A Simple Toy Example of a Distributed System: On the Design of a Connecting Switch

Thomas F. Gritzner

Institut für Informatik
Technische Universität München
Postfach 20 24 20, Arcisstr. 21
W–8000 München 2 (Germany)

Abstract

In this paper the development life-cycle of a design method for distributed systems is explained in close connection with the example of the connecting switch. A connecting switch is a system where stations may get connected, may then send actions, and may get disconnected; it exhibits the behaviour of a very simple protocol. A development life-cycle commonly includes four phases: (1) requirement specification; (2) design specification; (3) abstract program design; (4) concrete program design. This requires collecting various description formalisms into one design method. Each transition of one phase to a more concrete one is guided by a design decision; each verification of the corresponding transition result remains as proof obligation. By the case study of a connecting switch the application of the design method will be demonstrated by treating this example within all four phases in order to make the design method more transparent.

1. Introduction

A distributed system consists of a family of components that are able on the one hand to work independently (*concurrency*) and on the other hand to communicate with each other in order to exchange data or to synchronize for avoiding conflicts (*cooperation*). A development method for distributed systems provides a framework for organizing the development of a distributed system. Systematic development of distributed systems is important, because a distributed system with more than one active parallel components is hard to test appropriately.

A suitable development method should provide an appropriate collection of development steps descending several levels of abstraction guided by design decisions. Usually, one may list the following levels integrated in a development life-cycle starting from the informal level as the most abstract one:

0. informal problem description
1. requirement specification
2. design specification
3. abstract program (aiming at applicative programming languages)
4. concrete program (aiming at procedural programming languages for efficient implementations)

Different levels of abstraction may require various description formalisms to be integrated into one design method. The formalisms of the approach to be presented are listed as follows:

1. trace specifications (predicates over traces)
2. functional specifications (stream processing functions)
3. implementation (dealing with abstract & concrete programs)

Trace specifications aim at the description of the required behaviour of a distributed system. *Functional specifications* allow to specify networks of agents representing the family of components of the concerning distributed system. The *implementation* step includes two notational programming languages: an applicative one for writing abstract programs, and a procedural one for writing concrete programs.

A design method should also support the derivation of a less abstract specification from a more abstract one. This comprises exactly the transition step from one level of abstraction to the next to be also described formally. These transition steps are two-way: The downwards direction is guided by a design decision, the upwards direction is the verification of the derived more concrete specification against the more abstract one. The latter one requires a collection of proof techniques as formal representation. Closely related to transitions are *refinements* of a specification within one level of abstraction. In fact, any design decision leads to a refinement of a specification in the usual sense.

When building a family of specifications along the development life-cycle one observes particular (pairs of) aspects as follows:

- *Safety/Liveness:* The behaviour of a distributed system may be captured by a set of histories. Mathematically, histories are represented by finite or infinite sequences of actions, while observations correspond to prefixes of histories. A **safety** property is a predicate that holds for a history iff none of the finite observations violates the condition, which rules out unwanted reactions of the system. A **liveness** property is a predicate that requires that each finite observation can be continued to a liveness correct history. These two aspects can be combined together by demanding for liveness that only *safety* correct finite observations need to be continued to a liveness and also *safety* correct history.

- *System/Environment:* The following three pairs of aspects are closely connected. The first concerns the fundamental decomposition of a distributed system in the whole into two large components. One is called the **system** in the sense of a *main task*, i.e. a "system" includes all components to be implemented. The other is called the **environment**, which works rather like an operating system, because it has to prepare some services for the "system".

- *Open/Closed System View:* A distributed system consists of a collection of components that interact. While the "environment" part feeds actions to the "system" part, the "system" part gives back the amount of reactions to the "environment". Thus, a distributed system in the whole may be regarded as a *closed* circulation; this is what is called the **closed system view**. The **open system view** rests on the consideration of the specification only of certain selected components of the concerning distributed system; the treatment of the specification of the "system" part only appears as an important special case.

- *Rely/Guarantee (Assumption/Commitment):* The open system view, as a generalisation of the division into system and environment, leads to a division of the distributed system into the considered and into the unconsidered part. Sometimes the specification of the considered part may be done without respect to the unconsidered, i.e. the rest may behave arbitrarily. But, as e.g. in the case of the circulation thought, in the most cases the unconsidered part has to follow some restrictions: the considered part **relies** on the right behaviour of the unconsidered one, and only then it **guarantees** the right reaction. This concept is comparable to *Hoare triples* $\{p\}STEP\{q\}$, which says "if condition p is satisfied, then the program step $STEP$ leads to condition q after its execution".

By our simple example of a connecting switch we want to present a large cross-section of the design method meeting every level of abstraction and we will also give formalizations and examples of the just discussed aspects of distributed system design. This case study is intended to be not only a kind of cross section of a "state of the art" of the chosen approach, but also a feed-back, i.e. this case study shall show how the approach can be refined for a certain class of special cases.

This paper is organized as follows:

The following section contains introduction of basic notions and basic notations. In order to deal with definitions more uniformly, we propose a notational framework similar to algebraic specification techniques.

Then, in section 3 we first introduce our example of a fairly simple protocol: the connecting switch. We also treat the techniques of building trace specifications: pure trace logic, macro techniques for a more state-oriented view, transition systems as a directly state-oriented concept, TRANSACT for specifying systems that execute a fixed recurrent sequence of actions more conveniently. The different forms of trace specifications of our example are verified one against another. Furthermore, the aspects of modifiability of trace specifications with respect to manipulations of the action set are dealt with: action refinement, action enrichment, notion of persistent trace specifications, and the specification with REACTION, which is a technique to build specifications with actions pressed into a more uniform form.

Mechanisms to step forward from a requirement specification to a design specification is explained in section 4. First, the technique of component-oriented trace specification is explained, which allows to form component trace specifications according to the selected separation of components from the distributed systems. Then, the notion of a differentially strategic system is introduced; differentially strategic systems give insight to the causality structure of a component trace specification in order to enable the construction of an agent specification from the component specification. The last part of the section is dedicated to the proper application of component-orientation to our example: a general design decision comes out from the system/environment view as described above; the component trace specifications of "system" and "environment" for the connecting switch are given and proved correct.

Section 5 deals with two forms of functional specifications. The first one is that of partial order processing specifications, which is intended to express and exploit the causality of inputs or of outputs. The other is the well-known one of stream processing functions. But, the continuity requirement of stream processing functions is strong enough to establish the immediate transition from design specification to the first implementation level of abstract programs.

The two phases of implementation are described in section 6: abstract programs written in functional style, concrete programs written in procedural style. Finally, the transition from abstract program to concrete program is sketched by giving examples for transformation rules.

2. Basic Structures

Streams and Operations on Streams

Let a set $\mathcal{A}$ be given, then $\mathcal{A}^*$ will denote the set of all finite sequences over $\mathcal{A}$, and $\mathcal{A}^\infty$ denotes the set of all infinite sequences over $\mathcal{A}$. The set of all **streams** over $\mathcal{A}$ is given by $\mathcal{A}^\omega := \mathcal{A}^* \cup \mathcal{A}^\infty$.

On streams we use the following operations:

ε	denotes the empty sequence,
$\langle a_1, a_2, \ldots \rangle$	denotes for $a_i \in \mathcal{A}$ the sequence made of $a_1, a_2, \ldots$,
$s\&t$	denotes the sequence yielded by the concatenation of $s, t \in \mathcal{A}^\omega$,
ft s	denotes the *first* (left-most) element of the sequence $s \in \mathcal{A}^\omega \setminus \{\varepsilon\}$,
rt s	denotes the *rest* of the sequence, i.e. the sequence omitting the first element if any (i.e. rt $\varepsilon := \varepsilon$),
lt s	denotes the *last* (right-most) element of the finite sequence $s \in \mathcal{A}^* \setminus \{\varepsilon\}$,
ld s	denotes the *lead* of the sequence, i.e. the sequence omitting the last element if any (i.e. ld $s := s \iff s = \varepsilon \vee s \in \mathcal{A}^\infty$),
$\#s$	denotes the length of $s \in \mathcal{A}^\omega$, i.e. the *number* of its elements,

$\mathcal{S}©s$	denotes for $\mathcal{S} \subseteq \mathcal{A}$ and $s \in \mathcal{A}^\omega$ the "subsequence" of s consisting of elements of $\mathcal{S}$ only (*filtering*),
$\mathcal{S} \S s$	abbreviates $\#(\mathcal{S}©s)$,
x **in** s	abbreviates the condition $\{x\} \S s > 0$.

Moreover, on streams we define the **prefix ordering** $\sqsubseteq$ by:

$$\forall\, s, t \in \mathcal{A}^\omega\colon\ s \sqsubseteq t \ :\Longleftrightarrow\ \exists\, s' \in \mathcal{A}^\omega\colon\ s \& s' = t.$$

A function $f\colon \mathcal{A}^\omega \to \mathcal{B}^\omega$ is called a **stream processing function** iff f is **prefix continuous**, i.e. continuous wrt. $\sqsubseteq$. Let $(\mathcal{A}^\omega \to \mathcal{B}^\omega)$ denote the set of all stream processing functions from $\mathcal{A}^\omega$ into $\mathcal{B}^\omega$.

Additional Notations

We make use of the following notations which occur usually in connection with functional programming languages:

$$\begin{array}{rcl} a \textbf{ where } b & := & \text{<<expression } a\text{, where condition } b \text{ is satisfied>>;} \\ (\ b \textbf{ ? } e_1 : e_2\ ) & := & \begin{cases} e_1, & \text{if condition } b \text{ holds,} \\ e_2, & \text{if condition } b \text{ does not hold} \end{cases}; \\ \phi[x_0 \leftarrow v_0] & := & \phi' \textbf{ where } \forall x\colon \phi'(x) = (\ x = x_0 \textbf{ ? } v_0 : \phi(x)\ ). \end{array}$$

$(\ b \textbf{ ? } e_1 : e_2\ )$ sometimes serves also as abbreviation of the formula $(b \Rightarrow e_1) \wedge (\neg b \Rightarrow e_2)$. We also use different ways of denoting function application: $\phi(x), \phi[x], \phi\{x\}$, as usual, or $\phi.x$, or even ϕ_x.

A Notational Framework for Introductions

If any definition within a specification has to be given, we use algebraic specification style so that each such definition looks like a part of a complete algebraic specification. For this we use the following notations:

[used] sort *Sort*
: marks the introduction of a sort *Sort*; *Sort* is a set which contains exactly one undefined element usually denoted by $\perp_{Sort}$. The **used** attribute means that *Sort* has no further particular requirements to meet; usually, *Sort* is assumed to be isomorphic to the basic sort Nat of natural numbers by a function also denoted by *Sort* (e.g. $Sort(\perp_{Nat}) = \perp_{Sort}$).

fct *constant*: *Sort*
: introduces a constant *constant*; *constant* is an element of *Sort*; *Sort* may also be a composed sort like $Sort_1 \times Sort_2$, but not a functional sort.

fct *function*: *functionality*
: introduces a strict continuous function *function* with functionality *functionality*; *functionality* has not only the common form $Sort_1[\times \ldots \times Sort_k] \to Sort_0$, but also special forms to indicate the notation of the application of *function*.

fun *operation*: *functionality*
: introduces a function *operation* with functionality *functionality*; *operation* need neither satisfy any strictness nor any continuity condition except particularly marked: e.g. $\underline{Sort}$ says that *operation* must be strict and continuous wrt. this argument place.

rel *predicate*: *domain*
: introduces a predicate *predicate* over the set *domain*; *predicate* may be used as a set, namely as subset of *domain*, but predicate application is usually denoted by $predicate[d_1, \ldots, d_k]$ instead of $(d_1, \ldots, d_k) \in predicate$.

such that *formula*
indicates that the objects mentioned in the formula *formula* have to satisfy *formula*, i.e. it establishes a mobile notation for the satisfaction of axioms as "laws".

Predefined objects: Sorts: Nat (natural numbers 0, 1, ...), $\overline{Nat}$ (closed line of natural numbers $0, 1, \ldots, \infty$ linearly ordered), $Bool$ (boolean values uu, tt, ff where $uu = \perp_{Bool}$), $Pot(Sort)$ (power set of $Sort \setminus \{\perp_{Sort}\}$) etc.; predicates: δ_{Sort} (definedness predicate concerning $Sort$), etc.

As an example we treat the specification of streams over a sort S:

used sort S;
sort S^ω;
fct ε: S^ω;
fct ft: $\downarrow S^\omega \to S$;
fct rt: $\downarrow S^\omega \to S^\omega$;
fun &: $S \overset{\downarrow}{\times} \underline{S^\omega} \to S^\omega$ /* & is an overloaded operation symbol, here = append */
such that $\forall x:S, s:S^\omega \left[\begin{array}{l} \neg\delta_S[\text{ft } \varepsilon] \wedge \text{rt } \varepsilon = \varepsilon \\ \wedge\ \text{ft } x\&s = x \wedge (\ \delta_S[x]\ \mathbf{?}\ \text{rt } x\&s = s : x\&s = \varepsilon\ ) \end{array} \right]$;

3. Introduction to the Example and Trace Specifications

In this section we introduce the example of the connecting switch. In order to try a more formal introduction we also treat the variants of trace specifications of this system.

3.1 A Very Simple Protocol: The Connecting Switch

The concerning example in this form, including the first variant of a trace specification of it, is taken from [Broy 88, 3.1.1–3.1.2]. The informal description reads as follows:

> "A connecting switch is a system where stations may get connected, may then send messages and may get disconnected."

