Towards a Formal Foundation of the Specification and Description Language SDL

Manfred Broy
Institut für Informatik
Technische Universität München
Postfach 20 24 20, 8 München 2

Abstract

For the CCITT specification and description language SDL which provides a graphical concept for representing designs of systems interacting by signals a formal model is described. It is based on functional descriptions of interactive systems by the concept of streams and stream processing functions. The foundations for this functional modelling of interactive systems are introduced. The graphical constructs of SDL are related to this model. In particular it is shown how the meaning of SDL graphical forms can be represented by specifications in the form of conditional equations or by functional programs. Based on the functional formal model the application of specification and verification methods for SDL is demonstrated. Small examples are treated. A number of tools are discussed that can be based on that model.

Note:

This study does not define a semantic model for the specification and design language SDL, but it gives

- an introduction to the functional modelling of distributed systems,
- an explanation how to translate SDL designs into functional descriptions,
- an extended example for the translation of an SDL design,
- explores the possibilities of supporting SDL designs by such a semantic foundation.

This study was carried out in cooperation with Siemens AG ZFE F2 SOF1.

1. Introduction

After several decades of extensive research for sequential systems a general framework for their design is available. The design of sequential systems follows classical patterns starting from an informal problem description going to a requirement analysis leading to a requirement specification. Afterwards a design specification is derived and from this design specification by a number of implementation decisions implementations for the specified system are derived. In principle all the steps involved can be done in a purely functional framework within a formal calculus or at least can be formally verified.

In the design of distributed programs questions of requirements, design, correctness and of behaviour are even more important than for sequential programs. The reason is quite obvious. Communicating programs and systems thereof exhibit two properties that make it practically impossible to test them in a sufficient way. On one hand they might be nondeterministic, on the other hand they might exhibit an infinite or at least unbounded behaviour. Furthermore distributed programs are very often used in applications the correctness and reliability of which are of high importance. They are used to control traffic systems, production control systems, and for all kinds of other software systems controlling physical processes. Their incorrect behaviour could lead to disasters. Moreover, distributed systems are more difficult to design and to understand due to their combinatorial complexity. Therefore a proper formal methodological framework for the design of distributed systems is urgently needed.

Nowadays the treatment of distributed systems in practical applications is carried out generally at a semiformal level. Especially graphical formalisms such as Petri-nets, graphical design representation languages such as MASCOT or SDL are very popular, since they give some overview on the system structure. However, such graphical formalisms are only helpful in a more technical way, if they provide a proper, precise, uniform, consistent model for a system. Here a formal model as well as formal semantic definition might be helpful. In addition, it may provide a basis for formal specification, design, and verification techniques for the graphical formalism such as the one of SDL.

Note that graphical formalisms like Petri-nets, MASCOT, or SDL are from a theoretical point of view not much different to textual formalisms - they just provide graphical representations instead of textual syntax. Often for these graphical formalisms in addition textual representations are provided.

There are many adjectives that have been and are used in connection with the type of systems described by SDL such as

(1) *cooperating, coordinated, communicating, reactive, interactive systems,*

(2) *concurrent, parallel systems,*

(3) *distributed systems,*

(4) *nondeterministic systems.*

All these adjectives refer to particular characteristics of systems that are described by SDL the formal models for which we may classify by the following two complementing views:

- models that describe the dynamic behaviours of systems,
- models that describe the internal structure of systems such as their components and communication connections.

In a requirement specification for instance we are mainly interested in the behaviours of a system, while in a design specification, we may be interested in structuring the system into a family of components. SDL with its suggested extensions provides graphical representations for both levels of description: sequence charts sketch behaviours and block diagrams describe the internal structure of systems in terms of their subsystems. Finally process diagrams describe the behaviour of system components at the lowest level. The different graphical forms can be related to a formal model for describing interactive systems.

In general an available formal framework serves two important purposes. First of all it gives a proper foundation such that it is clear what it means that a program is correct or that it can be verified. Second support tools that should give substantial support aid have to be based on formal methods. This is why formal methods get more and more into practical use, at least, if systems with high reliability are required.

Communication and interaction make system models more complex. Even in the design of sequential systems communication aspects may be vital. This is especially true if dialogue oriented systems are to be designed. Here classical techniques for specification and verification do not work without proper extension. Special calculi have to be provided. Nowadays it seems a general pragmatic assumption that many of the simple dialogue systems are to be designed by engineering techniques not so much taking into account formal verification techniques. This is true since such dialogue oriented systems often have rather simple control structures and therefore can be easily tested. However, this is certainly not true for components that are to be used within distributed systems cooperating with other interactive components. Here highly complex concepts of control and of data flow have to be considered that cannot be tested due to overwhelming combinatorial explosion. Often especially those systems are used in applications where reliability is vital. The classical applications for SDL show these characteristics. Therefore a proper methodology based approach to system design is necessary.

In the following sections after some short remarks on SDL we give an introduction into the functional description and modelling of communicating, interactive (reactive) systems. We introduce a functional model and a number of concepts and notions that are helpful in the specification, analysis, and verification of interactive systems. Then we relate SDL graphical concepts to this model. We treat some small examples. Finally we discuss possibilities of basing tools for supporting specification, design, verification, simulation and implementation on the introduced functional models.

2. The Specification and Description Language SDL

The language SDL has been developed by CCITT in 1976 and revised in 1976, 1980, 1984 and 1988 for use in the design of telecommunication systems. It is for the description of the behaviour and internal logical structure of telecommunication switches, but is also suited for modelling other applications.

SDL is said to be based on the concept of finite state machines. Strictly speaking, however, the SDL processes do not represent finite state machines, since they may exhibit (at least theoretically) infinite data spaces. In fact their control state space is finite. Hence systems are described by state machines with finite control state spaces and signals as input and output. For SDL both a textual and a graphical representation is provided. Both give a syntactic formalism for which a meaning is to be given.

This study is based on the SDL document [CCITT Z.100]. This document gives the syntax and an informal semantics of SDL.

In the following the SDL concepts are related to a functional treatment of systems by stream processing functions. Such a functional treatment has been worked out in [Broy 82] and [Broy 86] and applied to various language definitions and system specifications.

The advantages of a functional treatment are manifold. In particular it provides by the functional calculus a well-studied formal calculus for specification of and reasoning about systems. Furthermore it has nice properties of modularity: the meaning of a system building block that is itself composed from several subsystem building blocks can be derived from the meaning of these subsystem building blocks by appropriate compositional forms. This is demonstrated for SDL block diagrams. Therefore it is possible to reason about the overall behaviour of composed systems given sufficient information about the behaviours of subsystems.

3. Functional System Descriptions and State Machines

In this section we give an introduction into the basic concepts for the functional description and specification of interactive systems.

3.1 Stream-processing Functions

Let us start by introducing the concept of stream processing function and then relate it to state machines. A stream of elements over a given set of data M is a finite or infinite sequence of elements from M.

By M^* we denote the finite sequences over the set M. We denote the finite sequence consisting of the elements $x_1, ..., x_n$ by $\langle x_1 \ldots x_n \rangle$ or by $\langle x_{i+1} : i < n \rangle$. M^* includes the empty sequence which is denoted by $\langle\rangle$.

By M^∞ we denote the infinite sequences over the set M. M^∞ can be understood to be represented by the total mappings from the natural numbers $\mathbb{N}$ onto M. We denote the infinite sequence consisting of the elements $x_1, x_2,...$ by $\langle x_{i+1}: i < \infty \rangle$.

We denote the set of streams over the set M by M^ω. Formally we have

$$M^\omega = M^* \cup M^\infty$$

For simplicity we write also a^n for $\langle x_{i+1}: i < n \rangle$ with $x_{i+1} = a$ for all i, $0 \le i < n$. Similarly we write S^n for the set of all streams $\langle x_{i+1}: i < n \rangle$ with $x_i \in S$ for all i, $0 \le i < n$.

Streams can be understood to represent the history of communications between components of interactive systems. We introduce a number of functions on streams that are useful in system descriptions.

