klein, Bernhard Rumpe, Manfred Broy
Institut für Informatik, Technische Universität München
80333 München, Germany
e-mail: (grosu|klein|rumpe|broy)@informatik.tu-muenchen.de

July 15, 1996

Abstract

In this paper, we present a general concept of state transition diagrams well-suited for various modeling purposes. Our notation is tailored for the description of asynchronous time-independent agents. We start by proposing a graphical and textual syntax, and define an abstract syntax for both notations. The semantics of state transition diagrams defined by translating the abstract syntax into timed port automata and to timed input/output relations on streams. To make the graphical notation practical, we partition the (possibly infinite) state space of the state transition diagrams with state predicates and define transitions with pre- and post-conditions.

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

Contents

1 Introduction

The purpose of this technical report is to define a description formalism for interactive components which is not only well-suited to be used in engineering practice but also provided with a solid formal basis. By an interactive component we mean a component which processes messages received on a bunch of input ports and sends the results along a bunch of output ports. The finite or infinite sequence of messages flowing along one port is called a communication history. The behavior of a component is modeled by the relationship between the histories of the input ports and the histories of the output ports. A formal model in which systems and their components are modeled this way is given in [KRB96] and [GKR96].

The behavior of a component can be modeled in a functional way by (sets of) stream processing functions, in a relational way with relations between input and output histories, in a trace oriented way with interleaved sequences of messages and last but not least, in a state based way [BDD$^+$93]. In contrast to the other techniques, the state based description explicitly uses the concept of component's state. Depending on the intention of use of the modeling technique, the state space of a component may be more or less abstract. A more abstract state space may be appropriate if the state automaton should be used to describe purely the semantic interface of a component, while a more concrete state space will be used if one aims at also specifying implementation details.

All state based specification techniques can be seen as a notation to define a state transition diagram (or automaton). It is the aim of this paper, to provide a general concept of state transition diagram. The semantics of the state transition diagrams is given in two different ways. Firstly by translating them into *timed port automata* [GR95]. Secondly, by translating them into *timed input/output relations* [Bro96]. We will also discuss the relationship between these approaches.

Moreover, we also present two notations for state transition diagrams, a textual one and a graphical one. Of course, the preferred notation in a project is a matter of taste and skill of the various persons reading and writing the specifications, and also of the available tool support. Therefore, the *notations* we present should only be viewed as a proposal, which may be modified and changed if necessary. The main goal of the paper is to provide the necessary syntactic and semantic *concepts* of state transition diagrams.

The paper is organized as follows: In Section 2 we introduce state transition diagrams using the example of interactive stacks. In Section 3 we fix the context in which state transition diagrams are used, and define the concrete and the abstract syntax for state transition diagrams. In Section 4 we define the semantics of state transition diagrams by translating them into timed port automata as well as by translating them into timed input/output relations. In section 4.3 we discuss some extensions. Finally in Section 5 we draw some conclusions and discuss future work.

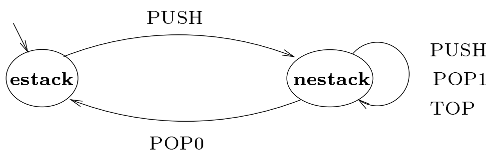
2 Example: Stacks

In order to informally present our ideas, we use an interactive stack as an example. An *interactive stack* is a component storing a stack of integers. The stack can only be accessed by sending to it messages. The message $Push(a)$ requests the stack to push the integer a on the top of the stack. The message Pop requests the stack to throw away its top element and the message Top requests the stack to deliver its top element. The input- and output ports of the interactive stack are named i and o. i and o carry messages of the datatypes In and Out, which using Gofer [Jon91] notation can be defined as follows:

```
data In  = Push Int | Pop | Top
type Out = Int
```

In the graphical notation annotated by declarations we propose, the stack can be specified as follows:

```
std stack = {
   input      i :: In
   output     o :: Out
   attributes l :: [Int]
   std-states estack  ≝ #l = 0;
              nestack ≝ #l > 0
   start      estack
```



name	precondition	input	output	postcondition
PUSH		$i?[Push(a)]$		$l' = a : l$
POP0	$\#l = 1$	$i?[Pop]$		$l' = []$
POP1	$\#l > 1$	$i?[Pop]$		$l' = rest\ \ l$
TOP		$i?[Top]$	$o![first\ \ l]$	$l' = l$

}

This specification can be understood as follows:

- The specification of a state transition diagram starts with the keyword **std**, followed by the name of the state transition diagram. The body of the specification is enclosed in braces $\{\ldots\}$.
- The stack has one input port i carrying messages of type In and one output port o carrying messages of type Out.
- The state space of a component is given by a set of attributes, each having a name and a type. In our case, the stack has an attribute l of type [Int], where [Int] denotes lists of integers.
- To specify the behavior of a component, we also use a finite set of control states (or "vertices"). Each control state is labeled by a condition which has to hold when the automaton is in that state. In our example, the control states are named estack and nestack, representing the empty stack and the non-empty stack, respectively.
- Initially, the automaton is in the start state estack.
- Transitions are given by a precondition, a set of input patterns, a postcondition and a set of output expressions. Input patterns, well-known from functional programming languages such as GOFER, provide a convenient way to define the message sequences read from the input port during the execution of a transition, and to decompose these sequences into their components. Output expressions are arbitrary expressions which specify the outputs of a transition on the different ports. In patterns and expressions, input and output ports are indicated merely by a question mark ? and respectively an exclamation mark ! following the port name, similarly to CSP [Hoa85].

  Because transitions may be bulky and because the same transition may occur more than once in a state transition diagram, transitions may be named (e.g. by PUSH) and arranged in a table.

  The table only defines macros. If one doesn't want to use macros, the transitions may be directly written within the state transition graph diagram. Transition TOP may be written for instance as

  $$\{\}\ i?[Top],\ o![first\ l]\ \{l' = l\}$$

  In both cases, in patterns, a one-element sequence $[a]$ may be written as a, and a pattern with an empty sequence [] may be omitted at all.
- The transition relation is specified by a finite, directed graph, were nodes represent control states and arcs are labeled by transitions. A start state is indicated by an arrow without source state.