We present now a more formal treatment by giving a first requirement specification of it, i.e. a first variant of trace specification. By this we explain the notion of a trace specification and demonstrate by which means such a specification can be composed.

3.1.1 Fundamental Remarks to Trace Specifications

Traces and Trace Specifications: Trace specifications are intended to describe the behaviour of a distributed system in an abstract manner such that one needs not think about placing the components. For this, a behaviour is expressed by a complete observation of the history of the *actions* executed by the system. Such a history is made under the assumption of a sequential observer, i.e. parallel sequences of executed actions are sequentialised. Instead of a history we speak of a *trace*. Technically we represent traces as streams: **traces** are *streams* of *actions*[1]. Now we can also treat the notion of a trace specification formally: a **trace specification** is a *predicate* over *traces*, i.e. the behaviour of a distributed system is described by a set of traces, namely the truth set of the predicate of the concerning trace specification.

[1] Therefore, traces in our sense are *finite* or *infinite sequences*.

Writing Trace Specifications: When composing a trace specification one has to follow at least the following lines:

- specify a set of actions, say Act; thence, the set of all possible traces is Act^ω;
- now specify a predicate over traces, say TS by
  rel TS: Act^ω **such that** $\forall s : Act^\omega \left[TS[s] \iff formula \right]$,
  where $formula$, the kernel of the trace specification, is written in first-order predicative style[2].

Actions: Actions are *atomic execution steps*. A general *execution step* is carried out by one of the components of the distributed system and causes a transformation of the system's state. *Atomic* means that in the sequential observer's view an action is an observation unit, because an action is regarded as "uninterruptable" or "undivisible". For example, let $Act := \{a, b\}$, i.e. we have two different actions a and b. The trace set describing the parallel composition a$\|$b of the two actions is simply equal to $\{\langle a, b\rangle, \langle b, a\rangle\}$. Now assume a and b being not atomic program steps, then their parallel composition yields more possibilities of traces, since it is unknown in which atomic step the control changes the active process.

Structuring Actions by Action Generating Functions: $Act := \{a, b\}$ contains unstructured and uninterpreted actions. But, in general, we have often objects the actions refer to. At this, algebraic specification techniques may be used to get more structured action sets. More precisely, into a (new) signature we collect some sort symbols expressing the action set, say Act, the object sets, and other things together with some function symbols of the functionality $Obj\_Sort_1 \times \ldots \times Obj\_Sort_k \rightarrow Act$ representing **action generating functions**. Now, take the *term algebra* belonging to this signature as the desired set of actions.

Auxiliary Concepts for Conveniently Writing Trace Specifications: First, we give the definition of the *actual* predicate, which concerns the consideration of an observed actual situation and which is therefore of good use for specifying safety conditions of all possible trace specifications:

rel $actual$: $Act^\omega \times Act \times Act^\omega$
such that $\forall p{:}Act^\omega,\ a{:}Act,\ s{:}Act^\omega \left[actual[p, a, s] \iff p \in Act^* \wedge p \,\&\, \langle a \rangle \sqsubseteq s \right]$.

Intuitively, $actual[p, a, s]$ holds iff p is a *finite* observed "**p**ast" and a is to be the considered **a**ctual action of trace s.

Now, we want to introduce two auxiliary concepts concerning action generating functions and the action set. Assume that we have introduced an action generating function act by

fct act: $OS_1 \times \ldots \times OS_k \rightarrow Act$
which we abbreviate by **Act fct** act: $OS_1 \times \ldots \times OS_k$.

The word symbol **Act fct** says that:
(1) the sort symbol for the set of all possible actions is called Act;
(2) act has Act for its result type;
(3) act is variously overloaded by the following definition scheme:

fun act **overload**: $OS_1 \times \ldots \times OS_i \rightarrow Pot(Act)$ {**forall** $0 \leq i \leq k$}
such that $\forall x{:}OS_1 \times \ldots \left[act(x) = \{a{:}Act \mid \exists y{:}OS_{i+1} \times \ldots \times OS_k\ [a = act(x, y)]\} \right]$.

[2] The underlying logic is called *trace logic*, because, above all, operations on traces are allowed as special operations.

In addition to this, we extend the domain of ©, and thence also of §, so that it accepts elements of sorts of the form $OS_1 \times \ldots \times OS_i$ belonging to one or more action generating functions, say to

Act fct act_l: $OS_1 \times \ldots \times OS_i \times RestSort_l$,

where l ranges between 1 and a fixed number, on its first argument place. Then, © has the following meaning:

fun © overload: $OS_1 \times \ldots \times OS_i \times Act^\omega \to Act^\omega$

such that $\forall x : OS_1 \times \ldots \times OS_i \left[x \copyright s = \left(\bigcup_l act_l(x) \right) \copyright s \right]$,

and the same for §.

3.1.2 The First Trace Specification of the Connecting Switch: Pure Trace Logic

Now, we develop the first formal treatment of the connecting switch following along the lines of composing trace specifications.

Action Set Specification: *Objects that actions will refer to:* There are two not further specified sorts of objects:

used sort *Stations* /* sort of possible stations */;

used sort *Messages* /* sort of messages to be sent */.

Action generating functions: According to the description, we have three appropriate functions:

Act fct *conn*: *Stations* × *Stations* /* connect to another station */;

Act fct *send*: *Stations* × *Messages* /* send a message to the (unique) partner */;

Act fct *disc*: *Stations* /* give up the (unique) communication link */.

Trace Specification: A formal requirement specification for the connecting switch is now given by the predicate *con_switch* over Act^ω:

rel *con_switch*: Act^ω

such that

$$\forall s : Act^\omega \left[con\_switch[s] \iff \forall t : Stations \left[\begin{array}{ll} \text{(L1)} & conn(t) \,\S\, s = disc(t) \,\S\, s \wedge \\ & \forall p : Act^\omega,\ a : Act \left[actual[p, a, s] \Longrightarrow \right. \\ \text{(S1)} & \Big((a \in conn(t) \quad \Rightarrow \quad conn(t) \,\S\, p = disc(t) \,\S\, p) \wedge \\ \text{(S2)} & \ \ (a \in send(t) \quad \Rightarrow \quad conn(t) \,\S\, p > disc(t) \,\S\, p) \wedge \\ \text{(S3)} & \ \ (a = disc(t) \quad \Rightarrow \quad conn(t) \,\S\, p > disc(t) \,\S\, p) \Big) \Big] . \end{array} \right] \right]$$

Explanation: The *con_switch* predicate basically specifies the following properties:

(L1) A station may only be and is altogether as often disconnected as being connected; this says, whenever a connection has been established, this connection has to be given up once in the future.

And in every intermediate state of the system the following conditions must hold:

(S1) A station may only be allowed to get connected, if it has been as often disconnected as connected.

(S2) A station may only be allowed to send a message, if it has been more often connected than disconnected.

(S3) A station may only be allowed to get disconnected, if, again, it has been more often connected than disconnected.

As one can see, this specification, which is taken directly from [Broy 88, 3.1.2] (with slight changes), is not a really ad-hoc variant of a trace specification of a connecting switch. It is presented here in order to show how a trace specification can be stated in "pure" trace logic. Later variants of trace specifications will show several techniques for obtaining a trace specification more intuitively and more systematically.

3.1.3 Safety and Liveness Conditions at Trace Specifications

As above mentioned, for a specification of a distributed system one can distinguish safety and liveness conditions for the behaviour of the system. In fact, the division of a collection of requirements contained by a system specification according to these two categories is complete in the sense that every specification can be obtained by stating safety and liveness requirements separately and then composing them [Dederichs Weber 90].

For trace specifications the notions of safety and of liveness may be formalized as follows:

- A predicate S over traces of Act^ω is called a **safety predicate** iff
  $$\forall t : Act^\omega \Big[S[t] \iff \forall t' : Act^* \, [\, t' \sqsubseteq t \;\Rightarrow\; S[t'] \,] \Big].$$
- A predicate L over Act^ω is called a **liveness predicate** iff
  $$\forall t' : Act^* \Big[\exists t : Act^\omega \, [\, t' \sqsubseteq t \wedge L[t] \,] \Big].$$
- (*Combination of both aspects:*) Let S be a *safety predicate* over Act^ω. A predicate L_S over Act^ω is called **liveness predicate for** S iff
  $$\forall t' : Act^* \Big[S[t'] \implies \exists t : Act^\omega \, [\, t' \sqsubseteq t \wedge L_S[t] \wedge S[t] \,] \Big].$$

We consider from the first trace specification only the conditions (S1) and (L1). (S1) reads in full detail as follows:

$$\textbf{rel } S1\text{: } Act^\omega \textbf{ such that } \forall s : Act^\omega \Big[S1[s] \iff$$
$$\forall\, t : Stations,\; p : Act^\omega,\; a : Act \left[\begin{array}{l} actual[p, a, s] \wedge a \in conn(t) \\ \implies conn(t) \,\S\, p = disc(t) \,\S\, p \end{array} \right] \;\Big];$$

and is, therefore, obviously recognizable as a safety condition. Whereas (L1) is the following predicate:

$$\textbf{rel } L1\text{: } Act^\omega \textbf{ such that } \forall s : Act^\omega \Big[L1[s] \iff \forall\, t : Stations \, [conn(t) \,\S\, s = disc(t) \,\S\, s] \Big],$$

which may easily be identified as a mere liveness condition: add to an observation the missing *conn* or *disc* actions to complete it. In order to prove that (L1) is a liveness condition wrt. the conjunction of (S1), (S2), and (S3), which is assumed to be represented by a predicate over Act^ω called $safe\_con\_switch$, one has to show that

$$(\star) \quad \forall\, s : Act^\omega,\, t : Stations \Big[safe\_con\_switch[s] \implies conn(t) \S s \not< disc(t) \S s \Big]$$

holds; then, one knows that one has only to add an appropriate sequence of *disc* actions to complete a safe observation to a safe trace.

3.2 State-Oriented Trace Specifications

As we have seen, a trace specification describes the operational behaviour of a distributed system by (sequential) histories of the actions executed by the system. However, an action, and so does any execution step, causes a transformation of one system's state to another. Therefore, it seems to be equivalent to use state-oriented concepts for requirement specifications. As we will show in this subsection, working with states can be used to obtain trace specifications more easily.

3.2.1 Applying Macro Techniques

For the state-oriented concept to be explained now, a state is conceived as a property of traces, because a trace as a sequence of actions is an execution step transforming one of the potential initial states to the current state. Here, we use the realisation of states rather as another auxiliary concept like *actual* and so on.

Thus we introduce the "states" belonging to the connecting switch in a special way: we use appropriate auxiliary predicate definitions as the following:

$$\begin{aligned}&\textbf{rel } state\_one\colon Stations \times Act^{\omega};\\ &\textbf{rel } state\_two\colon Stations \times Act^{\omega}\\ &\quad \textbf{such that } \forall t\colon Stations, p\colon Act^{\omega} \left[\begin{array}{lcl} state\_one[t,\,p] & \Longleftrightarrow & conn(t)\,\S\,p = disc(t)\,\S\,p \wedge \\ state\_two[t,\,p] & \Longleftrightarrow & conn(t)\,\S\,p > disc(t)\,\S\,p \end{array}\right].\end{aligned}$$

The aimed reformulation of the trace specification of *con_switch* is not difficult and is left up to the reader. But we give some remarks on particular circumstances:

1. *Initial state* is *state_one*, because $\forall t\colon Stations\big[state\_one[t,\,\varepsilon]\big]$ holds.
2. *"Final" state* is *state_one*, as it may be read from the liveness condition (L1).
3. *conn* actions may only be performed iff the assumed protocol machine is in *state_one*.
4. *state_two* is reached by *conn* actions and enables the performing of either a *send* or a *disc* action, but it may be left by a *disc* action only.

These remarks may be depicted as shown in FIGURE 1.

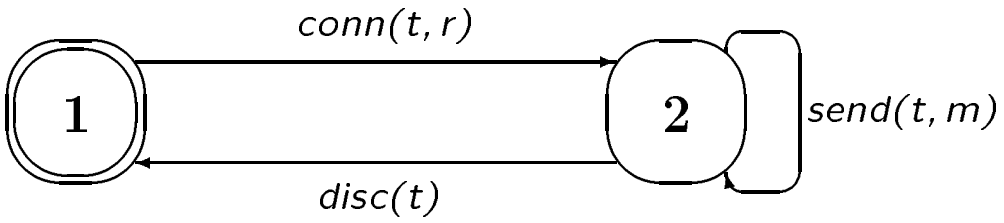

Figure 1: Transition System for each Station t of the Connecting Switch

There are two concluding remarks remaining: (1) we could have chosen some more placative names for the predicates representing the states, but the present selection is done intentionally with respect to the second technique described next and in order to stress the macro character of these predicates; (2) another example how one can write a trace specification of this style is shown in [Dederichs 90], where the well known problem of the dining philosophers is treated.

3.2.2 Using Transition Systems

The foregoing state-oriented concept does not introduce the set of states directly. In contrary to this, we explain now a state-oriented concept which uses such a set of states explicitly: the notion of a *transition system*. Such a transition system gets a trace as its input, and works on it by changing the state for each executed action starting from one of the possible initial states.

More precisely, we introduce a *sort* of states as concrete objects in the following manner: while actions are built up by action generating functions, thence we may speak of being *synthesized*, the set of states is assumed to be sufficiently large and each state can be "read" by predicates having them as parameters, hence we may speak of being *analysed* and of *state reading predicates*.