For every stream s we may define its *length*. The length is infinite or a natural number. It will be denoted by #s. Formally we have

$$\#: M^\omega \to \mathbb{N} \cup \{\infty\}$$

with

$$\#\langle x_{i+1}: i < n \rangle = n.$$

A classical operation on sequences is the *concatenation* which we denote by ^. The concatenation is a function that takes two sequences (say s and t) and produces a sequence as result starting with s and continuing with t. If s is infinite then the result of concatenating s with t yields s again. Formally we have for the concatenation the following functionality:

$$.\hat{}. : M^\omega \times M^\omega \to M^\omega$$

Its precise meaning is defined for streams $s = \langle s_{i+1}: i < n \rangle$ and $t = \langle t_{i+1}: i < m \rangle$ by

$$s\hat{}t = \langle \textbf{ if } i < n \textbf{ then } s_{i+1} \textbf{ else } t_{i-n+1} \textbf{ fi}: i < n+m \rangle$$

Note that the definition for ^ also works for infinite n (i.e. $n = \infty$).

On the set M^ω of streams we define a *prefix ordering* $\sqsubseteq$. We write $s \sqsubseteq t$ for streams s and t if s is a *prefix* of t. Formally we have for streams $s = \langle s_{i+1}: i < n \rangle$ and $t = \langle t_{i+1}: i < m \rangle$

$$s \sqsubseteq t \quad \text{iff} \quad n \le m \wedge \forall\, i, i < n: s_{i+1} = t_{i+1}$$

The prefix ordering defines a partial ordering on the set M^ω of streams. If $s \sqsubseteq t$, then we also say that s is an *approximation* of t. The set of streams ordered by $\sqsubseteq$ is even complete in the sense that every directed set $S \subseteq M^\omega$ of streams has a *least upper bound* denoted by $\bigsqcup S$. A set S which is subset of a partially ordered set is called *directed*, if

$$\forall\, x, y \in S: \exists\, z \in S: x \sqsubseteq z \wedge y \sqsubseteq z.$$

With the technique of least upper bounds of directed sets of finite streams we are able to describe infinite streams. Infinite streams are also of interest as fixpoints of prefix monotonic functions. Note that the streams associated with feedback loops in interactive systems correspond to such fixpoints.

A *stream processing function* is a function

$$f: M^{\omega} \to N^{\omega}$$

that is *prefix monotonic* and *continuous.* The function f is called *monotonic,* if for all streams s and t we have

$$s \sqsubseteq t \Rightarrow f.s \sqsubseteq f.t.$$

For better readability we often write for the function application f.x instead of f(x). The function f is called *continuous,* if for all directed sets $S \subseteq M^{\omega}$ of streams we have

$$\bigsqcup \{f.s: s \in S\} = f.\bigsqcup S$$

If a function is continuous, then its results for infinite input can be already predicted from its results on all finite approximations of the input. Note that infinite elements (such as infinite streams) are used for giving meaning to recursive declarations in terms of least fixpoints.

By $\perp$ we denote the pseudo element which represents the result of nonterminating computations. We write $M^{\perp}$ for $M \cup \{\perp\}$. Here we assume that $\perp$ is not an element of M. On $M^{\perp}$ we define also a simple partial ordering by:

$$x \sqsubseteq y \quad \text{iff} \quad x = y \vee x = \perp$$

We use the following functions on streams

$$ft: M^{\omega} \to M^{\perp},$$

$$rt: M^{\omega} \to M^{\omega},$$

$$.\&.: M^{\perp} \times M^{\omega} \to M^{\omega}.$$

They are defined as follows: the function ft selects the first element of a stream, if the stream is not empty:

$$ft.\langle s_{i+1}: i < n \rangle = \begin{cases} \perp & \text{if } n = 0 \\ s_1 & \text{if } n > 0 \end{cases}$$

The function rt deletes the first element of a stream if the stream is not empty:

$$rt.\langle s_{i+1}: i < n \rangle = \begin{cases} \langle\rangle & \text{if } n = 0 \\ \langle s_{i+2}: i < n-1 \rangle & \text{if } n > 0 \end{cases}$$

We sometimes use iterated applications of the function rt. Then we write:

$rt^0.s = s$,

$rt^{k+1}.s = rt^k(rt.s)$,

$rt^{\infty}.s = \langle\rangle$.

The function & appends an element to a stream, if the element is defined:

$$x \,\&\, \langle s_{i+1} : i < n\rangle = \begin{cases} \langle\rangle & \text{if } x = \bot \\ \langle s_i : i < n+1\rangle & \text{if } x \neq \bot \text{ with } s_0 = x \end{cases}$$

The definition of this function has been chosen carefully to keep it monotonic and continuous w.r.t. the prefix ordering. However, there are also nonformal reasons for taking this definition: as soon as one tries to send the value x of an expression as the first member of a stream the computation of which never terminates (i.e. $x = \bot$) the transmission of the first element can never take place and the resulting stream is empty.

Sometimes it is useful to work with a filter function on streams. Given a set $S \subseteq M$ and a stream $x \in M^{\omega}$ we write S©x for the substream of x consisting only of the elements of x contained in S. Formally we define

$S©\langle\rangle = \langle\rangle$,

$S©(d \,\&\, x) = d \,\&\, (S©x)$ if $d \in S$,

$S©(d \,\&\, x) = S©x$ if $\neg(d \in S)$.

For convenience we introduce the concept of a *resumption* f_x of the stream processing functions f for the input stream x. It is specified by

$f_x(s) = f(x\hat{}s)$

The prefix monotonicity of a stream processing function is essential for its interpretation as representation of the meaning of a communicating device. A property equivalent to monotonicity w.r.t. the prefix ordering is the following

$f(s\hat{}r) = f.s\hat{}g.r$ where for all streams t we have $g.t = rt^{\#f(s)}(f(s\hat{}t))$

This equation shows that for a prefix monotonic function f the output of $f(s\hat{}r)$ is concatenated from the output f.s produced by f on input s and the output produced by g on the further input r. Thus every prefix of the input determines a prefix of the output.

Note that the operations ft, rt, and & are prefix monotonic and continuous, but the concatenation ˆ as defined above is not prefix monotonic.

Prefix monotonicity reflects a characteristic property of interactive systems: communicated data cannot be changed after being shown as output. If f.s is the stream that results as output from the input of the stream s then if we continue with the input r, i.e. give after all the input s^r, then we get some additional output g.r, i.e. obtain after all the output f.s^g.r. In addition monotonicity gives the formal platform for handling communication in feedback loops: feedback is translated to fixpoint equations, which are known to have solutions, if the involved functions are monotonic.

3.2 Stream Processing Functions and State Machines with Input and Output

A stream processing function f can be easily understood as a state machine with input and output. This will be explained in detail in this section.

A *deterministic state machine* is given by the quintuple (S, I, O, δ, σ_0) the components of which denote the following items:

- a set of states S,
- a set of input elements I,
- a set of output elements O,
- a transition function with input and output:
  $\delta: I \times S \to O \times S$,
- an initial state σ_0.

Every stream processing function

$f: M^{\omega} \to N^{\omega}$

can be seen as a state machine $(M^{\omega} \to N^{\omega}, M^{\perp}, N^{\omega}, \delta, f)$ with

$\delta(m, f) = (f.\langle m \rangle, g)$ where for all x: $g.x = rt^{\#f.\langle m \rangle}(f(m\&x))$

Note that g in the definition above denotes the "state" of the considered machine after we have observed all the output caused by the input m. This shows that stream processing functions are a very concise and elegant representation of state machines with input and output. For explaining our notions let us consider a simple example:

Example: A simple store
A simple store that may store exactly one data element at a time from a given set D of data can be defined as follows: let the set of "input messages" M be defined by

$M = \{put.d: d \in D\} \cup \{get\}$.

We define a stream processing function f

$f: M^{\omega} \to D^{\omega}$

by the following equations (note that the store gets broken, if data is requested before data was written):

$f(put.d \,\&\, \langle\rangle) = \langle\rangle$,

$f(get \,\&\, s) = \langle\rangle$,

$f(put.d \,\&\, put.d' \,\&\, s) = f(put.d' \,\&\, s)$,

$f(put.d \,\&\, get \,\&\, s) = d \,\&\, f(put.d \,\&\, s)$.