  Intuitively, the automaton may go from state s to state s' if there exists a transition such that

- input x according to the input patterns of the transition has been received on the input ports and
- input x and state s satisfy the precondition and
- there exists an output y such that the postcondition for s, x, s' and y is satisfied and
- y is written to the output ports.

A more formal description of the syntax is given in section 3 and of the semantics in section 4.

3 Syntax

Before we start to propose the syntax of time-independent state transition diagrams, let us have a look at the basic concepts. In the remainder of this section we first introduce the abstract syntax, then the textual syntax in BNF Notation, and finally the graphical syntax.

3.1 Context

State transitions diagrams are not intended to be a stand-alone description technique, but we see them as part of an integrated suite of description techniques which are tailored for modeling different views of a system and its components. These views may for instance describe the data types of the system, the structure of a system, or the processes performed by a system.

In particular, we assume that there exists a *type language* for defining datatypes. Data types are used to provide a type discipline for state values and messages. The set of all types is denoted by the non-terminal <type-exp>. Throughout this paper, τ denotes type expressions. The interpretation of a type τ, denoted by $\mathcal{I}(\tau)$, is a set of *values*. For convenience instead of $\mathcal{I}(\tau)$ we often simply write D.

The set of all expressions is denoted by <expr>, and we assume that all expressions are well-typed. The set of patterns is a subset of the set of expressions, each pattern consisting only of constructive functions and variables. The interpretation of an expression e is denoted by $\mathcal{I}(e)$. If e is a closed expression of type τ, i.e. an expression without variables, $\mathcal{I}(e)$ is a value in $\mathcal{I}(\tau)$. The interpretation of an expression with free variables $v_1, \ldots v_n$ is a mapping from a record of values for the variables $v_1, \ldots v_n$ to $\mathcal{I}(\tau)$. If $\{v_1 : val_1, \ldots v_n : val_n\}$ is such a named product, and if the sequence $v_1 \ldots v_n$ of all free variables in e is clear from the context, $\mathcal{I}(e)(\{v_1 : val_1, \ldots, v_n : val_n\})$ is liberally abbreviated as $e(val_1, \ldots, val_n)$.

To represent lists of messages, we assume that there exists a type constructor $.^*$, constructing the type τ^* of all sequences of elements from a given type τ. If $e_1, \dots e_n$ are expressions of type τ, the notation $[e_1, \dots, e_n]$ is used to write a finite list of type τ^* with elements $e_1 \dots e_n$. A one element lists $[e]$ may be abbreviated as e.

We further presuppose the existence of a description technique for predicate expressions (formulas). The set of all predicate expressions is denoted by the non-terminal <pred-exp>. The predicate expressions may be arbitrary first- or higher-order formulas. We view predicate expressions as expressions of type Bool, such that the above conventions for the interpretation of closed and open expressions carry over to predicate expressions.

We require that if c denotes a variable of type τ^* for some τ and e an expression of type τ^*, then $c = e$ denotes a predicate expression. The variable c, called a *port variable*, denotes a port of a component from which the sequence of messages denoted by e is read or written. Predicate expressions of this kind are called *pattern predicates*.

If c is an input port, the expression e is only allowed to be a *pattern expression*. Pattern expressions, denoted by <pattern-exp>, are well-known from many functional languages such as GOFER [Jon91]. Usually, they may only be built from variables using constructor functions of the respective datatype. They therefore provide a convenient notation to decompose incoming message sequences into their components, and they may also be used to automate some checks, e.g. for overlapping predicates. An example for a pattern is $i = [Push(m)]$. It consists of the port variable i and the pattern $[Push(m)]$.

If c is an output port, e is not restricted to be a pattern, and therefore it may be an arbitrary expression.

Patterns for multiple input- and output ports are just conjunctions of patterns for single ports, i.e. predicate expressions of the form $c_1 = e_1 \land \dots \land c_n = e_n$.

The above syntactic categories may be provided by a functional programming language such as GOFER, as well as by an algebraic specification language such as SPECTRUM [BDD+93]. In SYSLAB, a simplified version of SPECTRUM, called MINI-SPECTRUM [Het96] is used to define datatypes. MINI-SPECTRUM is tailored for the needs of SYSLAB. Therefore, in the remainder of this section we assume that types, expressions, and predicate expressions are provided by MINI-SPECTRUM.

3.2 Abstract Syntax

In order to define the semantics of state transition diagrams as well as to reason about them, we now get rid of the syntactic sugar of the concrete graphical and textual notations used in this paper. The abstract syntax of state transition diagrams can easily be extracted from both. It is defined as follows:

Definition 1 (Abstract STDs) *An abstract state transition diagram is a tuple* $STD = (I, O, A, V, V_o, \mathcal{V}, \tau, T, \mathcal{T})$ *where:*

- I, O, A *and* V *are disjoint sets of identifiers for the input ports, the output ports, the attributes and the vertices (or control states).* $V_0 \subseteq V$ *is the set of start vertices.*
- $\mathcal{V}$ *is a mapping from the set of vertices* V *to the corresponding predicate expressions.*

  $\mathcal{V} : V \rightarrow <\textit{pred-exp}>$
- τ *is a typing mapping such that* τ_c *is the type of the identifier* $c \in (I \cup O \cup A)$.

  $\tau : I \cup O \cup A \rightarrow <\textit{type-exp}>$
- T *is a set of control transitions, i.e.* $T \subseteq V \times V$. *If* $(v, v') \in T$, v *is called the* source vertex *and* v' *the* destination vertex.
- $\mathcal{T}$ *is a mapping from the set of transitions to the corresponding transition labels.*

  $\mathcal{T} : T \rightarrow \mathcal{P}(<\textit{pred-exp}> \times <\textit{pred-exp}> \times <\textit{pred-exp}> \times <\textit{pred-exp}>)$

  If for a transition t, $(pre, ipat, oexp, post) \in \mathcal{T}(t)$, *then* $pre, post \in <\textit{pred-exp}>$ *are the precondition and the postcondition predicate expressions and* $ipat, oexp \in <\textit{pred-exp}>$ *are the input and the output pattern predicates. Powersets of pattern predicates are used, because the same control transition may be labeled by different automaton transition.*

This context free syntax will be restricted below by introducing some context conditions. □