Now we explain the necessary components of a **transition system**:

1. a sort of actions: **sort** Act;
   together with some action generating functions;
2. a sort of states: **sort** $States$;
   together with some state reading predicates: **rel** $stat$: $\ldots \times States \times \ldots$;
3. the transition relation: **rel** $\longrightarrow$: $States \times \nwarrow Act \nwarrow \times States$;
4. a subset of states containing the initial states, say $Init$;
5. finally, a subset of states containing the liveness states, say $Final$.

As usual, one may introduce the extension $\Longrightarrow$ of $\longrightarrow$ from actions to finite traces:

$$\begin{array}{l}\textbf{rel } \Longrightarrow\colon States \times \nwarrow Act^* \nwarrow \times States \\ \quad \textbf{such that} \\ \quad \forall s\colon Act^*,\, \sigma_0, \sigma_1\colon States \Big[\sigma_0 \overset{s}{\Longrightarrow} \sigma_1 \iff \left(\begin{array}{l}(s = \varepsilon \wedge \sigma_0 = \sigma_1) \vee \\ \exists a\colon Act\,[s = \langle a \rangle \wedge \sigma_0 \xrightarrow{a} \sigma_1] \vee \\ \exists \sigma'\colon States\,[\sigma_0 \overset{\text{ft } s}{\longrightarrow} \sigma' \wedge \sigma' \overset{\text{rt } s}{\Longrightarrow} \sigma_1]\end{array}\right)\Big]\end{array}$$

Once the transition system is completely specified, a trace specification is constructed as a predicate of *accepted* traces:

Case 1: a *finite* trace $s \in Act^*$ is called **accepted** iff
$\exists \sigma_0, \sigma_1\colon States \left[\sigma_0 \in Init \wedge \sigma_0 \overset{s}{\Longrightarrow} \sigma_1 \wedge \sigma_1 \in Final\right]$ holds;

Case 2: a *infinite* trace $s \in Act^\infty$ is called **accepted** iff
$\forall\, s'\colon Act^* \Big[s' \sqsubseteq s \Longrightarrow \exists \tilde{s}\colon Act^*\,[\,s' \sqsubseteq \tilde{s} \wedge \tilde{s} \sqsubseteq s \wedge \tilde{s} \text{ is accepted}\,]\Big]$ holds.

We let Acc_Traces denote the set of all accepted traces of the specified transition system.

Now we want to apply this technique to the example of the connecting witch. What remains to do is specifying the components $States$, $Init$, $\longrightarrow$, and $Final$.

State reading predicate: There is only one state reading predicate. It collects all currently connected stations:

$$\begin{array}{l}\textbf{rel } is\_conn\colon States \times Stations \\ \quad \textbf{such that } \texttt{/* } States \texttt{ is sufficiently large */} \\ \quad \forall\, T\colon Pot(Stations)\Big[\exists \sigma\colon States\,[\forall t\colon Stations\,[\,is\_conn[\sigma, t] \iff t \in T\,]]\Big],\end{array}$$

where $is\_conn[\sigma, t]$ says station t to be connected in state σ.

Initial configuration; Liveness Condition: The following condition must hold in the unique initial state $init$, i.e. $Init = \{init\}$:

$$\textbf{fct } init\colon States \textbf{ such that } \forall t : Stations\Big[\neg is\_conn[init, t]\Big].$$

$init$ is also the only liveness state, i.e. $Final = \{init\}$.

State transition: The transition relation $\longrightarrow$ is specified as follows:

$$\begin{array}{l}\textbf{rel } \longrightarrow\!: States \times \setminus Act \setminus \times States \textbf{ such that} \\ \quad \forall\, a\!:\!Act,\, \sigma_0, \sigma_1\!:\!States \Big[\sigma_0 \xrightarrow{a} \sigma_1 \iff \\ \qquad \exists\, t\!:\!Station\,[a \in conn(t) \wedge \neg is\_conn[\sigma_0, t] \wedge is\_conn[\sigma_1, t]] \vee \\ \qquad \exists\, t\!:\!Station\,[a \in send(t) \wedge is\_conn[\sigma_0, t] \wedge is\_conn[\sigma_1, t]] \vee \\ \qquad \exists\, t\!:\!Station\,[a = disc(t) \wedge is\_conn[\sigma_0, t] \wedge \neg is\_conn[\sigma_1, t]]\Big].\end{array}$$

A kind of transition graph, or its projection to each station, may be found in FIGURE 1.

Trace Specification: As explained above, the trace specification is constructed from Acc_Traces as follows:

$$\begin{array}{l}\textbf{rel } Acc\_con\_switch\!:\, Act^{\omega} \\ \quad \textbf{such that } \forall s : Act^{\omega} \Big[\, Acc\_con\_switch[s] \iff s \in Acc\_Traces \,\Big],\end{array}$$

where $s \in Acc\_Traces$ stands for the specification of the conditions of accepted traces mentioned above: $(\, s \in Act^{*} \;?\; init \stackrel{s}{\Longrightarrow} init \,:\, \forall\, p\!:\!Act^{*}\Big[p \sqsubseteq s \Longrightarrow \exists \tilde{s}\!:\!Act^{*}\,[\, p \sqsubseteq \tilde{s} \wedge \tilde{s} \sqsubseteq s \wedge init \stackrel{\tilde{s}}{\Longrightarrow} init\,]\Big]\,)$.

Comparing the two trace specifications con_switch and Acc_con_switch, as for *correctness* aspects, one wants to prove that, at least, one of them is a refinement of the other. But, actually, they are equivalent, i.e. they have the same truth set.

Theorem 1 *con_switch and Acc_con_switch are equivalent predicates, i.e.* $\forall s : Act^{\omega}\Big[con\_switch[s] \iff Acc\_con\_switch[s]\Big]$ *holds.*

Proof: "$\Longrightarrow$": First we prove the following

Lemma: Assume that $s \in Act^{*}$ and $safe\_con\_switch[s]$ are satisfied. Then

$$\forall t : Stations\,[\, conn(t)\S s = disc(t)\S s \iff \forall \sigma : States\,[\,(init \stackrel{s}{\Longrightarrow} \sigma) \Longrightarrow \neg is\_conn[\sigma, t]\,]\,]$$

holds.

Proof of lemma: Let $t \in Stations$ and $s \in Act^{*}$ be given.

For convenience, we abbreviate the condition of the left hand side, $conn(t)\S s = disc(t)\S s$, by $L1[t, s]$ and the condition of the right hand side, $\forall \sigma : States\,[\,(init \stackrel{s}{\Longrightarrow} \sigma) \Longrightarrow \neg is\_conn[\sigma, t]\,]$, by $\Lambda 1[t, s]$.

The case of $s = \varepsilon$ is clear. Assume that s is non-empty and the claimed condition is satisfied (*induction hypothesis*).

Let $a \in Act$. *Case 1:* Assume $a \in conn(t)$. If $L1[t, s]$ holds, neither $L1[t, s\&\langle a\rangle]$ holds, nor does $\Lambda 1[t, s\&\langle a\rangle]$. But $\neg L1[t, s]$ turns to $\neg con\_switch[s\&\langle a\rangle]$ because of (S1). *Case 2:* If $a \in send(t)$, obviously $L1[t, s\&\langle a\rangle]$ and $\Lambda 1[t, s\&\langle a\rangle]$ both hold (if $L1[t, s]$) or fail (if $\neg L1[t, s]$). *Case 3:* If $a = disc(t)$, $L1[t, s]$ is ruled out by ($\star$) or by (S3). But if $\neg L1[t, s]$, then $\Lambda 1[t, s\&\langle a\rangle]$ holds. $L1[t, s\&\langle a\rangle]$ may be shown by (S1): a $conn(t)$ action, say c, is only allowed for some past $p \in Act^{*}$ iff $L1[t, p]$, but then $\neg L1[t, p\&\langle c\rangle]$ holds, thence, s is only allowed to contain one $conn(t)$ action more than the number of $disc(t)$ actions. $\diamondsuit$

We have also shown by the lemma that:

$$(s \in Act^{*} \wedge con\_switch[s]) \Longrightarrow \forall \sigma : States\,[\,(init \stackrel{s}{\Longrightarrow} \sigma) \Longrightarrow \sigma = init\,].$$

From the obvious property $s \in Act^{*} \wedge safe\_con\_switch[s] \Longrightarrow \exists \tilde{s} : Act^{*}\,[\, s \sqsubseteq \tilde{s} \wedge con\_switch[\tilde{s}]\,]$ we can conclude that it remains to show that $(s \in Act^{*} \wedge safe\_con\_switch[s]) \Longrightarrow \exists \sigma : States\,[\, init \stackrel{s}{\Longrightarrow} \sigma\,]$:

Case 1: $s = \varepsilon$ directly implies $init \stackrel{s}{\Longrightarrow} init$. *Case 2:* Let a be an action, and assume the claimed condition being satisfied for $s \in Act^{*}$ (*induction hypothesis*). Further, assume $safe\_con\_switch[s\&\langle a\rangle]$, and let $t \in Stations$. *Case 2.1:* If $a \in conn(t)$, by (S1) $\neg L1[t, s]$ is ruled out. Let $\sigma \in States$ be such that $init \stackrel{s}{\Longrightarrow} \sigma$ holds; σ exists because of the induction hypothesis. By the lemma and $L1[t, s]$ we get $\neg is\_conn[\sigma, t]$. The specification of $\longrightarrow$ and the assumption of $States$ being sufficiently large now gives us a state σ' such that $\sigma \xrightarrow{a} \sigma'$ (and $is\_conn[\sigma', t]$). It follows that $init \stackrel{s\&\langle a\rangle}{\Longrightarrow} \sigma'$. *Cases 2.2, 2.3:* may analogously be established. $\diamondsuit$"$\Longrightarrow$"

"$\Longleftarrow$": While assuming $s \in Act^{*}$ and also $init \stackrel{s}{\Longrightarrow} init$ instead of $safe\_con\_switch[s]$, we reprove the lemma above.

Reproof of lemma: Let $t \in Stations$. The case of $s = \varepsilon$ is trivial. Assume that $s \neq \varepsilon$ and the claimed condition holds for s (*induction hypothesis*).

Let $a \in Act$. *Case 1:* Assume $a \in conn(t)$. If $L1[t, s]$ holds, then $L1[t, s\&\langle a\rangle]$ and $\Lambda 1[t, s\&\langle a\rangle]$ both fail to hold. But $\neg L1[t, s]$ is tackled by the specification of $\longrightarrow$, because by induction hypothesis this says

$$\exists \sigma : States\,[\,(init \stackrel{s}{\Longrightarrow} \sigma) \wedge is\_conn[\sigma, t]\,],$$

which is equivalent to $\forall \ldots$, for $\longrightarrow$ is deterministically specified, and, thence, implies $\neg\exists\sigma' : States\,[init \stackrel{s\&\langle a\rangle}{\Longrightarrow} \sigma']$.

Now it is clear, how the reproof of the lemma can be completed: instead of using the safety conditions (S1) to (S3) one takes the corresponding subformulas of the specification of $\longrightarrow$. $\diamondsuit$

From the lemma we can derive $Acc\_con\_switch[s] \implies L1[s]$. It remains to deal with the safety parts: $\exists\sigma : States\,[init \stackrel{s}{\Longrightarrow} \sigma] \implies safe\_con\_switch[s]$.

Case 1: If $s = \varepsilon$ holds, $init \stackrel{s}{\Longrightarrow} init$ and $safe\_con\_switch[s]$ both obviously are satisfied. *Case 2:* Let a be any action, and assume that there is a state σ' such that $init \stackrel{s\&\langle a\rangle}{\Longrightarrow} \sigma'$. But then we have also a state σ such that $init \stackrel{s}{\Longrightarrow} \sigma$ and $\sigma \stackrel{a}{\longrightarrow} \sigma'$. Assume that $safe\_con\_switch[s]$ already holds (*induction hypothesis*). *Case 2.1:* Assume $a \in conn(t)$. Then, we have $\neg is\_conn[\sigma, t]$, and by the lemma, which may be (re)proved by assuming $\exists\sigma : States\,[init \stackrel{s}{\Longrightarrow} \sigma]$ instead of $Acc\_con\_switch[s]$, it follows that $L1[t, s]$ holds, which, in turn, implies (S1) being satisfied for $s\&\langle a\rangle$. But $a \in conn(t)$ is not concerned by (S2) and (S3) and, therefore, $s\&\langle a\rangle$ is proved safe. *Cases 2.2, 2.3:* may be shown analogously. $\diamondsuit$"$\Longleftarrow$" □

The proof of our first theorem shows, how safety and liveness are expressed by the two description techniques. Further, it exposes a proof technique which is merely a classical Noetherian induction, because $\sqsubseteq$ well-orders Act^* (and even Act^ω). This technique is used throughout all proofs concerning safety properties. But, in this example, it has also turned out that finding a proposition on some kind of states, like the ones of the macro technique, may lead to an easy proof of liveness: the lemma contains $L1[t, s]$ which is obtainable from $L1[s]$ by omitting the universal quantification over stations. In general, proofs of liveness properties are harder to achieve; they require an analysis of the underlying formulae of the liveness conditions belonging to the concerning trace specification.

3.3 TRANSACT – A Notational Framework for Trace Specifications of Transactional Systems

Now we discuss a technique which deals with a class of distributed systems that may be characterised by the property of repeatedly carrying out certain sequences of actions called **transactions** and that we therefore call **transactional**. This class of distributed systems comprises the behaviours of all protocols, because protocols describe such appropriate sequences of actions, which have to be executed by the participating communication partners.