These equations characterize the function f uniquely, since for every pattern of the input stream there is an equation. f denotes the initial state of the store where no value is stored and therefore any attempt to read the value by sending the message "get" leads to the empty output. If the first input is the message put.d then the value d is stored and given as an answer to the signal "get" as long as no other message put.d' arrives. The resumption $f_{\langle put.d\rangle}$ denotes the state of the store where the data element d is stored. ◊

Assume we send the input m (in SDL terminology the signal m) to a module (in SDL terminology process or block) represented by the function f. Then we obtain a finite or infinite stream y of output signals from f as a response to m. Formally we get

$$y = f.\langle m\rangle$$

Assume y contains $k \in \mathbb{N} \cup \{\infty\}$ elements, i.e. $k = \#y$. We specify the behaviour of the module after we have observed the input m and all the output y by the stream processing function g where for all streams x:

$$g.x = rt^k(f(m \,\&\, x)) = rt^k(f_{\langle m\rangle}(x))$$

So far we have shown that stream processing functions define state machines. It is not difficult to associate for a given deterministic state machine $(S, I, O, \delta, \sigma_0)$ with every state $\sigma \in S$ in a stream processing function

$$f_\sigma: I^\omega \to O^\omega$$

specified by

$$f_\sigma(i \,\&\, s) = o \,\&\, f_{\sigma'}(s) \qquad \text{where} \qquad (o, \sigma') = \delta(i, \sigma).$$

Note that this is a recursive definition for the family of functions f_σ. Due to its specific form there is a unique fixpoint that solves the defining equation. For each state σ the function f_σ has the same "input/output" behavior as σ.

Working with stream processing functions instead of state machines with input and output has a number of advantages. In particular we can use the functional calculus for reasoning about the behaviour of communicating systems.

The state machines considered so far are deterministic. Every input corresponds to exactly one output and a unique successor state. Now we show a functional model for nondeterministic state machines.

A *nondeterministic state machine with input and output* is given by the quintuple

$(S, I, O, \delta, \sigma_0)$

the components of which denote the following items:

- a set of states S,
- a set of input elements I,
- a set of output elements O,
- a transition relation with input and output:
$$\delta: I \times S \to \wp(O \times S)\backslash\emptyset,$$
- an initial state σ_0.

We consider predicates on stream processing functions (let $\mathbb{B}$ denote the set of truth values):

$$Q: (M^{\omega} \to N^{\omega}) \to \mathbb{B}$$

Every such predicate Q can be seen as a nondeterministic state machine

$$((M^{\omega} \to N^{\omega}) \to \mathbb{B}, M^{\perp}, N^{\omega}, \delta, Q)$$

with

$$\delta(m, R) = \{(f.\langle m\rangle, H): R.f \wedge \forall\, h: H.h \equiv \exists\, g: R.g \wedge f.\langle m\rangle = g.\langle m\rangle \wedge h.x = rt^{\#g.\langle m\rangle}(g(m\&x))\}$$

Note that H in the definition above denotes the "state" of the considered machine after we have observed all the output caused by the input m. This shows that predicates on (or equivalently sets of) stream processing functions are a very concise and elegant representation of state machines with input and output. The general representation of nondeterministic state machines predicates on or equivalently by sets of stream processing functions is possible, too. The definitions are rather technical and therefore are omitted.

3.3 Stream Processing Functions and State Transition Machines

A stream processing function f can also be understood as a state transition machine.

A *state transition machine* is given by the quadruple

$(S, A, \rightarrow, \sigma_0)$

the components of which denote the following items:

- a set of states S,

- a set of actions A,
- a transition relation $\xrightarrow{a} \subseteq S \times S$ for every action a in A,
- an initial state σ_0.

Every stream processing function

$$g: M^{\omega} \to N^{\omega}$$

can be seen as a state transition machine

$$(M^{\omega} \to N^{\omega}, \{in.x: x \in M^{\perp}\} \cup \{out.x: x \in N\}, \longrightarrow, g)$$

with

$$f \xrightarrow{in.d} h \text{ iff } \quad f(d\&s) = h.s,$$

$$f \xrightarrow{out.d} h \text{ iff } \quad f.s = d\&h.s.$$

This shows that stream processing functions are a very elegant representation also of simple state transition machines (which for convenience we also call state machines).

For explaining our notions let us consider again our simple example:

Example: A simple store
Let the store function f now be defined as above. With the modelling by a state transition machine input and output is produced in two completely independent steps. As the states of this state machine we get resumptions such as $f_{\langle put(d)\rangle}$ and also stream processing functions g with a lot of pending output such as

$$g.s = f(put.d \,\&\, get \,\&\, get \,\&\, s)$$

which is the state obtained after the actions in(put(d)), in(get), in(get). In contrast to this state machines with input and output show all the output caused by the input in one step. In state transition machines the connection between input and output is more loosely given: an output caused by some input may appear much later. ◊

For a state machine a (finite) trace t is a finite sequence $\langle t_1 \ldots t_n\rangle$ of actions, i.e. there exist states σ_i, such that for all i, $1 \le i \le n$:

$$\sigma_{i-1} \xrightarrow{t_i} \sigma_i$$

For a stream processing function f which can be understood as a finite state machine a stream t of input and output actions is a trace, if the predicate trace(f, t) holds, where:

$$trace(f, \langle\rangle) = true,$$

$$trace(f, in.d \,\&\, t) = trace(f_{\langle d\rangle}, t),$$

$$trace(f, out.d \,\&\, t) = \exists\, g: trace(g, t) \wedge \forall\, x: f.x = d\&g.x.$$

Every predicate Q on stream processing functions:

$$Q: (M^{\omega} \to N^{\omega}) \to \mathbb{B}$$

can again be seen as a state transition machine

$$((M^{\omega} \to N^{\omega}) \to \mathbb{B}, \{in.x: x \in M^{\perp}\} \cup \{out.x: x \in N\}, \longrightarrow, Q)$$

with

$$R \xrightarrow{in.d} H \quad \text{iff} \quad \forall\, g: H.g \equiv \exists\, f: R.f \wedge \forall\, s: g.s = f(d\&s),$$

$$R \xrightarrow{out.d} H \quad \text{iff} \quad \exists\, g: H.g \wedge \forall\, g: H.g \equiv \exists\, f: R.f \wedge \forall\, s: f.s = d\&g.s.$$

This shows that stream processing functions and predicates specifying them are a very elegant representation also of particular state transition machines (which for convenience we also call state machines). However, not every state machine can simply be represented by stream processing functions. First of all the actions have to be characterized into input and output actions. Additional assumptions about the input and output actions are needed, such as the assumption that every input action is always (in every state) possible. Nevertheless, the definitions above allow us to associate traces with stream processing functions acting as states of a state machine. This will be used later to associate sequence charts with process diagrams in SDL.

4. Relating SDL Concepts to Functional System Descriptions

In this section we give a first informal and incomplete explanation how SDL graphical system descriptions can be related to functional system descriptions. Here we concentrate on the aspect of system interaction and only briefly touch those aspects of SDL that are not related to this issue. We in particular do not discuss any syntactic issues of SDL.

4.1 Relating SDL Terminology to Functional System Descriptions

In SDL a fixed terminology is introduced without giving a formal model for the used notions. In this section we start by relating the SDL terminology to our functional model for distributed systems.

We choose the following formalisation within the functional model for the SDL terminology as used in [CCITT Z.100]:

signals	are elements of special sorts or sets that are used as messages,
channels	are identifiers for a stream (or two streams in the case of bidirectional channels),
blocks	are (specifications of) subsystems,
processes	are extended state machines, represented by stream processing functions.