Example 1 (The stack automaton, abstract syntax) *The abstract syntax for the state transition diagram of stacks is as follows:*

stack_{STD}=$(\{\mathsf{i}\}, \{\mathsf{o}\}, \{\mathsf{l}\}, \{\mathsf{estack}, \mathsf{nestack}\}, \{\mathsf{estack}\}, \mathcal{V}, \tau, T, \mathcal{T})$

where

τ_i=In, τ_o=Out, τ_l=$[\mathsf{Int}]$

$\mathcal{V}_{estack} \stackrel{\text{def}}{=} \#s.\mathsf{l}{=}0, \quad \mathcal{V}_{nestack} \stackrel{\text{def}}{=} \#s.\mathsf{l} > 0$

T=$\{(\mathsf{estack}, \mathsf{nestack}), (\mathsf{nestack}, \mathsf{nestack}), (\mathsf{nestack}, \mathsf{estack})\}$

$$
\begin{array}{lllll}
\mathcal{T}(\mathsf{estack}, \mathsf{nestack}) = \{ & (\mathsf{True}, & \theta.\mathsf{i}{=}[\mathsf{Push}(a)], & \theta.\mathsf{o}{=}[], & s'.\mathsf{l}{=}[a])\} \\
\mathcal{T}(\mathsf{nestack}, \mathsf{nestack}) = \{ & (\mathsf{True}, & \theta.\mathsf{i}{=}[\mathsf{Push}(a)], & \theta.\mathsf{o}{=}[], & s'.\mathsf{l}{=}a : s.\mathsf{l}), \\
 & (\#s.\mathsf{l} > 1, & \theta.\mathsf{i}{=}[\mathsf{Pop}], & \theta.\mathsf{o}{=}[], & s'.\mathsf{l}{=}rest(s.\mathsf{l})), \\
 & (\mathsf{True}, & \theta.\mathsf{i}{=}[\mathsf{Top}], & \theta.\mathsf{o}{=}[first(s.\mathsf{l})], & s'.\mathsf{l}{=}s.\mathsf{l})\} \\
\mathcal{T}(\mathsf{nestack}, \mathsf{estack}) = \{ & (\#s.\mathsf{l}{=}1, & \theta.\mathsf{i}{=}[\mathsf{Pop}], & \theta.\mathsf{o}{=}[], & s'.\mathsf{l}{=}[])\}
\end{array}
$$

□

The identifiers for the input ports, the output ports and the attributes of the component specified by a state transition diagram are given by the sets I, O and A. The types of the messages on the input- and output ports, as well as the types of the attributes, are given by the mapping τ.

The *data state space* S of an automaton is not monolithic but it is a named product. Let $A = \{a_1 \ldots a_k\}$. Then the data state is of record type $\{a_1 : \tau_{a_1}, \ldots, a_k : \tau_{a_k}\}$, i.e. it is a record type with a_i as record labels and with τ_{a_i} as types for the labels a_i. In the following we use $\prod_{a \in A} \tau_a$ where $A = \{a_1, \ldots, a_k\}$, as a compact notation for the above record type. If s is a data state of a record type with attribute a, then $s.a$ is the content of s for the attribute a.

Usually the set of data states of a component as well as the set of messages are infinite. To get a finite representation of a possible infinite state space we had to do some abstraction. We therefore introduced a finite set of *control states* or *vertices* V. The two different layers of states are connected via a predicate expression $\mathcal{V}(v)$ for each control state v. $\mathcal{V}(v)$ contains a free state variable s of type S. By interpreting the $\mathcal{V}(v)$ a set of data states is associated with the control state v. However, in contrast to other approaches [RK96] we do not require that the control states partition the data states. Arbitrary overlapping data states as well as data states with no corresponding control states are allowed.

Similar to the state layers, we get two layers of transitions. The finite representation of the state transition relation is a directed graph consisting of control states from V and of a set of edges $T \subseteq V \times V$, called *control transitions*. The non-empty subset $V_0 \subseteq V$ contains at least one *start state*. For each (abstract) control transition $t = (v, v'), t \in T$ a set of labels of the form $(pre, ipat, oexp, post)$ exist.

By interpreting these predicate expressions, the transitions between control states are mapped onto a fine grained set of *transitions on data states*, for short *data transitions*. This set is denoted by the predicate expression $\mathcal{P}_{(v,v')}$. $\mathcal{P}_{(v,v')}$ characterizes a set of data transitions. Therefore, $\mathcal{P}_{(v,v')}$ contains the following free variables:

- The variable $s : S$ denotes the source data state satisfying the source control state predicate, i.e. $\mathcal{V}_v(s)$ has to hold,
- The variable $s' : S$ denotes the destination data state satisfying the destination control predicate, i.e. $\mathcal{V}_{v'}(s')$ has to hold,
- The action variable $\theta : Act$ denotes the action performed during the transition (see below).

In this way, we can give a finite representation of an infinite transition space $S \times Act \times S$.

To keep the notation general, we allow sequences of messages to be sent or received within one transition. I/O-automata ([LS89, SHB96]) as well as the automata in [RK96] - that allow just one input message, but arbitrary sequences of output messages per transition - are therefore particular cases of our notation. Even spontaneous []-transitions are allowed. We call the exchange of input/output messages during one transition an *action* and denote the set of actions by Act. Hence, the actions we consider are the receiving of sequences of messages over a set of input ports I and the sending of sequences of messages over a set of output ports O. Let $C = I \cup O$ be the set of all ports. Each port $c \in C$ has a particular type τ_c. Therefore, the type of actions Act is the record type $\{c_1 : \tau^*_{c_1}, \ldots, c_k : \tau^*_{c_k}\}$ where $\tau^*_{c_i}$ is the type of finite sequences with elements of type τ_{c_i}. In the following we use $\prod_{c \in C} D^*_c$ as a compact notation for the above record type.

A *Message* usually contains a *message name* and a bunch of arguments, each of them with its own type. Regarding the message name as constructor with according "proper" arguments, the set of messages can itself be seen as a functional datatypes. Moreover, finite sequences of messages, that flow on ports, can as well be seen and used as functional datatypes. We therefore use *patterns* to decompose messages and sequences of messages into their components. Recall that from a logical point of view, a pattern is a predicate of the form $c = e$, where c denotes a sequence of messages read from port c, and e is an expression built from free variables using constructor functions. The free variables in e may be used in the precondition, in the postcondition as well as in the output patterns to refer to the components of the incoming messages.