The technique to be explained offers a notational framework for formulating special predicates over traces called $TRANSACT$. $TRANSACT$ allows to specify transactions as sequences of actions by some kind of *regular* composed language over the set of actions as the alphabet.

A special aspect of the composition of a sequence of actions rests on the fact that each transaction may have one unique owner. In the example of the connecting switch, we have the simple case of any action belonging to the station found in its (first) argument place. Therefore, the complete "transaction" of connecting, then sending, and finally disconnecting has one station as its owner. Such a special situation is called the **owner principle**.

A $TRANSACT$ specification takes this principle also into account; it consists, therefore, of an owner part expressed by the sort of potential owners and of the transaction part specifying the sequence of actions to be carried out. Now we explain this in detail:

`[` *owner* `::` *transaction* `]` is the usual $TRANSACT$ statement; it specifies the requirement of following the sequence of actions *transaction*, which is owned by some element of the sort *owner*.

owner is, as we already mentioned, the sort of potential owners, rather its name.

transaction specifies a sequence of actions; it is composed as follows:

- atom: *action*;
- sequential composition: *transaction*.*transaction*;
- nondeterministic choice: *transaction*/*transaction*;

– "Kleenian star": $*transaction*$;

– bracketing: $(transaction)$.

$action$ is of the form $actionname\left[\text{-}arg2\left[\text{-}arg3\ldots\right]\right]$, where $arg2$ etc. are names of sorts.

We only have given the general syntactical concerns of $TRANSACT$; the semantical aspects will be considered when dealing with our example.

Now, the intended $TRANSACT$ specification of the connecting switch simply reads as follows:

rel con_switch_∞: Act^ω

such that `[ Stations :: conn-Stations.*send-Messages*.disc ]`,

which is equivalent to the following (extended) trace logic specification:

$$\ldots\textbf{such that } \forall s : Act^\omega \Big[\, con\_switch_\infty[s] \iff \forall t : Stations \left[\exists \left\{ \begin{array}{l} k : \overline{Nat}, \\ r : \overline{Nat} \to Stations, \\ m : \overline{Nat} \to Messages, \\ n : \overline{Nat} \to Nat \end{array} \right\} \left[\begin{array}{l} t \copyright s = \\ \quad \&_{i=1}^{k} [\, \langle conn(t, r_i) \rangle \& \\ \qquad \langle send(t, m_i) \rangle^{n_i} \& \\ \qquad \langle disc(t) \rangle \,] \end{array} \right] \right] ,$$

where $t \copyright s$ is one of the auxiliary concepts, $\&_{i=1}^{k}$ has the usual meaning, in particular, for $k = \infty$ it leads a least upper bound expression, and $\langle send(t, m_i) \rangle^{n_i}$ is the n_i-length sequence consisting of $send(t, m_i)$ actions only.

As for the transition system specification, we want to compare the TRANSACT specification with the pure trace logic specification con_switch: it turns out that con_switch is equivalent to con_switch_∞.

Theorem 2 *con_switch_∞ is equivalent to con_switch, i.e.*
$\forall s : Act^\omega \Big[con\_switch_\infty[s] \iff con\_switch[s] \Big]$ holds.

Proof: "$\Longrightarrow$": Let $s \in Act^\omega$ be such that $con\_switch_\infty[s]$. $L1[s]$ is obvious, as is $L1[t, s]$ (see proof of Th.1) for each station t.

For convenience, we abbreviate the condition on the right hand side in the trace logic specification of con_switch_∞ by $con\_switch_\infty[t, s]$ for $t \in Stations, s \in Act^\omega$.

Fix a station t. *(S1):* Let $p \in Act^*$ and $a \in conn(t)$ be such that $p\&a \sqsubseteq s$, i.e. $actual[p, a, s]$ holds. From the special form of s we may conclude that $con\_switch_\infty[t, p]$ because $conn(t)$ actions follow only and exclusively $disc(t)$ actions in $t \copyright s$. But then $L1[t, p]$ holds and (S1) is satisfied. *(S2):* Let $p \in Act^*$ and $a \in send(t)$ be such that $p\&a \sqsubseteq s$. Then, we get a $p' \in Act^*$ such that $p' \sqsubseteq p$, $con\_switch_\infty[t, p']$, and $p = p'\&(\langle conn(t, r) \rangle \& \langle send(t, m) \rangle^n)$ for some appropriate r, m, n. Thence, we have

$$conn(t)\S p - disc(t)\S p = 1 > 0$$

and (S2) is satisfied. *(S3):* may be shown analogously to (S2). $\diamond$"$\Longrightarrow$"

"$\Longleftarrow$": Let t be any station. Let $s \in Act^\omega$ be such that $safe\_con\_switch[s]$ and $L1[t, s]$ (see proof of Th.1) are satisfied. The case of $s = \varepsilon$ is clear, thus assume that $s \neq \varepsilon$. W.l.o.g. assume $s = t \copyright s$. Let p, a be such that $actual[p, a, s]$ and $con\_switch_\infty[t, p]$ are satisfied. *(induction hypothesis)*

We show that there exists $\tilde{p}$ such that $p\&\langle a \rangle \sqsubseteq \tilde{p}$, $\tilde{p} \sqsubseteq s$, and $con\_switch_\infty[t, \tilde{p}]$.

By $con\_switch_\infty[t, p]$ it follows that $L1[t, p]$. But, by (S2), (S3), and $L1[t, p]$ we immediately get $a \in conn(t)$. Furthermore, by $L1[t, p]$ and $L1[t, s]$ we get some b such that $actual[p\&\langle a \rangle, b, s]$. But by (S1) $b \notin conn(t)$ holds. If $b = disc(t)$ we immediately have $\tilde{p} := p\&\langle a, b \rangle$.

But if $b \in send(t)$, there exists some c such that $p\&\langle a, b \rangle \& c \sqsubseteq s$, $conn(t) \copyright c = \varepsilon$, and lt $c = disc(t)$ by $L1[t, s]$. By (S1) and (S3) ld c must be a (maybe empty) sequence of $send$ actions. But, then, put $\tilde{p} := p\&\langle a, b \rangle \& c$.

The last claim shows $con\_switch_\infty[t, s]$, and we are done. $\diamond$"$\Longleftarrow$" □

We have shown, how liveness and safety conditions may be composed within a compact notation by $TRANSACT$: **safety:** any admissible trace must follow the prescribed transaction when restricted to each station, **liveness:** any begun transaction must be completed.

$TRANSACT$ specifications may not only be used to abbreviate the specifying formula as described above, but also to define predicates over traces itself. This allows to use $TRANSACT$ statements freely in a trace logic specification in order to combine transaction requirements with trace logic constraints. Taking these remarks into account, for our example, the following specification is obtained, where the $TRANSACT$ specification appears now as a separate predicate:

$$\textbf{rel}\ con\_switch_\infty\colon Act^\omega$$
$$\textbf{such that}\ \forall s : Act^\omega \Big[\, con\_switch_\infty[s] \iff$$
$$\texttt{[Stations :: conn-Stations.*send-Messages*.disc]}[s] \Big].$$

3.4 Modifiability of Trace Specifications

In this subsection we want to explain some techniques or aspects that concern modifications of an existing trace specification to get another more refined or supplemented trace specification; we only want to consider modifications aiming at the set of all possible actions. It is also explained how the proof obligation of verifying the resulted trace specification against the older one may be solved.

3.4.1 Action Refinement

Action refinement says that the actions of the old specification are to be more detailed. In a development process it is convenient to give a trace specification based on a set of abstract actions first and, then, to use appliances to get a correct trace specification containing lesser abstract actions.

An action refinement is characterised by a (one-one) mapping from the old action set into some execution units consisting of elements of the new action set. Actually, this mapping is in the simplest case from Act_{old} into $Pot(Act^*_{new})$, or, more elaborately, if some reference to the past is needed, from $Act^*_{old} \times Act_{old}$ into $Pot(Act^*_{new})$.

Technically, we may proceed as follows: let ρ be an action refinement, i.e. a mapping $Act^*_{old} \times Act_{old} \to Pot(Act^*_{new})$ respectively; we extend ρ to a mapping $\bar{\rho}$ by

$$\textbf{fun}\ \bar{\rho}\colon Act^\omega_{old} \to Pot(Act^\omega_{new})$$
$$\textbf{such that}$$
$$\bar{\rho}(\varepsilon) = \{\varepsilon\}\ \wedge$$
$$\forall s\colon Act^\omega_{old} \Big[\, s \neq \varepsilon \Longrightarrow \forall s' : Act^\omega_{new} \Big[\, s' \in \bar{\rho}(s) \iff$$
$$\forall p : Act^*_{old}, a : Act_{old}, q : Act^\omega_{old}\, [\, s = p\&\langle a\rangle\&q \iff$$
$$\exists p', a', q' : Act^\omega_{new}\, [\, p' \in \bar{\rho}(p) \wedge a' \in \rho(p,a)\ \wedge q' \in \bar{\rho}(q)\ \wedge s' = p'\&a'\&q'\,]\,]\Big]\Big]\ ;$$

let $T_{old} \subseteq Act^\omega_{old}$ be the trace set of the old specification; then, we are able to form $T_{new} \subseteq Act^\omega_{new}$, the trace set of the new specification, by putting

$$T_{new} := \{s' : Act^\omega_{new} \mid \exists s : Act^\omega_{old} \Big[\, s \in T_{old} \wedge s' \in \bar{\rho}(s) \Big]\}.$$

Thus, an action refinement in our sense is fully described by the specification of the components of the following quintuple:

$$(Act_{old}, T_{old}, Act_{new}, \rho).$$

The first two components are part of the presupposed trace specification; Act_{new} is specified with the same method as for Act_{old} (term algebra over some set of action generating functions).

Here, we want to give now a notational framework for specifying the proper component ρ of an action refinement:

*pred*1 **refines** *pred*2 **by** $refinement$ says that the predicate *pred*1 over Act^{ω}_{new} is obtained from the predicate *pred*2 over Act^{ω}_{old} by an action refinement specified by $refinement$. Thus, *pred*1 is a reference to Act_{new} and $refinement$ concerns ρ.

$refinement$ ::= '[{' $refmapentry\left\{\text{','}\ refmapentry\right\}^{*}$ '}]' specifies ρ; each string $refmapentry$ belongs to the specification of ρ.

$refmapentry$ ::= $old\_act\_term \mapsto block$ says how to refine an action of the form old_act_term into execution sequences specified by $block$.

old_act_term is simply a term over Act_{old}. It may contain free variables together with type information (may be omitted if the context is very clear), which, in turn, may used in $block$, the right side of a $refmapentry$ specification.

$block$ ::= [$statement$] determines a set of finite sequences over Act_{new}, which is needed for the specification of any image of any action under ρ.

$statement$ is the kernel of our notational framework; it allows to build a set of (finite) execution sequences by the aid of programming-language-like combinators; there are the following possibilities:

- atom: new_act_term, a term over Act_{new};
- empty command: **skip**;
- sequential composition: $statement$ ';' $statement$;
- conditional choice: **if** $condition$ **then** $statement$ **else** $statement$ **fi**,
  where $condition$ is a (trace logic) formula, which contains references to the past by the word symbol **past**;
- binary nondeterministic choice: $statement$ **or** $statement$;
- arbitrary nondeterministic choice: **select** new_act_term **where** $condition$;
- very special implicit conditional choice: new_act_term **if needed**, see later.

We enrich now our notational framework by adding some object-based concepts. Now we assume for trace specifications that all objects, which actions refer to, have a state of existence: an action may only occur in a trace, if all objects found as its arguments are in the state "existent", otherwise these objects have to be *created*. But, if all action generating functions concerning some sort are defined in the common way, we assume that the system creates all objects of this sort *before* executing any action; namely, in order to bring the object-based concepts into the foreground we use two new word symbols **creat** and **delet** in the place of **Act fct**:

creat Act_fct_name: $sort\_name$
/* Act_fct_name is a create action for objects of $sort\_name$ */;

delet Act_fct_name: $sort\_name$
/* Act_fct_name is a delete action for objects of $sort\_name$ */.

Whenever **creat** and **delet** is used, each trace specification implicitly contains safety requirements saying that any action referring to any sort concerned by any **creat** or **delet** line may only occur iff the corresponding objects are created by a create action before, but not deleted by a delete action.

Now, we are able to apply the concept of an action refinement to the connecting switch. The example specification reads as follows:

/* Act_{new} (here: Act') */
sort $Links = Stations \times Stations$;
creat $create$: $Stations \rightarrow Act'$;
delete del: $Stations \rightarrow Act'$;
creat $link$: $Links \rightarrow Act'$;
delete $unlink$: $Links \rightarrow Act'$;
Act fct $send$ **overload**: $Stations \times Messages \rightarrow Act'$
/* $Messages$ is a sort carrying always existent objects! */;
/* Refinement */
rel $refd\_con\_switch$: Act'^{ω} **such that**
$refd\_con\_switch$ **refines** con_switch **by**

$$\left[\left\{\begin{array}{l} conn(t,r) \mapsto [create(t)\ \textbf{if needed}; create(r)\ \textbf{if needed}; link(t,r)], \\ send(t,m) \mapsto [send(t,m)], \\ disc(t) \mapsto [\textbf{select}\ unlink(t,r) \\ \quad \textbf{where}\ (\ conn(t)\textcircled{c}\textbf{past} \neq \varepsilon \\ \quad\quad ?\ \mathrm{lt}\ (conn(t)\textcircled{c}\textbf{past}) = conn(t,r) \\ \quad\quad :\ r \in Stations \setminus \{t\}\ ) \\ \quad ; \\ \quad [\textbf{skip or}\ del(t)]] \end{array}\right\}\right] .$$

From this specification, one immediately reads that

$$\begin{array}{rcl} \rho(p, send(t,m)) & = & \{\langle send(t,m)\rangle\} \\ \rho(p, disc(t)) & = & (\ conn(t)\textcircled{c}p \neq \varepsilon \\ & & \quad ?\ \{\langle unlink(t,r)\rangle, \langle unlink(t,r), del(t)\rangle | \mathrm{lt}(conn(t)\textcircled{c}p) = conn(t,r)\} \\ & & \quad :\ \{\langle unlink(t,r)\rangle, \langle unlink(t,r), del(t)\rangle | r \in Stations \setminus \{t\}\}\ ) \end{array}$$

As an exception, $\rho(p, conn(t,r))$ is not so simple reconstructable; the **if needed** construct adds a safety requirement to the new system specification that refers not to the past of the *conn* action, but to the past of the translation itself: e.g. "$create(t)$ **if needed**" says that, iff the past of the translation does not support the state of existence for t, we have to put $create(t)$ onto its corresponding place, otherwise the construct stands for **skip**.