At a first level of detail a SDL specification of a system is defined by giving (cf. [CCITT Z.100]):

- a system name,
- signal and signal list definitions that are modelled by particular data sets representing signals,
- channel definitions that correspond to identifiers and include references to the signal sets,
- data definitions that correspond to sets defined by sorts of algebraic specifications,
- block definitions that correspond to names of subsystems,
- macros (which we do not consider in this study).

A block models a subpart of a system and may include the following parts

- block name,
- signal and signal list definitions that are modelled by particular data sets representing signals,
- signal route definitions,
- process definitions that specify the behaviour of processes or process types,
- data definitions that correspond to sets defined by sorts of algebraic specifications.

We do not treat in the following the naming conventions of SDL which can be formalized by using the concept of environments from denotational semantics, but rather concentrate on the modelling of the behaviour.

4.2 Data, Signals, and Timing in SDL

In SDL data are modelled by abstract data types. This is nowadays quite well-understood. Therefore in this study we concentrate on aspects of interaction and do not treat the data type aspects of SDL explicitly.

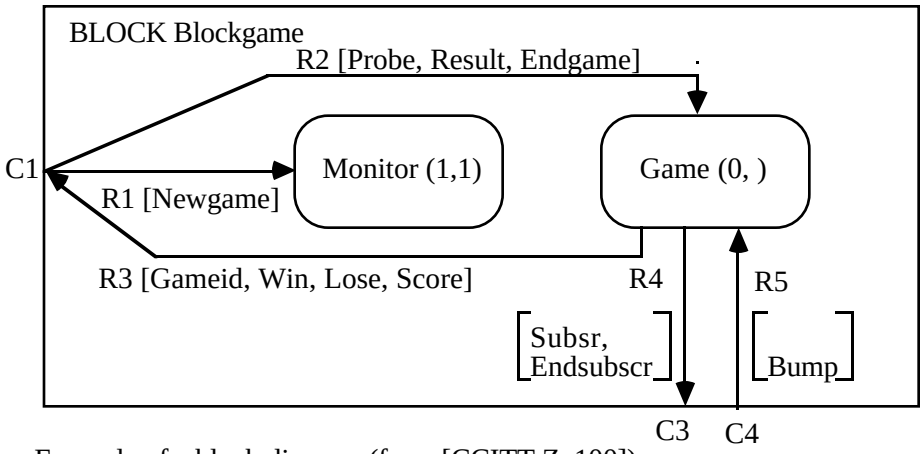
A signal corresponds to a message that is to be sent. Each signal carries the process instance identifier with it where it came from. Signals are denoted in SDL in process diagrams in a schematic way. So a signal denotation A stands for a class SIGNAL.A of signals matching the pattern A. Furthermore for every signal in SIGNAL.A the pattern A defines an update of the data state.

Time is represented by a special signal T. The arrival of the signal T can be seen as the message that one unit of time has passed. T therefore may cause a timeout.

4.3 Blocks and Processes in SDL

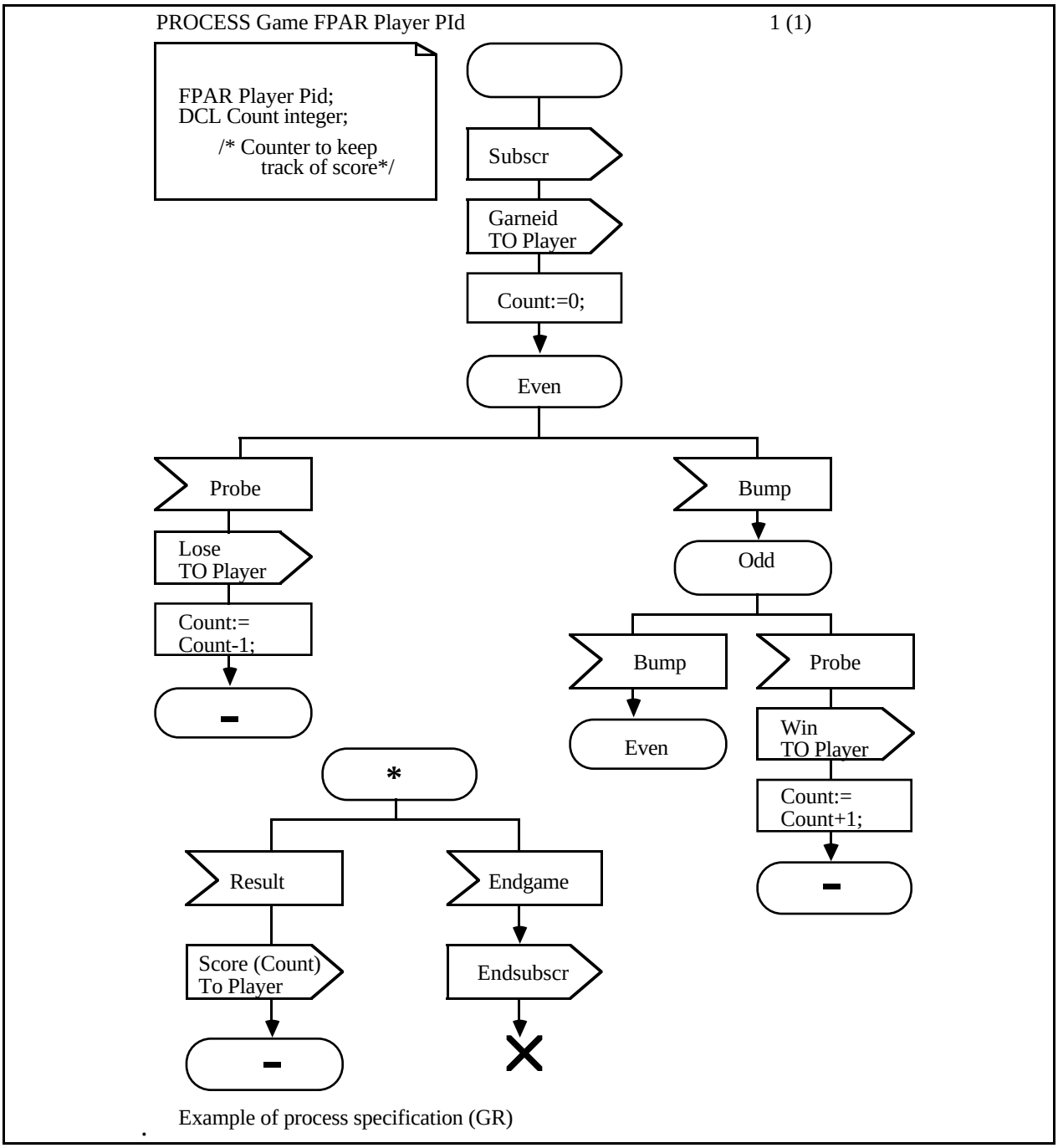
A block describes a system or subsystem. It is mainly defined by its signal lists that indicate which sets of signals are processed and by the description of its block diagram and/or the processes by process diagrams that are contained in the block.

A block diagram consists of a number of subblocks and connecting lines.



Example of a block diagram (from [CCITT Z. 100])

The behaviour of a block can also be described by a process diagram



Example of process specification (GR)

A process in SDL is semantically defined by a process type represented by a process diagram. It contains

- a local state given by the set of variables,
- a number of communication connections.

The form of the local state is not relevant for the outside behaviour of a process. It can be seen of as an auxiliary construction for specifying the behaviour of a process.

The interface of a process is given by the set M of external input signals and the set N of external output signals that may occur for the process.

In the functional model the behaviour described by a process diagram is represented by a set of stream processing functions. Formally a process diagram defines to a class of stream processing functions that map the stream of input signals to the stream of output signals for each of its control states. Let M denote the set of input signals and N denote the set of output signals. With a process diagram we associate a subset of the function space

$$M^{\omega} \rightarrow N^{\omega}.$$

Every control state in the process diagram corresponds also to a subset from that function space. Which particular set of functions is defined by a process diagram will be specified in the following. We associate with SDL graphs logical statements that provide formal specifications for the stream processing functions associated with the control states of a process.

4.4 Channels

Channels connect blocks in an unidirectional or bidirectional way. A channel includes

- a channel name,
- communication paths that give the origin and the destination of the signals,
- signal lists which indicate signals of which sorts are transported on the channel,
- a channel substructure definition.