Therefore, $\mathcal{P}_t$ where $t = (v, v')$, is usually considered as a predicate expression of the following form:

$$
\begin{aligned}
&\mathcal{P}_{(v,v')}(s, \theta, s') \equiv \bigvee_{(pre,ipat,oexp,post) \in \mathcal{T}(v,v')} \\
&\qquad \exists \phi, \psi : \mathcal{V}_v(s) \ \wedge \ \mathcal{V}_{v'}(s') \ \wedge \\
&\qquad\qquad ipat(\theta|_I, \phi) \wedge pre(s, \theta|_I, \phi) \wedge post(s, s', \theta, \phi, \psi) \wedge oexp(s, s', \theta, \phi, \psi)
\end{aligned}
$$

The predicate is the disjunction of predicates for the different labels of a control transition ($\bigvee_{\ldots}$). ϕ and ψ are existentially quantified tuples of values for the variables bound by the input and respectively the output patterns. The input pattern *ipat* may refer to the input part of action θ and to ϕ. The precondition *pre* may in addition also refer to the source data state s. The postcondition *post*, as well as the output pattern *oexp*, may refer to θ, ϕ, ψ, s as well as to the destination data state s'. This way, the postcondition can be used to relate source and destination data state.

This form of expressing transitions clearly distinguishes the input, the output, the next state and the conditions they have to satisfy. We therefore support this form in the concrete syntax. Burying the input in the precondition and the output in the postcondition could be in some cases advantageous but it would also lead to a loss of structuring information. However, if desired, input patterns and output expressions can be omitted, and specifications, consisting of pre- and postconditions only, are obtained.

We do not enforce the precondition to be an enabling condition. In our approach, it may happen that the precondition holds but no output and destination data state can be found that satisfy the postcondition. In addition, we also do neither require the explicit stated precondition $Pre_{(v,v')}$ to be consistent with the source state predicate $\mathcal{V}_v(s)$, nor do we require that the postcondition is consistent with the destination state predicate $\mathcal{V}_{v'}(s')$. This gives the specifier a lot of freedom, but may possibly make a state transition diagram somewhat less intuitive as for e.g. [RK96].

3.3 BNF Syntax

The example in the introduction already contained some textual parts. The difference between the textual notation and the graphical notation is that the state transition relation is described textually in the textual notation. Therefore, the stack example is written as follows:

Example 2 (Interactive stack, textual version)

```
std stack = {
  input       i :: In
  output      o :: Out
  attributes  l :: [Int]
  std-states  estack  ≝ #l = 0;
              nestack ≝ #l > 0
  start       estack
  transitions
  from estack,nestack to nestack
    {}        i?Push(a),               {l' = a : l}
  from nestack to nestack
    {#l > 1}  i?Pop,                   {l' = rest l};
    {}        i?Top,      o![first l]  {l' = l}
  from nestack to estack
    {#l = 1}  i?Pop,                   {l' = []}
}
```

Note that transition names are not used within the textual representation. □

The syntax for state transition diagrams in BNF notation is given below. Here, we assume that the nonterminals <stateid>, <inpid>, <outid>, <attid> and <stdid> denote identifiers and are defined elsewhere:

```
<spec>::= std <stdid> = {
   input        {<inpid> :: <type> //;}*
   output       {<outid> :: <type> //;}*
   attributes   {<attid> :: <type> //;}*
   std-states   {<state-spec> //;}*
   start        {<stateid> //,}+
   transitions  {<transition-spec> }* }

<state-spec>      ::= <stateid> ≝ <pred-exp>
<transition-spec> ::= from {<stateid>//,}* to {<stateid>//,}*
       { {<pre>} { <act>//,}* {<post> }//; }*
<act>             ::= <inpid>?<pattern-exp>
                  |   <outid>!<expr>
<pre>,<post>      ::= <pred-exp>
```

The translation of the concrete textual syntax to the abstract syntax is obvious and has already been exploited in the example. However we want to point out that:

- The input and output pattern predicates are obtained from the abstract syntax by replacing first ? and ! with = in each <act> of all <transition-spec>, and then by conjoining the resulting actions of a transition together. The shorthand for empty and one-element lists are expanded. For example the input expression $i_1?Push(a), i_2?Pop$ is transformed into the predicate expression $i_1 = [Push(a)] \wedge i_2 = [Pop]$.
- Moreover, in each predicate expression
  - each attribute a is replaced by $s.a$,
  - each primed attribute a' is replaced by $s'.a$,
  - each port name c is replaced by $\theta.c$.

  $Thevariabless$ and s' are the state variables for the current state and the next state, respectively, and θ is the action variable. Thus, the predicate expression $i_1 = [Push(a)] \wedge i_2 = [Pop]$ is further translated into $\theta.i_1 = [Push(a)] \wedge \theta.i_2 = [Pop]$.

- In addition, the equation $s'.b = s.b$ is introduced for all attributes b, if b' does not occur explicitly on the left side of a pattern predicate. Likewise, the condition $\phi.c = []$ is added for all not explicitly mentioned ports.

3.4 Graphical Syntax

The concrete textual representation can be visualized by a graphic representation straightforwardly. Vice versa, the graphic representation can easily be translated into the textual representation. We therefore regard them as equal with respect to their expressiveness. However the user is free to use the representation, that is more suitable in his or her context. The only difference is the use of macros for transitions and their definition in an additional table to simplify the state transition diagram.

4 Semantics

We model an interactive system by a network of autonomous components communicating via directed ports in a *time-synchronous* and *message-asynchronous* way. Time-synchrony is achieved by using a global clock splitting the time axis into discrete, equidistant time units. Message-asynchrony is achieved by allowing arbitrary, but finitely many messages to be sent along a port in each time unit.

We model the *communication histories* of directed ports by infinite sequences of finite sequences of messages. Each finite sequence represents the communication history within a time unit. The first finite sequence contains the messages received within the first time unit, the second the messages received within the second time unit, and so on. Since time never halts, any complete communication history is infinite.