Now, the proof obligation of verifying $refd\_con\_switch$ against con_switch does not become visible, because it appears as a mere validation problem of the refinement itself. In the general case, one goes the other way round: let two trace logic specifications be given; so as to verify one specification against the other, there is the hard possibility of searching for a "suitable" refinement mapping ρ; remember ρ not only influencing the action sets but also the safety and liveness conditions.

3.4.2 Action Enrichment

Another modification option for the action set is the addition of actions. This immediately entails the addition of suitable conditions to the old specifications. On the one hand, the extension of the action set is a good test for formal description techniques about the flexibility of the underlying notational framework.

On the other hand, one gets the problem of which conditions being in some sense suitable for the new actions in connection with the older ones. As the technique of **action enrichment** is related to hierarchization of algebraic data types, we use the term **persistency** in the same situation: let two trace predicates p_1 and p_2 be given together with their action sets Act_1 and Act_2, where $Act_1 \subseteq Act_2$; p_2 is called **persistent** wrt. p_1 iff (a) for every trace t_2 of p_2 $Act_1 \copyright t_2$ is a trace of p_1 and (b) for every trace t_1 of p_1 there is a trace t_2 of p_2 such that $t_1 = Act_1 \copyright t_2$.

In our example, we want to extend the action set, which contains connecting, sending, and disconnecting actions so far, by a receive action. But, for simplicity, the receive action is not owned by the receiver, but by the sender. Moreover, we want to let any receiving action only occur as a consequence of a foregoing sending action (*safety*), but also vice versa, i.e. we expect for each sending action a corresponding receiving action (*liveness*). More detailed, the enriched specification reads as follows:

sort Act_r **extends** Act;
Act fct $recv$: $Stations \rightarrow Act_r$;
rel con_switch_r: Act_r^ω **such that**
con_switch_r **extends** con_switch **by**

$$s : Act_r^\omega \left[\forall t : Stations \left[\begin{array}{l} (\mathrm{L2}) \;\; send(t) \S s = recv(t) \S s \wedge \\ \forall p : Act_r^\omega,\, a : Act_r \left[actual[p, a, s] \implies \right. \\ (\mathrm{S4}) \quad (a = recv(t) \;\Rightarrow\; send(t) \S p > recv(t) \S p) \left. \right] \end{array} \right] \right],$$

where we have renounced to introduce the required extensions of our notational framework formally.

Now, the proof of persistency appears as the corresponding proof obligation with respect to verification:

Theorem 3 *con_switch_r is persistent wrt. con_switch.*

Proof: Let $s \in Act_r^\omega$. *Ad (a):* Assume $con\_switch_r[s]$, and put $s_0 := Act \copyright s$. $L1[s]$ holds, therefore, also $L1[s_0]$, because $conn(t) \copyright s = conn(t) \copyright s_0$ and $disc(t) \copyright s = disc(t) \copyright s_0$. For the same reason, from $safe\_con\_switch[s]$ we may conclude that $safe\_con\_switch[s_0]$ holds. Thus, $con\_switch[s_0]$ is satisfied.

Ad (b): Assume $con\_switch[s]$. We construct $s_1 \in Act_r^\omega$ from s as follows:

$s_1 = \varrho(s)$ **where**
$\varrho(\varepsilon) = \varepsilon \wedge \varrho(\langle a \rangle \& \hat{s}) = (\, a \in send \;?\; (\langle a, recv(t) \rangle$ **where** $a \in send(t)) : \langle a \rangle \,) \& \varrho(\hat{s})$.

$s = Act \copyright s_1$ is obvious; $L2[s_1]$ clearly holds by the definition of ϱ; by a simple induction with reference to ϱ one also shows $S4[s_1]$: if $actual[p_1, recv(t), s_1]$ holds, so does $\exists a : Act_r\, [\, a \in send(t) \wedge p_1 = p_1' \& \langle a \rangle\,]$ such that $send(t) \S p_1' = recv(t) \S p_1'$. Thence, we have shown $con\_switch_r[s_1]$. □

This kind of proof obligation and the notion of persistency, respectively, are important to the design method, because a persistent action enrichment allows to proceed with the development steps for the old specification independently forward, because the causality connections within the specified traces are not abandoned by the enrichment.

3.4.3 REACTION – Towards a More Natural Modelling

The given pure trace logic specification *con_switch* of the connecting switch is not the natural derivation from the informal requirements, as we have already admitted. As shown, one way is to develop a *TRANSACT* specification, possibly combined with trace logic; another is to specify a transition system.

But to make the view of a connecting switch more transparent, one goes back to the notion of object in the fundamental version: (a) objects have a state each, its *knowledge*, and (b) objects interact with another objects by *sending messages*. For the latter point, we may reorganize *Act* such that there is one uniform action generating function called *message* containing sender, receiver, and the message content.

This reorganization may also be used as a way to develop a design specification rather than a requirement specification. Once, all actions are transformed into the uniform representation by message actions, one has selected the causality structure of the system: objects may turn into agents.

In order to reflect the division of a distributed system into "system" and "environment" a little bit, we introduce a new object *environment*, to which the overall control of the system is assigned.

An important point of this view is the direct usage of the messages of *Messages* within the communications, i.e. the proper messages are sent by a station (within a send command) and are delivered via *environment* to the target station.

REACTION is a notational framework for descriptions of the interaction of the objects of a system. We do not go into detail, while the specification of our example with REACTION is listed in FIGURE 2.

From the REACTION specification, one obtains a trace specification by the following steps:
(a) add the object *environment* by introducing a new sort *Envs* and a constant definition, form **sort** $Comps = Stations + Envs$.
(b) reorganize *Messages* by *adding* messages of the form
$hello(t)$, $ok(t, \hat{m})$, $says\_hello(t)$, $error(t, \hat{m})$, $send(m)$, bye, and $says\_bye(t)$
where $\hat{m} \in \{hello(t), send(m), bye \mid t, m\}$, each m is an old message.
(c) reorganize *Act* by using **Act fct** *message*: $Comps \times Messages \times Comps$ alone.
(d) use the transition system technique: **send**'s are executed actions, and **knows**'s determine state transitions; the **final** and **initial** blocks have their usual meaning.

The model behind REACTION leads to a more realistic specification of a connecting switch also for another reason: within traces we also allow erroneous situations, which end up with *environment* sending an *error* message to the ordering station. This is because every **block** is intended to be executed by nondeterministic choice. But, in turn, this even requires an appropriate error handling.

Once, the trace specification, say *con_switch_reaction*, is derived from the REACTION specification, we are able to show by identifying the *ok* messages with the corresponding actions of the old set that the result is a *persistent* action enrichment, but we omit the proof here.

system CON_SWITCH_REACTION:
objects *environment*(ENVIRONMENT), r, t(STATION), m(MESSAGE);
global
environment **knows** *said_hello*(STATION), *said_hello*(STATION, STATION)
endglobal;
initial
environment **knows** $\forall t[\neg$*said_hello*$(t)] \wedge \forall t, r[\neg$*said_hello*$(t, r)]$
endinitial;
final
environment **knows** $\forall t[\neg$*said_hello*$(t)] \wedge \forall t, r[\neg$*said_hello*$(t, r)]$
endfinal;
block *stations may get connected* :
$\forall t, r[t$ **sends** *hello*(r) **to** *environment*;
if $\neg$*said_hello*(t)
then
environment **sends** *ok*$(t,$ *hello*$(r))$ **to** t,
sends *says_hello*(t) **to** r,
knows *said_hello*$(t) \wedge$ *said_hello*(t, r)
else *environment* **sends** *error*$(t,$ *hello*$(r))$ **to** t **endif**]
endblock;
block *may then send messages* :
$\forall t, m[t$ **sends** *send*(m) **to** *environment*;
if *said_hello*(t)
then
environment **sends** *ok*$(t,$ *send*$(m))$ **to** t,
sends m **to** r **where** *said_hello*(t, r) /* here, m is delivered directly */
else *environment* **sends** *error*$(t,$ *send*$(m))$ **to** t **endif**]
endblock;
block *and, finally, may get disconnected* :
$\forall t[t$ **sends** *bye* **to** *environment*;
if *said_hello*(t)
then
environment **sends** *ok*$(t,$ *bye*$)$ **to** t,
sends *says_bye*(t) **to** r **where** *said_hello*(t, r)
knows $\neg$*said_hello*$(t) \wedge \neg$*said_hello*(t, r) **where** *said_hello*(t, r)
else *environment* **sends** *error*$(t,$ *bye*$)$ **to** t **endif**]
endblock
endsystem

Figure 2: A REACTION Specification for the Connecting Switch

4. From Trace Specifications to Design Specifications

Trace specifications are used for describing the behaviour of a distributed system on an abstract level. Towards an implementation, one does not consider the system in the whole, but one begins to distinguish some *components*. Basically, components themselves may be regarded as systems, to which a trace specification may be assigned. Technically, one partitions the set of actions and formulate requirements on traces appropriately restricted, as we will show in 4.1. Actually, a component is an entity that has its own causality structure of its own executed actions.

Besides the potentially arbitrary divisibility into components, one may follow the following principle: one divides the distributed system into the "system" component and the "environment". These two components interact within a closed circulation, as depicted in FIGURE 3. How this interaction

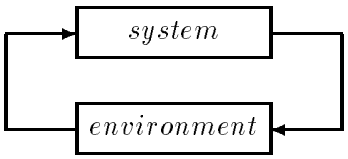

Figure 3: "System" and "Environment"

may be formally described, more with respect to a design specification, is shown in 4.2. But, in addition, this separation method requires the distinction of the input actions and the output actions of the "system" part in order to get a causality structure that may be later interpreted by mapping, as for e.g. by a stream processing function.

Finally, after we have prepared the technical appliances, in 4.3 we explain how the division into "system" part and "environment" part is made for a connecting switch. The described division will be the one that is taken into account for the aimed development process from the level of requirement specification down to an implementation.

4.1 Component-Oriented Specifications

A component-oriented specification deals with the formal division of a trace specification of a distributed system in the whole into single trace specifications of the aimed set of components.

We do not go more into detail and refer to [Broy et al. 91] for a better introduction to this method, but, at least, we describe here the fundamental setting applicable to simple cases: *Step 1:* divide Act into appropriate parts Act_1, Act_2, ...corresponding to the set of components (note that we do not require that the decomposition is obliged to generate disjoint sets of actions); *Step 2:* specify each component by a trace predicate yielding C_1, C_2, ...; *Step 3:* this is the verification step, prove that

$$\forall s : Act\Big[C_1[Act_1 ©s] \wedge C_2[Act_2 ©s] \wedge \ldots \implies C[s]\Big],$$

where C is the trace specification belonging to the whole system.

Now, we discuss two intuitive possibilities of decomposing the connecting switch. The first will go by (bidirected) links, the other by stations. But, to achieve this we have to assume $Stations$ to be finite, otherwise we obtain an infinite verification claim because the conjunction becomes infinite. This shows a drawback of the method, which, but, does not play a great rôle to practice.

Decomposition by Links: The connecting switch is regarded as a family of bidirected links $con\_switch_{\{t,r\}}$:

begin $\{$**forall** $t, r \in Stations \wedge t \neq r\}$
sort $Act_{\{t,r\}}$ **such that** $Act_{\{t,r\}} = \{a{:}Act \mid a \in \left\{\begin{matrix} conn(t,r), send(t), disc(t), \\ conn(r,t), send(r), disc(r) \end{matrix}\right\}$;
rel $con\_switch_{\{t,r\}}$: $Act^{\omega}_{\{t,r\}}$ **such that**
$\forall s{:}Act^{\omega}_{\{t,r\}} \left[con\_switch_{\{t,r\}}[s] \iff con\_switch[Act_{\{t,r\}} \copyright s] \right]$
end.

The proof obligation is immediately satisfied, the conjunction even is equivalent to the specification of the whole connecting switch itself.

Decomposition by Stations: Most naturally, one decomposes the connecting switch into the (*finite*) set of stations. The resulting specifications are determined by the following scheme:

begin $\{$**forall** $t \in Stations\}$
sort Act_t **such that** $Act_t = \{conn(t,r){:}Act \mid r \in Stations\} \cup \{send(t), disc(t)\}$;
rel con_switch_t: Act^{ω}_t **such that**
$\forall s{:}Act^{\omega}_t \left[con\_switch_t[s] \iff con\_switch[Act_t \copyright s] \right]$
end.

It is the same case like at the decomposition by links. Only for convenience, we have used *con_switch* itself and filtering © in the specifications of the components. Moreover, *con_switch* is clearly a *persistent* action enrichment wrt. con_switch_t for each single station t.