In the functional model the behaviour of a system w.r.t. an unidirectional channel is given by a stream. A bidirectional channel corresponds to two streams.

4.5 Control States, Data States, and Variables in SDL

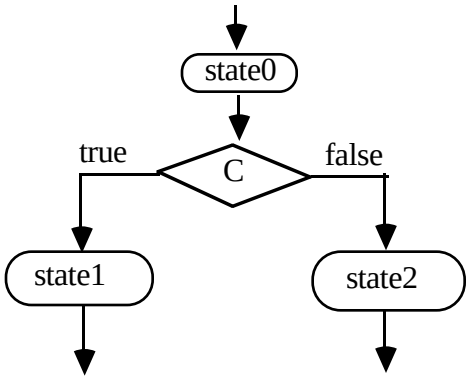
For SDL process diagrams local program variables can be declared. However, they are always local to the process instance which may change their values. Given a SDL process with input signal set M and output signal set N which declares the local variables

$v_1, ..., v_n$ which take values from the sets $D_1, ... , D_n$

and have the initial values $d_1, ..., d_n$, then the data state space D of this process is given by the set $D_1 \times ... \times D_n$. Let M be the set of input signals and N be the set of output signals of a SDL process diagram. We associate with each of the process control states s predicates [s] characterizing functions of the functionality

$$[s]: \ D \rightarrow ((M^{\omega} \rightarrow N^{\omega}) \rightarrow \mathbb{B}) \ .$$

The parameter from D is called the *local data state*. So for every element $d \in D$ we obtain by the predicate ([s].d).f the set of functions f the correspond to possible behaviours of the system. The data state parameter may be used for example for determining the behaviour of systems with branches:

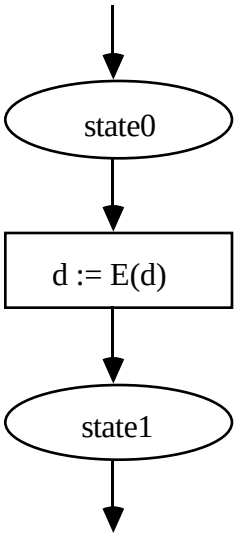


The information contained in this diagram is represented by the equations (assume that the predicates [state0], [state1] and [state2] resp. are associated with the control states state0, state1, state2)

$[state0].d \equiv [state1].d \Leftarrow C.d$,

$[state0].d \equiv [state2].d \Leftarrow \neg C.d$.

Similarly we may model tasks that update a local state. Consider the SDL diagram:



The information contained in this diagram can be translated into the following equation for the predicates associated with the control states:

$[state0].d \equiv [state1].E.d$.

By this definition the statement d := E(d) corresponds to an update of the local data state.

4.6 Input and Output in SDL

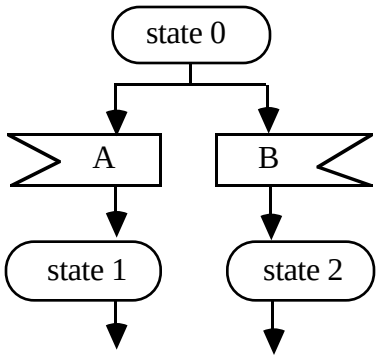
A control state is a notion that occurs within a process diagram. Every process diagram has an initial control state and a set of intermediate control states. A predicate

$$[s]: D \to ((M^{\omega} \to N^{\omega}) \to \mathbb{B})$$

is associated every control state s.

In SDL a control state is called "a point in the process where no actions are being performed". In the terminology of stream processing functions states are represented by predicates on stream processing functions. In the following we use states also for denoting points in a SDL diagram. They can be understood as *virtual* states that do not have explicit names in SDL diagrams. They can be replaced by arbitrary SDL diagrams according to the syntactic conventions.

In the graphical representation of SDL a state transition rule for a process is represented by a state transition diagram basically of the following form (the generalisation to input states with n branches is straightforward):



The information contained in such a diagram can be formalized in the state oriented view by instances of a state transition relation with

$$a \in \text{SIGNALS.A} \Rightarrow \text{state0} \xrightarrow{\text{in.a}} \text{state1},$$

$$b \in \text{SIGNALS.B} \Rightarrow \text{state0} \xrightarrow{\text{in.b}} \text{state2},$$

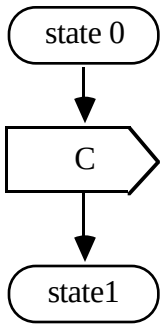
$$\neg(c \in \text{SIGNALS.A} \cup \text{SIGNALS.B}) \Rightarrow \text{state0} \xrightarrow{\text{in.c}} \text{state0}\ .$$

Since we associate with every state a predicate on stream processing functions we may assume the functions [state0], [state1], [state2] associated with the states state0, state1, state2 resp. Then the definitions above correspond to:

$$([\text{state0}].d).f \equiv \forall\ s:\ (ft.s \in \text{SIGNALS.A} \Rightarrow \exists\ g:\ f.s = g.rt.s \wedge ([\text{state1}].update(d,\ A,\ ft.s)).g)$$
$$\wedge$$
$$(ft.s \in \text{SIGNALS.B} \Rightarrow \exists\ h:\ f.s = g.rt.s \wedge ([\text{state2}].update(d,\ B,\ ft.s)).h)$$
$$\wedge$$
$$(\neg(ft.s \in \text{SIGNALS.A} \cup \text{SIGNALS.B} \cup \{\bot\}) \Rightarrow f.s = T\ \&\ f.rt.s)\ .$$

Here we assume a function update that for every input signal A or B defines how the data state has to be updated. If we assume a modelling without explicit representation of time in the third line the prefix "T &" has to be dropped.

Similarly we may formalise output. In SDL output that is produced from a given state state0 and then leads to a state1 is visualized by the following diagram:



The information contained in that diagram can be represented by the following instance of a state transition relation:

$$\text{state0} \xrightarrow{\text{out.C}} \text{state1}$$

This rule corresponds to:

$$([\text{state0}].d).f \equiv \forall\ s: \exists\ i, g: f.s = T^{i\wedge}(\text{result}(C, d)\ \&\ g.s) \wedge ([\text{state1}].d).g\ .$$

Here we assume a function result that defines for every output label C the output signal result(C, d). Before the output is produced a finite number of time ticks may occur. This reflects the principle of SDL that nothing can be assumed about the relative speed and quantitative timing.

Thus we may derive from a process diagram a set of equations for the stream processing functions associated with each of the states in the diagram.

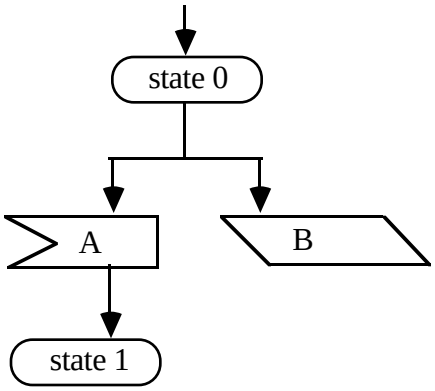
4.7 The Queueing Mechanism: Saved Signals and Enabling Conditions

In SDL signals may be queued, since one or more signals may be waiting for consumption when a process reaches a state. This queueing is modelled by the stream concept quite straightforward, since the model assumes implicit buffering.

If signals arrive simultaneously, according to the SDL User Guide-lines they are ordered arbitrarily in the input stream. This kind of nondeterminism is easily expressed by a merge function on the input streams. For a formal treatment see section 5.

The *save concept* in SDL gives a simple possibility to await a special signal while putting (saving) certain signals into a queue that are to be processed in later states. This is expressed by a special graphical representation.

If for a given state state0 we consider the following process diagram:



this expresses a *save* on B signals.