If D is a set of messages, then $(D^*)^\infty$ is the set of all *complete* communication histories[1] over D, and $(D^*)^*$ is the set of all *partial* communication histories over D. In the following we abbreviate $(D^*)^\infty$ by $D^{\aleph}$. Given $\alpha \in D^{\aleph}$ and $i \in \mathbb{N}$, then $\alpha\downarrow_i \in (D^*)^*$ is the partial communication history consisting of the first i finite sequences in the complete communication history α.

Given a timed communication history α, we denote by $\overline{\alpha}$ the communication history with time information abstracted away. This is achieved by concatenating all the finite sequences in α. Clearly $\overline{\alpha} \in D^*$ if only finitely many finite sequences in α are different from $[]$ and $\overline{\alpha} \in D^\infty$ otherwise. We denote by $D^\omega = D^* \cup D^\infty$ the set of all *untimed* communication histories.

Each component has a set of input ports I and a set of output ports O. Each port

[1]Given an arbitrary set D, we denote by D^* and D^∞ the set of finite and respectively infinite sequences over D. $(D^*)^*$ is the set of finite sequences over D^* and therefore different from D^*.

is typed, hence each port $c \in I \cup O$ has a corresponding type of messages D_c. As a consequence, the complete input and output communication histories of components are *named sequence-tuples* contained in $\prod_{i \in I} D_i^{\aleph}$ and $\prod_{o \in O} D_o^{\aleph}$. The partial and the untimed input and output communication histories are named sequence-tuples contained in $\prod_{i \in I}(D_i^*)^*$ and $\prod_{o \in O}(D_o^*)^*$ and respectively $\prod_{i \in I} D_i^{\omega}$ and $\prod_{o \in O} D_o^{\omega}$.

Time abstraction $\overline{\varphi}$ and cutting $\varphi\downarrow_i$ are overloaded to named communication histories φ and to sets of communication histories in a pointwise style. In the following we also refer to named communication histories as to communication histories when no confusion arises.

4.1 Timed Port Automata Semantics

The timed port automata semantics of state transition diagrams is given by translating state transition diagrams into timed port automata. In Section 4.1.1 we introduce timed port automata (TPA). In section 4.1.2 we define the semantic translation.

4.1.1 Timed Port Automata

Timed port automata were defined in [GR95]. We only repeat the relevant definitions here and extend them to the case of typed ports. For a detailed treatment of timed port automata, their operational and denotational semantics see [GR95].

The interface of a timed port automaton consists of a set of typed input and output ports.

Definition 2 (Port signature) *Let I, O, H be pairwise disjoint sets of* input, output *and* hidden or internal ports *respectively. Denote by $C = I \cup O \cup H$ the set of all ports. Let D be a typing mapping assigning to each port $c \in C$ a set D_c of* data values. *A* port signature *is a tuple $\Sigma = (D, I, O, H)$.* □

Timed port automata are supposed to communicate asynchronously. As a consequence, they are not allowed to block their environment. Therefore, in every state, they have to react to every possible input. Since the automata are timed, each reaction (or transition) takes place in a constant, least time interval. The input or output associated to a transition may consist of a finite sequence of messages. The empty sequence denotes the absence of any message.

The set $\prod_{c \in C} D_c^*$ is called the set of all actions over Σ. Similarly $\prod_{c \in I} D_c^*$, $\prod_{c \in O} D_c^*$ and $\prod_{c \in H} D_c^*$ are called the sets of input, output and hidden actions over Σ respectively. Input and output actions are also called *external actions*.

Definition 3 (Timed port automaton) *A* timed port automaton *$A = (\Sigma, S, S_0, \delta)$ is a tuple where:*

- Σ *is a* port signature,
- S *is a set of* states,
- $S_0 \subseteq S$ *is the set of* start states,
- $\delta \subseteq S \times \prod_{c \in C} D_c^* \times S$ *is the transition relation, which is required to be* reactive*:*

  $\forall s \in S, \iota \in \prod_{c \in I} D_c^* : \exists t \in S, \theta \in \prod_{c \in C} D_c^* :\ (s, \theta, t) \in \delta \wedge \theta|_I = \iota.$ □

Note that timed port automata are semantic entities. This is also the case for the signature, where D_c are sets denoted by abstract syntax type expressions τ_c. Note, that a record type $\prod_{c \in C} D_c^*$ is defined by $\prod_{c \in C} D_c^* = \{\theta : C \to \bigcup_{c \in C} D_c^* \mid \theta(c) \in D_c^*\}$, the set of type respecting mappings.

If $(s, \theta, t) \in \delta$ we also write it as $s \xrightarrow{\theta}_\delta t$ or simply as $s \xrightarrow{\theta} t$ if δ is clear from the context.

Definition 4 (Execution, schedule, behavior) *An* execution *of an automaton A is an infinite sequence $s^0, \theta^0, s^1, \theta^1, \ldots$ such that $\forall i :\ s^i \xrightarrow{\theta^i} s^{i+1}$. We denote the set of executions by $execs(A)$.*

The schedule *$sched(\alpha)$ of an execution α is a subsequence of α containing only* actions *in α. We denote the set of schedules by $scheds(A)$.*

The behavior *$beh(\alpha)$ of an execution or schedule α is the subsequence of α containing only* external actions. *We denote by $behs(\alpha)$ the set of all behaviors of α.* □

Note that schedules and behaviors are named communication histories. Given an automaton A and an input sequence-tuple $\iota \in \prod_{i \in I} D_i^{\aleph}$. We denote the set of behaviors of A with input ι by $A[\iota]$. Formally:

$$A[\iota] = \{\alpha \in behs(A) \mid \alpha|_I = \iota\}$$

4.1.2 The Translation

The semantic of a state transition diagram STD is given by constructing a timed port automaton A. This automaton has an important property: its behavior depends only on the sequence of messages it receives and not on their granularity in time.

More formally, for any two input sequences α and β such that $\overline{\alpha} = \overline{\beta}$ the sets of untimed behaviors of the automaton are identical, i.e., $\overline{A[\alpha]} = \overline{A[\beta]}$.