4.2 Differential Strategies

The decomposition of a distributed system into "system" and "environment" parts is of good use for developing a design specification with functional agents. "System" and "environment" interact like depicted in FIGURE 3. This view leads to a splitting of the action set in either two parts: the system's input actions In and the system's output actions Out such that $In \cup Out = Act$ and $In \cap Out = \emptyset$.

"System" and "environment" may be viewed as *game players* when they cooperate; both may apply so-called *strategies* [Broy Dederichs Dendorfer Weber 91, §2.2] to perform their interaction. But, in the case of transactional systems, the game view can be described by a *differentially* strategic system, whose kernel is intended to formalize reactions by mapping changes of the input trace to changes of the output trace for each of the players.

More formally, a **differentially strategic system** is a quadruple $\partial\Sigma = (\mathcal{I}, \partial\mathcal{C}, \partial\mathcal{E}, \mathcal{F})$ of sets such that:

$\mathcal{I} \subseteq In^*$ is the set of **initial strategies** to be applied by the environment, which acts always as the first player.

$\partial\mathcal{C} \subseteq In \to Out^*$, $\partial\mathcal{E} \subseteq Out \to In^*$ are the sets of **differential strategies** to be applied by the game players: $\partial\mathcal{C}$ concerns "system", while $\partial\mathcal{E}$ concerns "environment".

$\mathcal{F} \subseteq Out^*$ is the set of **final situations** concerning the output of the "system" component: a trace may be finished up, if the output "last" made by "system" is element of $\mathcal{F}$.

Once, a differentially strategic system is defined, one may introduce the set of traces generated by it in order to install a device for comparing differentially strategic specifications to trace specifications (cf. similar definitions in [Broy Dederichs Dendorfer Weber 91, §2.2]):

$$
\begin{aligned}
&traces(\partial\Sigma) := traces_\varepsilon(\partial\Sigma) \cup traces_+(\partial\Sigma) \cup traces_\infty(\partial\Sigma);\\
&traces_\infty(\partial\Sigma) :=\\
&\quad \left\{ \bigsqcup\{t_n : Act^\omega \mid n \in Nat\} \;\middle|\; \forall n : Nat \left[t_n \sqsubseteq t_{n+1} \wedge t_n \in ptraces_\infty(\partial\Sigma, n) \right] \right\};\\
&ptraces_\infty(\partial\Sigma, 0) := \mathcal{I} \setminus \{\varepsilon\};\\
&ptraces_\infty(\partial\Sigma, 2n) :=\\
&\quad \{s \& e \mid s \in ptraces_\infty(\partial\Sigma, 2n-1) \wedge \exists \partial E : \partial\mathcal{E} \, [\, e = \partial E(\mathrm{lt}\ (Out \copyright s))\,]\}\\
&ptraces_\infty(\partial\Sigma, 2n+1) :=\\
&\quad \{s \& c \mid s \in ptraces_\infty(\partial\Sigma, 2n) \wedge \exists \partial C : \partial\mathcal{C} \, [\, c = \partial C(\mathrm{lt}\ (In \copyright s))\,]\}\\
&traces_+(\partial\Sigma) := \bigcup\{ptraces_+(\partial\Sigma, n) \mid n \in Nat\};\\
&ptraces_+(\partial\Sigma, n) :=\\
&\quad \{s \& c \mid s \in ptraces_\infty(\partial\Sigma, 2n) \wedge \exists \partial C : \partial\mathcal{C} \, [\, c = \partial C(\mathrm{lt}\ (In \copyright s))\,] \wedge c \in \mathcal{F}\};\\
&traces_\varepsilon(\partial\Sigma) := (\ \varepsilon \in \mathcal{I}\ ?\ \{\varepsilon\}\ :\ \emptyset\ )
\end{aligned}
$$

Now, the differentially strategic specification of the connecting switch is a differentially strategic system $\partial\Sigma_{con\_switch}$ such that:

$$
\begin{aligned}
&In := conn;\ Out := send \cup disc;\\
&\mathcal{I} := \{\langle a \rangle \mid a \in In\} \cup \{\varepsilon\};\\
&\partial\mathcal{C} := \{\partial C\colon In \to Out^* \mid \exists n : Nat,\ m : Messages \, [\, \partial C = \partial C_{n,m}\,]\};\\
&\partial C_{n,m}(conn(t,r)) := \langle send(t,m)\rangle^n \& \langle disc(t)\rangle;\\
&\partial\mathcal{E} := \{\partial E\colon Out \to In^* \mid \partial E(disc(t)) \in conn\};\\
&\mathcal{F} := \{\langle send(t,m)\rangle^n \& \langle disc(t)\rangle \mid m, n, r, t\}.
\end{aligned}
$$

Theorem 4 *$\partial\Sigma_{con\_switch}$ satisfies con_switch, i.e.*

$$\forall s : Act^\omega \left[s \in traces(\partial\Sigma_{con\_switch}) \Longrightarrow con\_switch[s] \right]$$

holds.

Proof: For convenience, we write $\partial\Sigma$ for $\partial\Sigma_{con\_switch}$.

(1) $ptraces_\infty(\partial\Sigma, n)$ contains safe traces only for all $n \in Nat$: *$n = 0$:* $ptraces_\infty(\partial\Sigma, 0)$ contains only traces with a single *conn* action. *$n > 0$:* Assume $ptraces_\infty(\partial\Sigma, n)$ containing only safe traces (*induction hypothesis*). *Case 1:* $n = 2k$: $ptraces_\infty(\partial\Sigma, n+1)$ contains all traces $s\&c$ such that $s \in ptraces_\infty(\partial\Sigma, n)$ and there is a differential strategy $\partial C \in \partial\mathcal{C}$ such that $c = \partial C(conn(t,r))$ where $t, r \in Stations$ are appropriate. We get a $n \in Nat$ and a $m \in Messages$ such that $c = \langle send(t,m)\rangle^n \& \langle disc(t)\rangle$. Thence, each $s\&c$ is again safe. Moreover, each $s\&c$ is also live. *Case 2:* $n = 2k-1$: may be shown analogously: the differential strategy sees only the *disc* action last executed. $\diamond$"(1)"

(2) $ptraces_+(\partial\Sigma, n)$ satisfies *con_switch* for all $n \in Nat$: within the proof of (1) we have already shown in case 1 of the induction conclusion that $ptraces_\infty(\partial\Sigma, 2n+1)$ contains only safe and live traces, but by the definition of $\mathcal{F}$ the latter set is equal to the former; this proves (2). $\diamond$"(2)"

(3) $traces_+(\partial\Sigma_{con\_switch})$ satisfies *con_switch*: immediately by (2). $\diamond$"(3)"

(4) the traces in $traces_\infty(\partial\Sigma)$ satisfy *con_switch*: each one is also a least upper bound of a prefix-monotonic trace sequence (t_n) such that $t_n \in ptraces_\infty(\partial\Sigma, 2n+1)$, but every such trace is safe and also satisfies $\forall t : Stations \, [\, conn(t) \S t_n = disc(t) \S t_n\,]$. $\diamond$"(4)"

Because $con\_switch[\varepsilon]$ is true, we are done. □

What we have not shown in the last theorem is apparently clear: the differentially strategic specification catches only the normal forms of the traces performed by the connecting switch with respect to the partition of Act into In and Out. It can be shown, as done in 5.2, that a differentially strategic system may be used in order to obtain a functional specification by ordinary continuous stream processing functions.

To make the scope of this technique larger, one may attach states to the differential strategies, e.g. $\partial\mathcal{C} \subseteq States \times In \to States \times Out^*$. In the end, this specifies nothing else than a step relation of the form $\sigma \xrightarrow{i|o} \sigma'$ where σ, σ' are states, $i \in In$ is the seen input, and $o \in Out^*$ is the performed reaction. Then, such a system of "state-oriented differential strategies" would lead to a corresponding state-oriented functional specification, which is not discussed in this work.

4.3 "System" and "Environment" Parts of a Connecting Switch

In 4.1 we have discussed some examples of distributions of the components that does not have any overlapping actions. But, if one considers the "immediate" division of a distributed system into "system" and "environment" components, it turns out that all actions are shared by both of them. The reason for this is already displayed in FIGURE 3; namely, "system" and "environment" interact within a closed circulation, i.e. the domain of output actions of the one is exactly the domain of input actions of the other.

The component-oriented specification according to the separation of "system" and "environment" may be achieved by the following proceeding (cf. [Dederichs 90, §3]):

1. divide Act into In and Out where In contains the input actions of "system", while Out contains the output actions of "system"; as mentioned, this determines the corresponding sets for "environment";
2. according to the selected splitting of Act, divide the safety and liveness conditions into two corresponding groups each;
3. now group the safety and liveness conditions, which should be renamed e.g. by prefixing $In\_$ or $Out\_$ appropriately, together in several, but suitable ways to obtain the two component specifications; the following very rough variants are recommendable:
   (a) $s' \sqsubseteq s \Longrightarrow (I\_safe[s'] \Longrightarrow O\_safe[s'])$ or (a') $O\_safe[s]$,
   (b) $I\_safe[s] \wedge I\_live[s] \Longrightarrow O\_live[s]$.

The treatment of the connecting switch along this proceeding reads as follows:
(A) rewritten safety and liveness conditions:

rel In_S1: Act^ω **such that** $In\_S1 = S1$;
rel Out_S2: Act^ω **such that**
$Out\_S2[s] \iff (actual[p,a,s] \wedge a \in send(t) \Longrightarrow conn(t)\S p > disc(t)\S p)$;
rel Out_S3: Act^ω **such that**
$Out\_S3[s] \iff (actual[p,a,s] \wedge a = disc(t) \Longrightarrow conn(t)\S p > disc(t)\S p)$;
rel Out_L1: Act^ω **such that**
$Out\_L1[s] \iff (p \sqsubseteq s \wedge conn(t)\S p > disc(t)\S p \Longrightarrow$
$\exists p' [\, p \sqsubseteq p' \wedge p' \sqsubseteq s \wedge conn(t)\S p' = disc(t)\S p' \,])$

(B) component specifications:

rel cs_sys: Act^ω **such that**
$cs\_sys[s] \iff Out\_S2[s] \wedge Out\_S3[s] \wedge (In\_S1[s] \Longrightarrow Out\_L1[s])$;
rel cs_env: Act^ω **such that** $cs\_env[s] \iff In\_S1[s]$.

One easily verifies that cs_sys and cs_env together form a suitable component-oriented specification for con_switch, i.e.

$$\forall s \,[\, cs\_sys[s] \wedge cs\_env[s] \Longrightarrow con\_switch[s] \,],$$

even that "$\Longleftrightarrow$" holds.

Now, we are able to prove the correctness of the differential strategic system against the specification of the "system" part:

Theorem 5 *$\partial\Sigma_{con\_switch}$ is a causality structure for cs_sys, i.e.*

$$\forall s{:}Act^{\omega}\left[\begin{array}{l} s \in traces(\partial\Sigma_{con\_switch}) \Longrightarrow cs\_sys[s] \wedge \\ cs\_sys[s] \Longrightarrow \forall i{:}In, o{:}Out, p, q{:}Act^{\omega}\,[\,s = p\&\langle o,i\rangle\&q \Longrightarrow cs\_sys[p\&\langle i,o\rangle\&q]\,]\end{array}\right]$$

holds.

Proof: The first part of the condition is clear from:

$$s \in traces(\partial\Sigma_{con\_switch}) \stackrel{\mathrm{Th}_{\cdot}4}{\Longrightarrow} con\_switch[s] \Longrightarrow cs\_sys[s].$$

The second part first deserves some explanation: as we model asynchronous communication, the output may be postponed due to some delay; therefore, for all such modified traces, one is obliged to show that they can be carried out by the component.

Let $s \in Act^{\omega}$ be such that $cs\_sys[s]$, and let p, i, o, q be such that $s = p\&\langle o,i\rangle\&q$. Then, put $\tilde{s} := p\&\langle i,o\rangle\&q$. We have to show: $cs\_sys[\tilde{s}]$.

Assume $o \in send(t) \cup disc(t)$. If $i \notin conn(t)$, we are done, because $p\&\langle i\rangle$ satisfies the safety conditions for station t, as p does. Assume $i \in conn(t)$. $p\&\langle i,o\rangle$ is safe, i.e. it satisfies Out_S2 and Out_S3, and so does $\tilde{s}$. For $p\&\langle i,o\rangle$ does not satisfy In_S1 (either p itself does not satisfy In_S1, or $p\&\langle o\rangle$ and, thus, p satisfy In_S1, but then $p\&\langle i\rangle$ surely does not), it immediately follows that $\neg In\_S1[\tilde{s}]$. Thence, we have $In\_S1[\tilde{s}] \Longrightarrow Out\_L1[\tilde{s}]$, and $cs\_sys[\tilde{s}]$ is completely established. □

5. Functional Specifications

Design specifications consider the concerned distributed systems as a family of components, where the components may be more or less detailed parts of the system. Components of a distributed system exhibits their behaviour by taking input and forming output as a corresponding reaction, i.e. there is assumed a functional relation between the performed input actions and the performed output actions.

The formal description technique of functional specifications expresses this by describing components of a distributed system by *communicating agents*, which are connected by *directed asynchronous channels*. A communicating agent may have an arbitrary, but finite number of input and output ports depending on a decision of the designer.

As a basic technique, communicating agents may be modelled by a predicate over stream processing functions. Stream processing functions take the input stream(s) and produce from it the output stream(s) continuously. When the predicate does not specify the stream processing function uniquely, this means that the agent is intended to exhibit a nondeterministic behaviour.