The information contained in this diagram can be modelled in the framework of stream processing functions as follows:

([state0].d).f ≡ ∀ t, s: #t < ∞ ∧ SIGNALS.B©t = t ⇒

(ft.s ∈ SIGNALS.A ⇒ ∃ g: f(t^s) = g(t^rt.s) ∧ ([state1].update(d, A, ft.s)).g) ∧

(ft.s ∉ SIGNALS.A ∪ SIGNALS. B ∪ {⊥} ⇒ f(t^s) = T & f(t^rt.s)) .

The equation shows that all the input of signals B is ignored but saved until a signal A arises and only then the signals B are taken into account.

Enabling conditions can be handled in a similar style. Assume E is an enabling condition on certain signals. Accordingly such a signal is only received if it fulfils the condition E. This is expressed by

([state0].d).f ≡ ∀ t, s:

∃ g: (f(t^s) = g(t^rt.s) ⇐ ¬enable(E, t) ∧ #t < ∞ ∧ E.ft.s) ∧ ([state1].update(d, A, rt.s)).g .

where

enable(E, ‹›) = false,

enable(E, a & y) = (E.a ∨ enable(E, y)).

This shows that rather similar logical equations can be used for expressing the save concept and the concept of enabling conditions.

4.8 Timeouts

The modelling of time is especially delicate in many formal models for distributed systems. Basically there are two extreme ways of modelling time dependent behaviour:

(1) An explicit notion of time is introduced into the semantic model. This can be done by introducing special messages (such as T representing ticks of the clock) or by using the concept of time stamps (cf. [Broy 83] or [Broy et al. 87]).

(2) One may model special time requirements (without introducing time notions explicitly) by certain liveness conditions that guarantee that certain actions will eventually happen. So a communicating module either receives eventually some input or produces eventually some output (as a result of an implicit timeout, cf. [Broy 87a]).

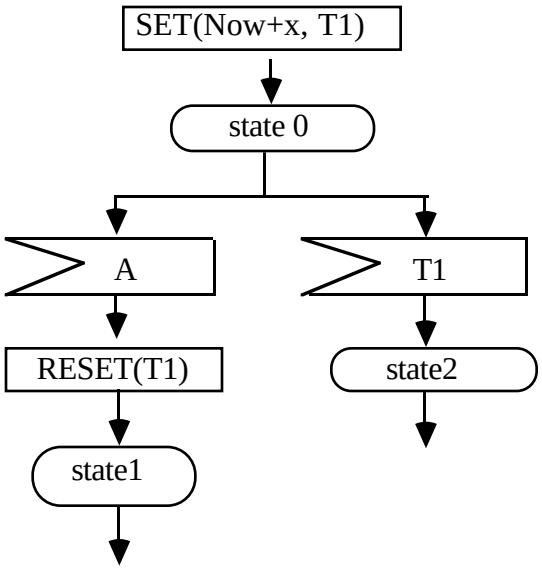
The approach (1) has the disadvantage that all parts of the system (even those for which time considerations are not important for explaining their functional behaviour) are polluted with time messages. However, even very time dependent behaviour can be modelled sufficiently.

The approach (2) has the disadvantage that strictly time dependent behaviour cannot be modelled sufficiently. Moreover, the theory of the functional model gets more complicated, since some problems with monotonicity may arise. However, all the parts of the system for which time considerations are not important for explaining their functional behaviour are not polluted with time messages.

In SDL a timeout is expressed in a process diagram by

- setting a time,
- setting an input signal.

Graphically this is expressed by the following diagram:



The meaning of the diagram can be interpreted in a "nondeterministic" way in our model with explicit representation of time ticks by the equations:

([state0].d).f ≡ ∀ s: ∃ e: TICKS.e ∧

(ft.e.s = T ⇒ ∃ h: f.s = T & h.rt.e.s ∧ ([state2].d).h) ∧

(ft.e.s ∈ SIGNALS.A ⇒ ∃ g: f.s = g.e.s ∧ ([state1].update(d, A, ft.s)).g) ∧

(¬(ft.e.s ∈ {T, ⊥} ∪ SIGNALS.A) ⇒ f.s = T & f.rt.e.s).

We model the setting of the timer by a predicate TICKS specifying functions that may consume an arbitrary but finite number of time tokens in the input stream before the timer is actually started. This is formally expressed by:

TICKS.f ≡ ∀ x: ∃ i: x = T^{i}^f.x

So by by the application f.x of a function f where TICKS.f holds we may express the situation that we may nondeterministically delete an arbitrary (but finite) number of elements T in the beginning of the input stream.

If we naively drop the time information to obtain a model without explicit timing we get a formula that may be fulfilled by several functions:

([state0].d).f ≡ ∀ s: ∃ h: (f.s = h.s ∧ ([state2].d).h) ∧

(ft.s ∈ SIGNALS.A ⇒ ∃ g: (f.s = g.rt.s ∧ ([state1].update(d, A, ft.s)).g) ∧

(¬(ft.s ∈ {⊥} ∪ SIGNALS.A) ⇒ f.s = f.rt.s)).

Note that with this specification the equation f.s = h.s can always be chosen, since without any assumption on the relative speed of the processes the timeout may always happen.

Note, however, that with this interpretation we run into difficulties with the monotonicity requirement for f. If x = ‹›, we obtain ¬(ft.x ∈ SIGNALS.A) and thus the equation boils down to

f.‹› = g.‹› .

This may exclude any behaviour specified by [state1], since even in cases where ft.x ∈ SIGNALS.A holds by the fact that ‹› is a prefix of every stream we have:

‹› ⊑ x

and by the monotonicity of f we obtain from the second part from the formula (*) above

g.‹› = f.‹› ⊑ f.x .

Therefore if ¬(g.‹› ⊑ f.rt.x) where ([state2].update(d, A, ft.x)).g, then the possibility f.x = g.rt.x where ([state1].update(d, A, ft.x)).g is logically excluded by the formula (∗) given above. This does not model the behaviour of SDL process diagrams appropriately. Therefore we weaken the equation (*) to the following equation:

([state0].d).f ≡ ∀ s:

(∃ h: f.s = h.s ∧ ([state2].d).h) ∨

((∃ g: ft.s ∈ SIGNALS.A ∧ f(A & s) = g.s ∧ ([state1].update(d, A, ft.s)).g) ∧

$\neg$(ft.s $\in$ {SIGNALS.A $\cup$ {$\perp$}}) $\Rightarrow$ f.s = f.rt.s)) .

The first part of the formula treats the case of a timeout. This equation, however, is a too liberal. This can be compensated by the introduction of an additional liveness predicate along the lines of [Broy 88]. For doing so we introduce the following predicate:

L: D $\rightarrow$ ([M^{ω} $\rightarrow$ N^{ω}] $\times$ M $\rightarrow$ $\mathbb{B}$)

which is given by the formula:

(L.d)(f, x) $\equiv$ ([state0].d).f $\wedge$ (SIGNALS.A)©x = ‹› $\Rightarrow$ $\exists$ h: f.x = h.rt^{i}.x $\wedge$ ([state2].d).h

The proposition L(f, x) expresses that the timeout finally occurs, if there is no input signal A. This way the nondeterministic choice of the function in an application may depend on the input in a nonmonotonic way.

The proposition L(f, x) thus has the following meaning: Due to the included nondeterminism there are several functions f that fulfil the predicate. When f is applied to some argument x one of these functions can be chosen. The proposition indicates that the choice has to be done in a way such that L(f, x) holds.

These properties can be very explicitly expressed in the functional programming language as described in [Broy 82]. We write

f.x = **if** (ft.x $\in$ SIGNALS.A) ∇ false **then** g.rt.x **else** h.x **fi**

Here ∇ denotes the so-called *ambiguity choice operator* that gives for (a ∇ b) the value a or b, but its result is defined (i.e. $\neq \perp$) as long as at least one of the values a or b is defined (i.e. $\neq \perp$).

We do not go deeper into this issue here. There are several possibilities to model time in SDL according to the different ways of formally modelling time in functional views of distributed systems. The more concrete choice of a specification technique for SDL diagrams with real time constraints depends very much on the overall goal of the formalisation of SDL.