Abstracting from time granularity is achieved by buffering. Buffering of input messages is performed because the sequence of messages that is processed by one transition of the state transition diagram δ' need not arrive within the same time unit. After an internal transition of δ' has fired, and the output messages have been produced, these messages may as well be buffered, modeling the fact that an internal

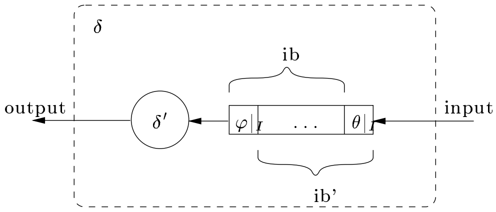


Figure 1: The timed port automaton A with input buffer

transition needs more than one unit of time to take place. Since buffering output messages does not change the behavior modulo time abstraction, it is not necessary to buffer output messages at all. Since it is simpler as well, we treat the variant with input buffers only. A graphical illustration where an input buffer is provided for each port $c \in I$ is given in Figure 1. In Figure 1, ib denotes the input buffer before a transition of the timed port automaton, and ib' denotes the input buffer after a transition of the port automaton. ib' is obtained from ib by removing the messages processed by the state transition diagram from the "front" and by adding the incoming messages within a time interval at the "rear".

We translate the abstract syntax of state transition diagrams first into a partial transition relation δ'. This untimed transition relation is then transformed into a timed port automaton, whose timed transition relation is called δ. The behavior of the timed port automaton can be then described as follows. If the input buffer contains a sequence φ such that (s, φ, s') is a transition in δ' then the automaton is allowed to perform this transition, to update the input buffer accordingly and to deliver the output. If there is another possible transition in δ', that is not yet enabled, but could perhaps become enabled by further input, the automaton is nondeterministically allowed to await this input as well. If no sequence φ can be found to perform a transition then there are two cases. In the first case, the input buffer can be extended to a sequence which has a corresponding transition in δ'. In that case the automaton remains in the same state and waits for future input. In the second case, such an extension is not possible. Then the automaton delivers arbitrary output from this point on. In other words, the component behaves chaotically.

Definition 5 (Semantics of STDs) *Given a state transition diagram* $STD = (I, O, A, V, V_0, \mathcal{V}, \tau, T)$*, and an interpretation function* $\mathcal{I}$*. The corresponding timed port automaton* $TPA = (\Sigma, S, S_0, \delta)$ *is defined as follows:*

$\Sigma = (D, I, O, \emptyset)$,

$S = \prod_{c \in I} D_c^* \times \prod_{a \in A} D_a$

$S_0' = \{([], s) \mid \exists n \in V_0 : \mathcal{V}_n(s)\}$.

$\delta = \{(\,(ib, s),\, \theta,\, (ib', s')\,) \mid \; Accept \;\vee\; Wait \;\vee\; Chaos \quad where$

The predicates Accept, Await and Chaos are defined and explained below. They refer to the partial untimed transition relation δ', which is defined as follows:

- $\delta' = \{(s, \theta, s') \;\mid\; \exists (v, v') \in T : \mathcal{P}_{(v,v')}(s, \theta, s')\}$

□

Let us comment the above definition. The internal transition relation δ' occurring in the definition of δ is untimed and contains transitions according to the given abstract state transition diagram. Just the predicates are expanded within δ'. As we see, the pre- and the postcondition are used symmetrically within the definition of δ'. Between pre- and the postcondition there is only the syntactic difference, that in the precondition the next state variable as well as the variables of the output may not be used. Thus it is always possible to remove the precondition and add it to the postcondition.

Let us now have a look at the definition of the external transition relation δ. First of all, the timed transitions are of form $(\,(ib, s),\, \theta,\, (ib', s')\,)$, where (ib, s) denotes the source state and (ib', s') the destination state. Both of these states contain the finite, untimed input buffer and the state of the internal state transition diagram δ'. δ is a disjunction of the predicates *Accept*, *Wait* and *Chaos*:

- The predicate *Accept* is defined as follows:

  $Accept \stackrel{def}{=} \exists \varphi : s \xrightarrow{\varphi}_{\delta'} s' \;\wedge ib^\frown \theta|_I = \varphi|_I{}^\frown ib' \;\wedge \theta|_O = \varphi|_O$

  It describes all external transitions that also have a corresponding internal transition. In this case, there is an internal action φ that enables δ' in state s to perform a transition resulting in destination state s'. The input part of the internal action $\varphi|_I$ is removed from the input buffer, and the output part $\varphi|_O$ is immediately sent. Note that $\varphi|_O$ may contain simultaneous reactions to inputs arrived in $\theta|_I$.

- The predicate *Wait* is defined as follows:

  $Wait \stackrel{def}{=}$
  $ib^\frown \theta|_I = ib' \wedge s = s' \wedge \theta|_O = [] \wedge (\exists \varphi, \psi, s'' : s \xrightarrow{\varphi}_{\delta'} s'' \;\wedge \varphi|_I \sqsubseteq ib'^\frown \psi \;\wedge \varphi|_I \not\sqsubseteq ib')$

  It allows the external transition relation to wait for further input messages and therefore to emit no messages and to perform no internal transition, if the input buffer ib' can be enlarged, such that a prefix $\varphi|_I$ of the enlarged buffer causes the internal transition relation δ' to fire. It is assumed that $\varphi|_I$ is initially not contained in ib'.

- Finally the predicate *Chaos* is defined as:

  $$Chaos \overset{def}{=} ib\frown\theta|_I = ib' \wedge s = s' \wedge \neg\exists\varphi, \psi, s'' : s \xrightarrow{\varphi}_{\delta'} s'' \;\wedge \varphi|_I \sqsubseteq ib'\frown\psi$$

  It handles the case that no internal transition is possible at the moment and no additional input ψ will ever enable an internal transition. In this case, arbitrary output (chaos) is allowed by the transition relation δ of the timed port automaton. transition system. The internal relation δ' just starves within the same state and while the timed port automaton just enlarges its input buffer.

  We therefore implicitly get a set of error states E, that can be characterized as follows:

  $$E(ib, s) = \neg\exists\varphi, \psi, s'' : \;\varphi \sqsubseteq ib\frown\psi \;\wedge\; s \xrightarrow{\varphi}_{\delta'} s''$$

  Obviously E is closed under concatenation of arbitrary input to ib. This simply means, that once trapped in an error state, the set of error states is never left again.