Unfortunately, the continuity requirement of stream processing functions may lead to an inconsistent specification, i.e. the predicate has an empty truth set: if two actions occur on two different channels, one never can reconstruct the causal ordering of these two actions, even if this is desired.

Now, in this section we first describe the technique of using functional specifications with partial order processing functions, which tackle the drawback of ordinary stream processing functions by the ability of expressing the causality of actions explicitly.

The continuity requirement of stream processing functions works for our example as a design decision that provides the transition from design specification to abstract program. This is described in 5.2, where also the technique of using stream processing functions is explained.

5.1 Partial Order Processing Functions

So far we have modelled concurrency by the principle of interleaving: $a \| b$ is rather expressed by the set of traces $\{\langle a,b\rangle, \langle b,a\rangle\}$. This view may be also communicated itself to the streams of inputs and outputs belonging to agents described by stream processing functions. A serious drawback of the view has been already mentioned: if we distribute the actions onto more than one channel, the causal ordering of these is abandoned and unreconstructable.

If one wants to reflect the causal ordering of executed actions more directly, one uses a technique supporting explicit concurrency modelling. For this, we describe the usage of *partial orders* instead of the merely sequential ordering of traces: one introduces a sort of events and provide it with an ordering relation expressing the causality of occurrences; furthermore, we label the events with the actions to be executed in their right place. Now, if we use continuous functions mapping such partial orders to each other, we represent the causality of incoming or of outgoing actions of an agents directly, and we do not need more than one input and one output port each for communications between *partial order processing* agents.

Now, the definition of the sort of **labelled partial orders** (lpo's) over Act reads as follows:

used sort $Events$;
sort $Act^{\wp}$;
fun $\mathcal{E}$: $Act^{\wp} \to Pot(Events)$;
fun α: $\downarrow Act^{\wp} \downarrow \times Events \to Act$
  such that $\forall p{:}Act^{\wp}, e{:}Events\,[\,\delta_{Act}[\alpha_p(e)] \iff e \in \mathcal{E}(p)\,]$;
rel $\prec$: $Events \times \downarrow Act^{\wp} \downarrow \times Events$
  such that $\forall p{:}Act^{\wp},\, e, e'{:}Events$
$$\left[\begin{array}{l}(\delta_{Bool}[e \prec_p e'] \iff \{e,e'\} \subseteq \mathcal{E}(p)) \wedge \\ (e \prec_p e' \implies \left(\begin{array}{l} e \neq e' \ \wedge\ e' \not\prec_p e \ \wedge \\ \forall e''\,[\,e \not\prec_p e'' \ \vee\ e'' \not\prec_p e'\,]\end{array}\right))\end{array}\right].$$

We introduce the causality relation $\leq_p$ based on the direct causality $\prec_p$:

rel $\leq$ **overload**: $Events \times \downarrow Act^{\wp} \downarrow \times Events$
  such that $\forall p{:}Act^{\wp}$
$$\left[\begin{array}{l}\leq_p = \prec_p^* \cup \prec_p^{\infty} \ \wedge \\ \forall e{:}Events\,[\,e \in \mathcal{E}(p) \implies \exists e_0\,[\,e_0 \leq_p e \wedge minimal[e_0,p]\,]\,]\end{array}\right];$$
rel $minimal$: $Events \times Act^{\wp}$ **such that**
  $\forall e{:}Events, p{:}Act^{\wp}\,\Big[\,minimal[e,p] \iff$
    $e \in \mathcal{E}(p) \wedge \forall e'\,[\,e' \in \mathcal{E}(p) \wedge e' \leq_p e \implies e = e'\,]\,\Big]$.

The second condition in the specification of $\leq_p$, which also influences the one of $\prec_p$, is called the **axiom of bounded causes** or the **axiom of the process start**, because it prescribes that a process cannot be started from infinity, whereas a process is able to be continued arbitrarily; this is also called **left-boundedness** of processes.
On lpo's there analogously exists a prefix ordering:

rel $\sqsubseteq$ **overload**: $Act^{\wp} \overset{\downarrow}{\times} Act^{\wp}$
  such that $\forall p, p'{:}Act^{\wp}\,\Big[\,p \sqsubseteq p' \iff$
    $p = p'|_{\mathcal{E}(p)} \ \wedge\ \forall e, e'\,[\,e \in \mathcal{E}(p) \wedge e' \in \mathcal{E}(p') \wedge e' \leq_{p'} e \implies e' \in \mathcal{E}(p)\,]\,\Big]$;
fun $|$: $Act^{\wp} \times \downarrow Pot(Events) \downarrow \to Act^{\wp}$
  such that $\forall p{:}Act^{\wp},\, E{:}Pot(Events)$
$$\left[\begin{array}{ll}(1) & \forall e\,[\,e \in \mathcal{E}(p|_E) \iff e \in \mathcal{E}(p) \cap E\,] \wedge \\ (2) & \forall e\,[\,e \in E \implies \alpha_{p|_E}(e) = \alpha_p(e)\,] \wedge \\ (3) & \forall e, e'\,[\,\{e,e'\} \subseteq E \\ & \qquad \implies (e \prec_{p|_E} e' \iff e \prec_p e')\,]\end{array}\right].$$

Like for streams, we are able to construct an *actual* predicate:

$$\begin{array}{l}\textbf{rel}\ actual\colon Act^{\wp} \times Act \times Act^{\wp}\ \textbf{such that}\\ \quad \forall p,q\colon Act^{\wp},\ a\colon Act\ \Big[\ actual[p,a,q] \iff \\ \qquad |\mathcal{E}(p)| < \infty\ \wedge\ p \sqsubseteq q\ \wedge\ \exists e\,[\,minimal[e, q|_{Events\setminus\mathcal{E}(p)}] \wedge \alpha_p(e) = a\,]\ \Big].\end{array}$$

Finally, the set of all **partial order processing functions** is denoted by $(\mathcal{A}^{\wp} \to \mathcal{B}^{\wp})$.

With these concepts, we are able to specify the connecting switch by:

$$\begin{array}{l}\textbf{rel}\ popf\_stations\colon (In^{\wp} \to Out^{\wp})\ \textbf{such that}\\ \quad \forall \phi\colon (In^{\wp} \to Out^{\wp})\ \Big[\ popf\_stations[\phi] \iff \\ \qquad \forall p\colon Act^{\wp}\,[\,\exists E\colon Events \to Pot(Events)\,[\\ \qquad\qquad \forall e\,[\,e \in \mathcal{E}(p) \implies \left(\begin{array}{l} E(e) \subseteq \mathcal{E}(\phi(p)) \wedge \\ \exists \partial C\colon \partial\mathcal{C}\,[\,|E(e)| = \#\partial C(\alpha_p(e))\,]\ \wedge \\ \exists e_d \left[\begin{array}{l} e_d \in E(e) \wedge \\ \forall e_E\,[\,e_E \in E(e) \implies e_E \leq_{\phi(p)} e_d\,] \wedge \\ (\alpha_{\phi(p)}(E(e)\setminus\{e_d\}) \subseteq send(t)\ \wedge \\ \alpha_{\phi(p)}(e_d) = disc(t)) \\ \qquad \textbf{where}\ \alpha_p(e) \in conn(t) \end{array}\right] \end{array}\right)\,] \wedge \\ \qquad \forall e,e'\,[\,\{e,e'\} \subseteq \mathcal{E}(p) \wedge e \neq e' \implies E(e) \cap E(e') = \emptyset\,] \wedge \\ \qquad \bigcup\{E(e)\,|\,e \in \mathcal{E}(p)\} = \mathcal{E}(\phi(p)) \wedge \\ \qquad \forall e_1,e_2\,[\,\{e_1,e_2\} \subseteq \mathcal{E}(\phi(p)) \implies \\ \qquad\quad (e_1 \prec_{\phi(p)} e_2 \iff \Xi(e_1) \preceq_p \Xi(e_2) \\ \qquad\qquad \textbf{where}\ \forall e[e \in \mathcal{E}(\phi(p)) \implies e \in E(\Xi(e))])\,]\,]_{\exists E}\,]_{\forall p}\ \Big].\end{array}$$

The specification says the following: every lpo p containing only *conn* actions is moved by a ϕ such that $popf\_stations[\phi]$ to a lpo $\phi(p)$, where the events of p are replaced by "pieces" of partial orders representing the reaction of the agent. Such a reaction of the agent is determined by the set $\partial\mathcal{C}$ of differential strategies of the "system" part: on any $conn(t)$ action it does an (finite) amount of $send(t)$ actions closing with a $disc(t)$ action. As expressed in the conditions for e_d it is not necessary to execute all $send(t)$ actions sequentially, but they may be executed in an arbitrary order; only the $disc(t)$ action is fixed, it marks the end of each reaction (cf. FIGURE 4).

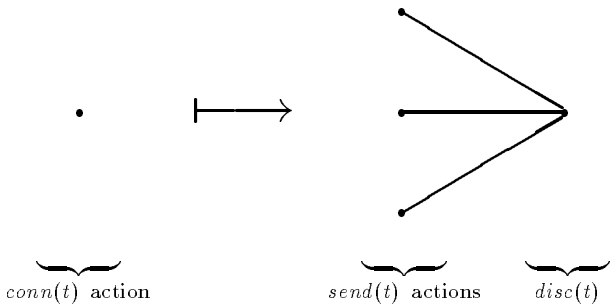

Figure 4: Example Reaction of $popf\_stations$

Unlike for the other techniques, we do not treat the topic of correctness of the given specifications here, except the informal argument that the aimed behaviour is completely determined by that set $\partial\mathcal{C}$ of differential strategies belonging to the differentially strategic system $\partial\Sigma_{con\_switch}$, which have been proved to be a causality structure for the specification of the "system" part in Th.5, and only the connection between the differentially strategic system and the partial order processing function constructed from it has to be established, while this is in our case easy to do: there is an "epimorphism" between the $\phi(p)$'s and the p's; namely, the one "compressing" $E(e)$ to e.

5.2 The Continuity Requirement of Stream Processing Functions as a Design Decision

Starting from a requirement specification, a design specification is achieved by the selection of components that the distributed system shall be divided into. Components may be regarded as communicating agents, which process their input and give their output as reaction. Whereas partial order processing functions reflect the causal ordering of the input and output, stream processing functions provide input and output channels with sequentially flowing messages.

More formally, each function in $(\mathcal{A}^\omega \to \mathcal{B}^\omega)$ is called a **stream processing function**; a **stream processing agent** is the extension of this notion to functions processing tuples of streams, where the underlying ordering of the stream tuples is the canonically extended prefix ordering.

In our example, the situation is like depicted in FIGURE 3: two stream processing functions *system* and *environment* are to be specified; these are combined together by a fixed point equation representing the closed circulation.

We only concentrate on the specification of *system*; we regard *environment* as to be implemented by someone else following the *open system view*.

An agent specification consists of a predicate over stream processing functions; thus, an agent is rather represented by a *set* of stream processing functions. This is because one wants to allow the concerning agent to exhibit a nondeterministic or, equivalently, an underspecified behaviour.

One method of building agent specifications is to define the concerning predicate recursively, i.e. it is obtained as the weakest solution of a predicate equation, which is commonly specified in the following form:

rel P: $(\mathcal{A}^\omega \to \mathcal{B}^\omega)$ **such that** $\forall f \left[P[f] \iff formula[P, f] \right]$,

where $formula[P, f]$ is mostly represented as conjunction of formulae of the form

$\exists g\,[\,P[g] \wedge \forall i, x\,[\,f(i\&x) = reaction(i)\&g(x)\,]\,]$

and of the form $f(\varepsilon) = initial\_reaction$.

The specification of *system* reads now as follows:

$$\begin{array}{l}\textbf{rel}\ spf\_stations\text{: } (In^\omega \to Out^\omega)\\ \quad \textbf{such that}\ \forall f \Big[spf\_stations[f] \iff\\ \qquad f(\varepsilon) = \varepsilon\ \wedge\\ \qquad \exists g, \partial C\,[\,spf\_stations[g] \wedge \forall a, x\,[\,f(\langle a\rangle\&x) = \partial C(a)\&g(x)\,]\,]\Big],\end{array}$$

where $\partial\mathcal{C}$ is the set of differential strategies to be applied by "system" belonging to $\partial\Sigma_{con\_switch}$.

The first proof obligation for functional specifications with stream processing functions does not concern any verification question, but we have to prove that the specification is able to be satisfied by at least one continuous stream processing function.

Proposition 6 *spf_stations is consistent, i.e.* $\exists f\,[\,spf\_stations[f]\,]$ *holds.*

Proof: Take that f such that f maps each $conn(t)$ action to $disc(t)$ within the processed stream. f is clearly continuous and satisfies *spf_stations*. □

In order to verify an agent specification against a trace specification, one has to show that the traces which can be generated by the agent meet the corresponding requirement specification. The

set of the (normal form) traces generated by an agent is defined as follows:

$$traces(P) := \{t{:}Act^{\omega} \mid \exists f\,[\,P[f] \wedge t \in traces(f)\,]\};$$

$$traces(f) := \left\{ t{:}Act^{\omega} \;\middle|\; \begin{array}{l} Out©t = f(In©t) \wedge \\ \forall t' \left[\begin{array}{l} (t' \sqsubseteq t \Longrightarrow Out©t' \sqsubseteq f(In©t')) \wedge \\ \forall i\,[\,actual[t', i, t] \Longrightarrow Out©t' = f(In©t')\,] \end{array} \right] \end{array} \right\}$$

The notion of a causality structure communicates itself from differentially strategic systems to agent specifications. Moreover, the correctness proof is achieved, when the agent specification forms a corresponding causality structure, because agents are connected by *asynchronous* channels, thus, output delay is possible.