4.9 Functionalities of Processes

According to what has been said so far we now may derive a uniform formal model for processes. A process may be associated with the following four sets:

- a set of input signals M,
- a set of output signals N,
- a set of local variable states D,
- a set of process states P.

For the process in addition we may assume an initial data state value d0 and an initial control state state0. Then the *external meaning* of a process diagram p is given by the predicate [p] with

$[p].f \equiv ([state0].d0).f$

This way the external meaning of a process is constructed from its internal meaning. Note that in general there are several functions that fulfil the specifications (i.e. equations) derived from a SDL process diagram, in particular, if nondeterminism is caused by timeouts. A proper treatment of the meaning of SDL process diagrams can be obtained along the lines of [Broy 88] by associating with a SDL process description a set of equations for the functions associated with each state. Then we may derive from the set of equations the set of functions that fulfil the safety properties and in addition we may derive a predicate that specifies the liveness properties in case of timeouts.

5. Compositional Forms

It is essential for a tractable modular system design and system description that systems can be composed from subsystems. It is essential for a compositional formal model that the meaning of a system can be constructed from the meanings of its subsystems.

5.1 Block diagrams

In SDL a system description can be composed from a number of process descriptions. The process descriptions are composed by block diagrams. For explaining how this can be modelled we just consider a specific scheme of composition.

Let B_i be blocks (given for instance by process diagrams) for $1 \leq i \leq n$ with an external meaning represented by the functions

$$p_i: (M_i^{\omega} \rightarrow N_i^{\omega}) \rightarrow \mathbb{B}$$

Since every signal is labelled by the process/channel where it came from, the sets M_i and N_i can be assumed to be pairwise disjoint.

Certain signals of a block diagram are sent to or from the environment of the block; these signals are called *external* signals. Other signals are sent between blocks being subblocks in the block diagram; these signals are called *internal* signals. Let M_0 denote the external input signals, i.e. the signals coming from the environment of the block and let N_0 denote the set of external output signals, i.e. the set of signals that are sent by the block to the environment.

Then the sets of signals exchanged by the system are defined by the union of the signals exchanged by the subsystems:

$$M = \cup \{ M_i : 0 \leq i \leq n \},$$

$$N = \cup \{ N_i : 0 \leq i \leq n \}.$$

The sets of actions performed in the system are defined by the following two sets:

$IN = \{ in.m: m \in M \}$

$OUT = \{ out.m: m \in N \}$

Note that M and N are not disjoint, in general. If subblock i sends signals to subblock j, then these signals occur both in N_i and M_j. We associate with the block diagram a predicate on stream processing functions

$p_0: (M_0^{\omega} \rightarrow N_0^{\omega}) \rightarrow \mathbb{B}$

as follows.

5.1.1 Trace-oriented Descriptions of Block Diagrams

We consider a block diagram with n blocks described by the predicates $p_1, ..., p_n$. We start by introducing the notion of a global trace. A *global trace* of the block described by the block diagram is a stream t of actions with:

$t \in (IN \cup OUT)^{\omega}$

which fulfils the property that if we filter out the resp. input and output actions we get equations fulfilled by the functions associated with the subblocks. We now give a formal definition of this requirement for a trace t.

Each communication of a signal can be understood as consisting of two actions: out.s denotes the action of sending the signal and in.s denotes the action of receiving a signal s.

A global trace t represents a history of all the internal and external actions performed within the block diagram during an execution. A global trace is the merge of all the local traces of the subblocks under the condition that an internal action in.s comes always after the resp. out.s. Technically this is expressed by the formula that ensures the causality between input and output (no signal can be received before it was sent):

$\forall\ p: p \sqsubseteq t \Rightarrow \#(\{in.s\} \copyright p) \leq \#(\{out.s\} \copyright p)$

A global trace is a stream t of actions such that there exist projections (disjoint subtraces) t_i, $1 \leq i \leq n$, where every local trace t_i is a proper trace for one of the external functions associated with process p_i. Technically we assume the existence of a stream $orac \in \{1,\ldots, n\}^{\omega}$ called the *global oracle* for which

$proj(t, orac, i) = t_i$

where

$proj(t, j \ \& \ orac, i) = ft.t \ \& \ proj(rt.t, orac, i)$ if $i = j$,

$proj(t, j \ \& \ orac, i) = proj(rt.t, orac, i)$ if $i \neq j$.

Intuitively speaking the global oracle indicates which action belongs to which subblock. Furthermore for all i, $0 \le i \le n$ we may assume streams $k.i \in \{1, 2\}^{\omega}$ called the *local oracles* such that

$proj(t_i, k.i, 1) = IN©t_i$,

$proj(t_i, k.i, 2) = OUT©t_i$.

Note that for every global trace the local and global oracles are defined uniquely, since all signals carry the process identifiers with them where they came from and their destination can be uniquely determined. As additional auxiliary functions we define the function drop

$drop: (IN \cup OUT)^{\omega} \rightarrow (IN \cup OUT)^{\omega}$

that gets rid of the in(.) and out(.) labels in a sequence of actions. It is formally defined by the equations

$drop.\langle\rangle = \langle\rangle$,

$drop(in.d \,\&\, x) = d \,\&\, drop.x$,

$drop(out.d \,\&\, x) = d \,\&\, drop.x$.

With the introduced functions we now may formulate the requirement for a stream t to be a global trace: Let f_i be an external function associated with process p_i. Given some input stream $x_0 \in M_0^{\omega}$ for the block diagrams t is a global trace, if $x_0 = M_0©drop(IN©t)$ and t is the least fixpoint of the system of equations

$t = schedule(t_1, \ldots, t_n, h)$

$t_i = schedule(IN©t_i, out^*(f_i(drop(IN©t_i))), k.i)$ for all i , $1 \le i \le n$,

where

$schedule(t_1, \ldots, t_n, i \,\&\, h) = ft(t_i) \,\&\, schedule(t_1, \ldots, t_{i-1}, rt(t_i), t_{i+1}, \ldots, t_n, h)$

and

$p_i.f_i$

and

$out^*(\langle\rangle) = \langle\rangle$,

$out^*(d \,\&\, x) = out.d \,\&\, out^*.x$.

Every local trace t_i can be decomposed such that

$drop(OUT©t_i) = f_i(drop(IN©t_i))$

With these definitions we obtain the equation for the external behaviour predicate p_0 associated with the block diagram:

$p_0.f_0 \equiv \forall\, x: f_0(x) = N_0©drop(OUT©t)$.

This definition based on the concept of traces looks rather technical. We may, however, give a more abstract definition for behavioural equations for f_0 by the following set of equations.

5.1.2 Functional Description of Block Diagrams

We may give a more simple description based on the predicates describing the functions associated with the blocks without looking at traces. We specify the predicate p_0 defining the behaviour of the block diagram as follows:

$$p_0.f_0 \equiv \forall\ x_0\colon f_0(x_0) = (N_0 © x)$$

where the x_i are defined for given x_0 by the least fixpoint of

$$x_i = f_i(M_i © \mathrm{merge}(x_0,..., x_n)) \qquad \text{where} \qquad P_i.f_i \qquad \text{for i, } 1 \le i \le n$$

Here merge is a continuous function that merges its argument streams in a fair way i.e. it fulfils the property

MERGE.merge

where

$$\mathrm{MERGE.merge} \equiv \forall\ x_0, ..., x_n\colon$$
$$\exists\ \mathrm{oracle}\colon \mathrm{oracle} \in \{0, ..., n+1\}^{\omega} \wedge \forall\ i\colon i \in \{0, ..., n+1\}\colon$$
$$\#\{i\} © x_i = \infty \wedge \mathrm{merge}(x_0,..., x_n) = \mathrm{sched}(x_0,..., x_n, \mathrm{oracle})$$

where $\forall\ i\colon i \in \{0, ..., n\}\colon$

$$\mathrm{sched}(x_0,..., x_n, i\ \&\ c) = \mathrm{ft}.x_i\ \&\ \mathrm{sched}(x_0,..., x_{i-1}, \mathrm{rt}.x_i, x_{i+1} ,..., x_n, c) \wedge$$
$$\mathrm{sched}(x_0,..., x_n, n+1\ \&\ c) = T\ \&\ \mathrm{sched}(x_0,..., x_n, c)\ .$$

The continuity of the functions merge fufilling the predicate MERGE.merge is straightforward, since sched is a continuous function. Again the system above can be very explicitly expressed in the functional programming language as described in [Broy 82] by a system of mutually recursive equations for streams.