The predicates *Accept*, *Wait* and *Chaos* are connected by an $\vee$, so if more than one possibility exist, either of this possibilities can be chosen nondeterministically. This means that even if there are sufficient messages in the input buffer for an internal transition to fire, if there is another one, that could possibly fire some time in the future, if more messages arrive, the timed port automaton is allowed to wait. Since we do not have fairness incorporated, it may happen that external transition relation waits infinitely long, if no further message arrives. The user is responsible to prevent this situation, or to cope with it.

For example, let us look at the timed port automaton A for the state transition diagram of stacks.

Example 3 (The stack automaton) *Given the abstract syntax for stacks from Section 3.2. Then we can easily construct the partial transition relation as follows:*

$$\delta' \subseteq \prod_{c\in\{l\}} D_c \times \prod_{c\in\{i,o\}} D_c \times \prod_{c\in\{l\}} D_c$$

$$\begin{array}{llllllll}
\delta' = \{(st, \theta, st') \mid & & & & & & & \\
\exists a.\,\texttt{\#}s.l\texttt{=}0 & \wedge\ \texttt{\#}s'.l\texttt{>}0 & \wedge\ \texttt{True} & \wedge\ \theta.\texttt{i=[Push(}a\texttt{)]} & \wedge\ \theta.\texttt{o=[]} & \wedge\ s'.\texttt{l=[}a\texttt{]} & \vee \\
\exists a.\,\texttt{\#}s.l\texttt{>}0 & \wedge\ \texttt{\#}s'.l\texttt{>}0 & \wedge\ \texttt{True} & \wedge\ \theta.\texttt{i=[Push(}a\texttt{)]} & \wedge\ \theta.\texttt{o=[]} & \wedge\ s'.\texttt{l=}a \texttt{ : } s.\texttt{l} & \vee \\
\texttt{\#}s.l\texttt{>}0 & \wedge\ \texttt{\#}s'.l\texttt{>}0 & \wedge\ \texttt{\#}s.l\texttt{>}1 & \wedge\ \theta.\texttt{i=[Pop]} & \wedge\ \theta.\texttt{o=[]} & \wedge\ s'.\texttt{l=}rest(s.\texttt{l}) & \vee \\
\texttt{\#}s.l\texttt{>}0 & \wedge\ \texttt{\#}s'.l\texttt{>}0 & \wedge\ \texttt{True} & \wedge\ \theta.\texttt{i=[Top]} & \wedge\ \theta.\texttt{o=[}first(s.\texttt{l})\texttt{]} & \wedge\ s'.\texttt{l=}s.\texttt{l} & \vee \\
\texttt{\#}s.l\texttt{>}0 & \wedge\ \texttt{\#}s'.l\texttt{=}0 & \wedge\ \texttt{\#}s.l\texttt{=}1 & \wedge\ \theta.\texttt{i=[Pop]} & \wedge\ \theta.\texttt{o=[]} & \wedge\ s'.\texttt{l=[]} & \}
\end{array}$$

Each conjunct describes a set of transitions $(st, \theta, st') \in \delta'$. The disjunctions take the union of these sets. Note that we allow vertex predicates to overlap as well as to *be false. The input patterns for different transitions may be overlapping (unifyable) as well.*

The timed port automaton $A = (\Sigma, S, S_0, \delta)$ is then defined as follows:

$$\begin{aligned} \Sigma &= (D, \{i\}, \{o\}, \{\}), \\ S &= D_i \times D_l, \\ S_0 &= \{(b, st) \mid st.l = [] \wedge b.i = [] \} \end{aligned}$$

The transition relation δ is defined as above by using δ'. □

4.2 Timed IO-Relation semantics

The timed input/output relation semantics of state transition diagrams is given by translating state transition diagrams into timed input/output relations. In Section 4.2.1 we introduce timed input/output relations. In section 4.2.2 we define the semantic translation.

4.2.1 Timed IO-Relations

Timed input/output relations are defined in [Bro96]. We only repeat the relevant definitions here and extend them to the case of typed ports. For a detailed treatment of timed input/output relations, see [Bro96].

The behavior of an interactive component is given in this case by a relation between complete input communication histories and complete output communication histories. For convenience, the relation is written as a *set valued function*

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

mapping complete input histories to sets of complete output histories.

Definition 6 (Timed and consistent relations) *We call an input/output relation $F : \prod_{i \in I} D_i^{\aleph} \rightarrow \wp(\prod_{o \in O} D_o^{\aleph})$* timed, *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 time guarded*, if the following stronger condition holds*

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

We call an input/output relation fully consistent *or* realizable *if there is a time guarded function $f : \prod_{i \in I} D_i^{\aleph} \rightarrow \prod_{o \in O} D_o^{\aleph}$ such that for all input histories $\alpha : \prod_{i \in I} D_i^{\aleph}$*

$$f(\alpha) \in F(\alpha)$$

We call F inconsistent *if there is an input history $\alpha : \prod_{i \in I} D_i^{\aleph}$, $F(\alpha) = \emptyset$ and* weakly consistent *if it is not inconsistent.* □

In [GR95] we show that all port automata A are timed (called weakly pulse-driven there), and therefore the set $behs(A)$ of behaviors denotes a timed, weakly consistent input/output relation. If a port automaton A is moreover time-guarded (strongly pulse-driven), then the set $behs(A)$ of behaviors denotes a time-guarded and fully consistent input/output relation.

4.2.2 The Translation

The semantics of a state transition diagram STD is given by constructing a timed input/output relation F. As with timed port automata, this relation has an important property: its definition depends only on the sequence of input messages messages and not on their granularity in time.

More formally, for any two input sequences α and β such that $\overline{\alpha} = \overline{\beta}$ the sets of untimed outputs of the relation F are identical, i.e., $\overline{F(\alpha)} = \overline{F(\beta)}$. In case of input/output relations, additional buffering is not necessary, because the relations are already defined over infinite sequences of finite sequences of messages.