But, because the differentially strategic system is directly involved in the construction of *spf_stations*, it seems to be more advantageous to prove that all (normal form) traces generated by *spf_stations* are also in $traces(\partial\Sigma_{con\_switch})$. For, if we have proved this, *spf_stations* obviously forms a causality structure for the *cs_sys* predicate, which is the component trace specification of *system*.

Theorem 7 *spf_stations forms a causality structure for cs_sys; therefore, spf_stations satisfies cs_sys.*

Proof: Fix a f such that $spf\_stations[f]$. What we have to show is:

$$traces(f) \subseteq traces(\partial\Sigma_{con\_switch}).$$

Let t be a trace of $traces(f)$. Assume $t \neq \varepsilon$, for ε is already in $traces(\partial\Sigma_{con\_switch})$. But then, we get a $t' \in Act^{\omega}$ and $i \in In$ such that $t = \langle i\rangle \& t'$, since $f(\varepsilon) = \varepsilon$.

Case 1: $Out©t' = t'$: it immediately follows that

$$f(\langle i\rangle) = f(In©t) = Out©t = Out©t' = t'.$$

Therefore, there is a differential strategy $\partial C \in \partial\mathcal{C}$ such that $t = \langle i\rangle \& \partial C(i) \in traces_{+}(\partial\Sigma_{con\_switch})$.

Case 2: There are $o \in Out^{*}$ and $j \in In$ such that $t = \langle i\rangle \& o \& \langle j\rangle \& t''$: first put $s := \langle i\rangle \& o$. From the definition of $traces(f)$ we conclude

$$f(\langle i\rangle) = f(In©s) = Out©s = o.$$

Similarly to case 1, we get a ∂C such that $t = \langle i\rangle \& \partial C(i) \& \langle j\rangle \& t''$. But, we have also a g such that $spf\_stations[g]$ and

$$\partial C(i) \& (Out©t'') = f(\langle i\rangle \& [In©(\langle j\rangle \& t'')]) = \partial C(i) \& g[In©(\langle j\rangle \& t'')].$$

It is not too difficult to show $\langle j\rangle \& t'' \in traces(g)$. By some kind of induction hypothesis, we assume that $traces(g)$ is a subset of $traces(\partial\Sigma_{con\_switch})$. Then, one may conclude that t is, finally, also a trace of $\partial\Sigma_{con\_switch}$. □

It is an interesting exercise to prove that all traces of the differentially strategic system are traces of the agent specification. The usage of differentially strategic systems for the construction of agent specification is a methodically advantageous device for the transition from trace specifications as requirement specifications to functional specifications as design specifications, for the verification of a differentially strategic system may be achieved more easily and immediately yields the correctness of the corresponding functional specification.

The continuity requirement of stream processing function is an incisive design decision: the stations agent behaves in some kind of monotonous matter; each *conn* action of a station t get its reaction not interspersed by actions concerning other stations than t itself. Indeed, the way to obtain the agent specification has been all the same as for partial ordering processing functions, where, by the way, their continuity requirements do not influence the causality of input and output each.

On the other hand, as described in the next section, the form of *spf_stations* is already appropriate to give the corresponding definition within an abstract program. Therefore, we view the continuity requirement for our example also as a design decision leading to the level of implementation.

6. Implementations of a Connecting Switch Simulator

The implementation phase is divided into two parts: the level of an *abstract* program and the level of a *concrete* program. Each part is provided with an appropriate language interface towards a more algorithmic language.

In the case of abstract programs one wants to implement the concerning distributed system by an applicative program. The usage of a functional language as a direct continuation of the level of a functional specification seems to be obvious. The transition step from design specifications to abstract implementations may be carried out by a correctness proof that compares the denotational semantics of the abstract program with the set of stream processing functions implied by the design specification.

Concrete programs are written in a more procedural-styled language. Procedural or imperative languages are a closer interface to current machine architectures. Once the distributed system to be implemented is given by an abstract program, one may apply *transformational programming*-style techniques: the transition step from abstract to concrete implementations requires the development of a set of semantically sound transformation rules to obtain correct programs. The appearing proof obligation is then solved by showing that the concrete program is resulted by a correct application of the transformation rules to the abstract program.

So as to write implementations, we make use of two languages called AL+ (*a*pplicative *l*anguage) and PL+ (*p*rocedural *l*anguage) which are slight modifications of Frank Dederichs' developments, AL and PL, [Dederichs 91].

AL+ provides concepts of applicative languages concentrating on the processing of streams:

- nondeterminism (mainly restricted on reactions);
- composition of functions;
- function applications;
- conditional choice expressions by an **if – then – else – fi** construct;
- stream equations as commands, stream expressions as instructions
- specification of functional modules:
  - **program**s as main units specifying each that stream outcoming from the closed system,
  - **agent**s as nondeterministic functions processing streams,
  - **funct**ions as auxiliary nondeterministic functions

More precisely, single functional modules processing streams, i.e. agents and programs, are represented by couples of stream equations of the form $stream = expression$, guarded by declarations of e.g. subagents and so on. Thus, AL+ supports the hierarchical structuring of networks of agents.

Agent definitions usually correspond to predicates of stream processing functions that are specified recursively by the method described above except the consideration of intial reactions, which is repeated here for convenience:

rel P: $(\mathcal{A}^\omega \to \mathcal{B}^\omega)$ **such that** $\forall f \left[P[f] \iff formula[P,f] \right]$,

where $formula[P,f]$ is mostly represented as conjunction of formulae of the rough form

$\exists g \, [\, P[g] \wedge \forall i,x \, [\, f(i\&x) = reaction(i)\&g(x) \,]\,]$

AL+ is a typed language; each defined name of any sort has its boldface-type equivalent. The composed types are specially treated: stream types are written as **chan Sort**, product types are replaced by their component types and a special tuple processor as $(component_1, component_2)$, sequence types appropriate to express reactions are written as **sequ Sort**.

We do not go into detail of the formal definitions of AL and its denotational semantics. Now the abstract program of a connecting switch simulator reads as presented in FIGURE 5, where the *stations* agent is (almost) the direct translation of *spf_stations*.

---

program *con_switch_AL+* = → **chan Out** *c* :
  agent *stations* = **chan In** *e* → **chan Out** *c* :
    funct *C* = **In** *a* → **sequ Out** :
      select ⟨*disc*(*t*)⟩ **where** *a*∈*conn*(*t*) **tceles**
      or (**select** ⟨*send*(*t*, *m*)⟩ **where** *a*∈*conn*(*t*) ∧ *m*∈*Messages* **tceles**&*C*(*a*))
    endfunct
    c = *C*(ft *e*)&*stations*(rt *e*)
  endagent
  agent *environment* = **chan Out** *c* → **chan In** *e* :
    funct *E* = **Out** *b* → **sequ In** :
      if *b* ∈ *disc*
      then select ⟨*conn*(*t*, *r*)⟩ **where** *t*, *r*∈*Stations* **tceles or empty**
      else empty fi
    endfunct
    agent *init* = **chan In** *i* → **chan In** *o* :
      o = (**select** ⟨*conn*(*t*, *r*)⟩ **where** *t*, *r*∈*Stations* **tceles or empty**)&*i*
    endagent
    e = *init*(*E*(ft *c*)&*environment*(rt *c*))
  endagent
  c = *stations*(*environment*(*c*))
endprogram

Figure 5: An AL+ Program for a Connecting Switch Simulator

The initial reaction is treated by a function appending the outcome of the initial reaction to the whole output of *environment*. This is because of the view of streams being a sequential or, better, a FIFO ordered storage of messages: initial reactions are to be the first elements of every output stream.

In addition to AL+, PL+ comprises some of the usual concepts of procedural or imperative languages:

- variables;
- assignments;
- loops;
- procedures (mainly functional ones);
- streams turn to channels used in order to establish directed one to one communication;
- send (*channel*?*var*) and receive (*channel*!*expr*) commands.

PL+ distinguishes two kinds of agent declarations:

- so-called *sequential* or *imperative* agents, whose body is the usual pair consisting of variable declarations and of imperative statements;
- *parallel*, *hierarchical*, or *equational* agents, whose body, like for AL+ agent declarations, consists of a couple of agent calls of the same form as AL+ stream equations.

I.e. the parallel combinator is replaced by an applicative-style syntax:

$t = f(s), s = g(r)$ is synonymous to something like [**call** $f(s\#t)$‖**call** $g(r\#s)$].

As for AL+, we do not treat the syntax or the denotational semantics here more formally. The concrete program for a connecting switch simulator may be found in FIGURE 6.

---

program *con_switch_PL+* = → **chan Out** *c* :
  chan In *e*;
  agent *stations* = **chan In** *e* → **chan Out** *c* :
    var In *a*; **var Bool** *finished*;
    loop
      e?*a*; *finished* := *ff*;
      do
        [*c*!**select** *disc*(*t*) **where** *a*∈*conn*(*t*) **tceles**; *finished* := *tt*]
        or *c*!**select** *send*(*t*, *m*) **where** *a*∈*conn*(*t*) ∧ *m*∈*Messages* **tceles**
      until *finished* **od**
    pool
  endagent
  agent *environment* = **chan Out** *c* → **chan In** *e* :
    var Out *b*;
    e!**select** ⟨*conn*(*t*, *r*)⟩ **where** *t*, *r*∈*Stations* **tceles or stop**;
    loop
      c?*b*;
      if *b* ∈ *disc*
      then *e*!**select** *conn*(*t*, *r*) **where** *t*, *r*∈*Stations* **tceles or skip fi**
    pool
  endagent
  c = *stations*(*e*),
  e = *environment*(*c*)
endprogram

Figure 6: A PL+ Program for a Connecting Switch Simulator

The verification obligation of the concrete program against the abstract one, as already mentioned, is achieved by showing that the correct application of some set of transformation rules leads from the abstract one to the concrete one.

The transformation rules mainly considered translate AL+ agents into semantically equivalent or, at least, refining PL+ agents. Proving the soundness of the transformation rules, therefore, requires the satisfaction proof of two predicates over stream processing functions.

We do not treat the latter topic here either and refer you to [Dederichs 91] for more detailed information, but we give some of the transformation rules decisive to our example:

agent f = **chan u** $i \rightarrow$ **chan v** o :
    $o = F[\text{ft } i]\&f(\text{rt } i)$
endagent

---

recursion-to-iteration

agent f = **chan u** $i \rightarrow$ **chan v** o :
    var u x;
    loop
        $i?x$; $o!F[x]_1$; ...; $o!F[x]_{\#F[x]}$
    pool
endagent

agent f = **chan v** $i \rightarrow$ **chan v** o :
    $o = E\&i$
endagent

---

initial-reaction

agent f = **chan v** $i \rightarrow$ **chan v** o :
    var v x;
    $o!E_1$; ...; $o!E_{\#E}$;
    loop $i?x$; $o!x$ **pool**
endagent

7. Concluding Remarks

The example of a very simple protocol has been subjected to a development process complete upto the implementation level inclusive. We have also dealt with the proof obligations, which mainly concern verification problems, i.e. correctness questions.

It has turned out that the example of the connecting switch is mainly appropriate to elucidate the aspects of the trace specification technique. But, even if one has liked to develop a more realistic implementation of a connecting switch rather than of a connecting switch simulator, the example as treated here has served for making the several description techniques and their positions in the development life-cycle more transparent.

Acknowledgements

I wish to thank Manfred Broy for his kind encouragements and for his important correction remarks on draft versions. Furthermore, I am grateful to Frank Dederichs for his stimulating contrariness; he has made the importance of the Devil's Advocate for scientific discussion visible again; it should be noted that, besides that, his contributions [Dederichs 90] and [Dederichs 91] have incisively inspired

this work. But also the contributions of my colleagues to [Broy et al. 91] have been of great help. I still have to mention the useful comments of the two anonymous referees.

A part of this work was presented in a talk with the same title at the *Second COMPASS Subgroup Meeting on "Concurrency and Object-Orientation", held in Braunschweig, on June 27–28, 1991. COMPASS* (= A *COMP*rehensive *A*lgebraic approach to *S*ystem *S*pecification and development) is the ESPRIT Basic Research Action No. 3264.

Finally, this work was partially supported by the *Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Architekturen" ("Tools and methods for the exploitation of parallel architectures").*

References

[Broy 88] Broy, M., *Towards a design methodology for distributed systems, in:* Broy, M.(ed.), *Constructive Methods in Computing Science*, Springer NATO ASI Series F **55** (1989), 311–364

[Broy Dederichs Dendorfer Weber 91] Broy, M., F. Dederichs, C. Dendorfer, R. Weber, *Characterizing the behaviour of reactive systems by trace sets*, Technische Universität München, Technical Report, SFB-Bericht Nr. 342/2/91 A (1991)

[Broy et al. 91] Broy, M., F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, R. Weber, *The design of distributed systems — an introduction, in preparation*

[Dederichs 90] Dederichs, F., *System and environment: the philosophers revisited*, Technische Universität München, Technical Report TUM–I9040 (1990)

[Dederichs 91] Dederichs, F., *A Transformational Calculus for Efficient Implementations of Distributed Systems, in preparation as doctoral dissertation*

[Dederichs Weber 90] Dederichs, F., R. Weber, *Safety and liveness from a methodological point of view, in:* Information Processing Letters **36:1** (1990), 25–30