As with timed port automata semantics, the abstract syntax of a state transition diagram is first translated into a partial transition relation δ. This untimed transition relation is then transformed in a timed input/output relation F parameterized over the data state space S of the state transition diagram

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

Given a state s we write for convenience F_s for $F(s)$. The behavior of the timed input/output relation F_s can then be described as follows. If the time abstracted input communication history contains a prefix sequence φ such that $(s, \varphi{+}\psi, s')$ is a transition in δ, then the time abstracted output of the relation F_s contains the prefix ψ. Moreover, the relation F_s switches to $F_{s'}$. If there are several alternatives, one of these alternatives is nondeterministically chosen. If no finite prefix sequence φ can be found such that $(s, \varphi{+}\psi, s')$ is a transition in δ, then the output of F_s is arbitrary, i.e. the component behaves chaotically.

Definition 7 (IO-Relation Semantics of STDs) *Let a state transition diagram $STD = (I, O, A, V, V_0, \mathcal{V}, \tau, T)$ be given. The corresponding timed input/output relation F is the largest time guarded relation (with respect to set inclusion) such that*

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

$$F_s(\alpha) = \{\beta \in \prod_{o \in O} D_o^{\aleph} \mid$$

$$(\exists \varphi \in \prod_{i \in I} D_i^*, \psi \in \prod_{o \in O} D_o^*, s' \in S, \alpha' \in \prod_{i \in I} D_i^{\aleph}, \beta' \in \prod_{o \in O} D_o^{\aleph} :$$

$$\overline{\alpha} = \varphi\hat{\ }\overline{\alpha'} \wedge \overline{\beta} = \psi\hat{\ }\overline{\beta'} \wedge s \xrightarrow{\varphi+\psi}_{\delta} s' \wedge \beta' \in F_{s'}(\alpha')) \quad \vee$$

$$\neg\exists \varphi \in \prod_{i \in I} D_i^*, \psi \in \prod_{o \in O} D_c^*, s' \in S : \varphi \sqsubseteq \overline{\alpha} \ \wedge \ s \xrightarrow{\varphi+\psi}_{\delta} s'\}$$

holds. The partial untimed transition relation δ is constructed as before:

$$\delta' = \{(s, \theta, s') \ \mid \ \exists (v, v') \in T : \mathcal{P}_{(v,v')}(s, \theta, s')\}$$

□

4.2.3 Discussion

Both semantic definitions are equal in the sense that they describe the same time abstracted behavior:

$$\forall \alpha : \overline{A[\alpha]} = \bigcup_{s \in V_o} \overline{F_s(\alpha)}$$

However, the important point about the second semantic definition is that we require F to be time guarded. This can easily be observed in the following example.

Example 4 *Suppose the state transition diagram contains two states s_1 and s_2 and one transition $s_1 \xrightarrow{i?aq} s_2$. Let the input stream on i contain $a \frown \epsilon^\infty$. In this case, if we had not required F to be time guarded, the component would behave chaotically, i.e. $F_{s_1}(i) = D_o^{\aleph}$ would hold. Time guardedness ensures that the component delivers only one output, ϵ^∞, as it does the constructed timed port automaton.* □

We think that both semantics have its advantages. The timed port automaton defines the behavior in a constructive way, which may be easier to understand for many users of state transition diagrams. On the other hand, the relational semantics is more abstract, since it is not based on the explicit construction of a buffer. This may make reasoning about a specification easier.

4.3 Non-Overlapping State Predicates

Often, it is convenient to use non overlapping state predicates. This can be easily achieved by extending the state with a control variable (attribute) *ctrl* with an enumeration type. This enumeration type can be constructed by using state names as constants. Then the state of a component can be immediately retrieved from its data state. For example, in the case of stacks one can use the datatype *StackCtrl* and redefine the state predicates:

`data StackCtrl = Estack` | `Nestack`

`estack'`$(s) \stackrel{\text{def}}{=}$ *ctrl* `= Estack` $\wedge$ `#`s `= 0`
`nestack'`$(s) \stackrel{\text{def}}{=}$ *ctrl* `= Nestack` $\wedge$ `#`s `> 0.`

Surely, in case of stacks this extension is not necessary because the predicates were already non overlapping.

5 Conclusions and Further Work

In this article we have defined the syntax and the semantics of time independent state transition diagrams. The semantical framework developed is however powerful enough to allow both the definition of composition and the extension to state transition diagrams with time constraints. These topics are of high priority for future development. Moreover, we have to investigate refinement and hierarchical structuring (à lá state-charts).

Another important topic is the definition of constructive refinement techniques for state transition diagrams, that correlate with the refinement relation on the used semantics.

Equally important, and already at work, is the implementation of a tool prototype, that allows us to test and to demonstrate the practical usefulness of the syntax and semantics of the description technique presented in this paper.

Acknowledgments

We thank Eva Geisberger, Barbara Paech, Jan Philipps, Alexander Schmidt and Veronika Thurner for discussions and for reading draft versions of this paper.

References

[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 – revised version –. SFB-Bericht 342/2-2/92 A, Technische Universität München, January 1993.

[Bro96] M. Broy. Some Relational Hocus Pocus with FOCUS. To be published, 1996.

[GKR96] R. Grosu, C. Klein, and B. Rumpe. Enhancing the SYSLAB system model with state. Technical Report TUM-I9631, Technische Universität München, 1996.

[GR95] R. Grosu and B. Rumpe. Concurrent timed port automata. Technical Report TUM-I9533, Technische Universität München, 1995.

[Het96] R. Hettler. Description techniques for data in the SYSLAB method. Syslab Project, to appear as Technical Report, 1996.

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

[Jon91] M. P. Jones. Introduction to Gofer 2.20. Technical report, Yale University, September 1991.

[KRB96] C. Klein, B. Rumpe, and M. Broy. A stream based mathematical model for distributed information processing systems. In Elie Najm, editor, *1st Workshop on Formal Methods for Open Object-based Distributed Systems, Paris 1996. Proceedings.* Chapmann & Hall, 1996. to appear.

[LS89] N. Lynch and E. Stark. A Proof of the Kahn Principle for Input/Output Automata. *Information and Computation*, 82:81–92, 1989.

[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. to appear.

[SHB96] B. Schätz, H. Hußmann, and M. Broy. Graphical Development of Consistent System Specifications . In Marie-Claude Gaudel James Woodcock, editor, *FME'96: Industrial Benefit and Advances In Formal Methods*, pages 248–267. Springer, 1996. Lecture Notes in Computer Science 1051.