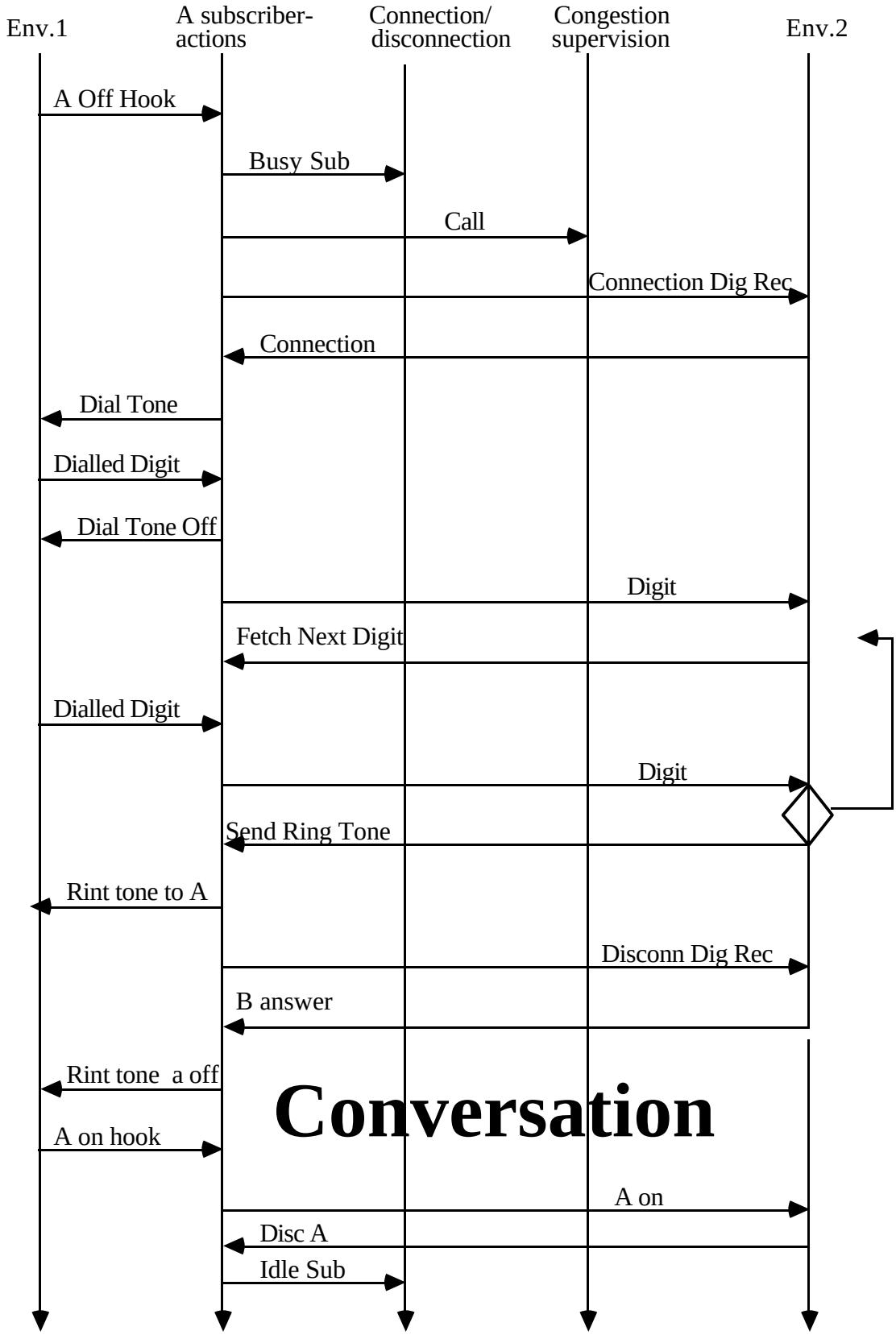
An interesting open question is whether the function merge should in fact be fair, which is equivalent to the question whether fairness assumptions are valid for SDL signal communications. This question is not answered by the SDL language description.

The definitions above show the close relationship of functional models of interactive systems with feedback to fixpoint theory. A function f_0 is the external function of a block diagram if every trace t_0 of f_0 fulfils the properties formulated for t_0 and f_0 above.

5.2 Sequence Charts

A sequence chart gives an example for the interaction between processes and their environment within a process diagram. In a sequence chart an example of a run of the system is given. Every subprocess and also the environment is considered as a sequential unit with a trace. Every such

trace is represented by an vertical line. Communications are represented by arrows between these lines. Every process is considered to be a sequential unit. The transmission of a signal consists of two actions for



Sequence chart. Service interaction in normal case.
Necessary signals for the A-subscriber party.

- generating the signal SIGNAL, represented by the action out.SIGNAL,

- consuming the signal SIGNAL, represented by the action in.SIGNAL.

Of course a signal can only be consumed after it had been received. Let all definitions be as in the preceding section.

Technically a sequence chart defines a global trace and some local traces which form a family of streams if we associate with every horizontal arrow in the chart labelled by signal SIGNAL two actions in.SIGNAL and out.SIGNAL. With every vertical line in the chart that corresponds to a process p we can then associate a trace of actions, called the *local trace* of process p. The local trace includes out.SIGNAL for outgoing edges and in.SIGNAL for incoming edges.

Then we can verify by the formulas given above whether the local trace of process p that we obtain from the sequence chart is actually a trace for the process diagram p.

However, in a sequence chart also the different traces are put into some relation. Technically a sequence chart defines a global trace t where the actions correspond to the signals exchanged between the agents and their environment.

5.3 The Overall Functional Modelling of SDL Diagrams

In SDL graphical system description basically the following kinds of diagrams occur:

- block tree diagrams show the decomposition of blocks into subblocks,
- system and block structure diagrams and substructure diagrams correspond to data flow diagrams,
- state overview diagrams,
- procedure diagrams,
- process diagrams give the transitions that lead from one state to the other,
- sequence charts that give examples of system behaviours.

All kinds of these diagrams can be translated into logical formulas for the functional model. We have just given some basic ideas and explanations how to do that, but it should indicate that such a translation is possible and appropriate and how the method works.

6. An Extended Example

In this section we treat an extended example of an SDL design as given in [CCITT Z.100] Annex D: User Guide-lines. It is part of the telephone service description.

6.1 Process Specification: Telephone service

We start by modelling the SERVICE A_subscriber_actions on page 168-170 of [CCITT Z.100] Annex D: User Guide-lines. We give a description of the A_subscriber_ actions.

As input signals we have the set:

M = { A_OFF_HOOK, CONNECTION, CONGESTION, A_ON_HOOK, DIALLED_DIGIT, FETCH_NEXT_DIGIT, SEND_RING_TONE, B_ANSWER, DISC_A }

As output signals we have the set:

N = { BUSY_SUB, CALL, CONNECT_DIG_REC, CONG_TONE, CONG_CALL, DIAL_TONE, DIAL_TONE_OFF, DIGIT, DISCONN_DIG_REC, RING_TONE_A_OFF, A_ON, IDLE_SUB, A_OFF, CONG_TONE_OFF }

The control states of the service are given by the set P:

P = { A_IDLE, AWAIT_A_ON_HOOK, AWAIT_CONN, AWAIT_FIRST_DIGIT, AWAIT_DIGIT, AWAIT_ANALYSIS, A_RINGING, AWAIT_DISC, CONVERSATION, AWAIT_A_ON_HOOK_2, IDLE_SUB }

We define the internal behaviour by the function f which has the following functionality:

$f: M^{\omega} \times D \times P \rightarrow N^{\omega}$

where we have for every control state s and data state d:

([s].d).g where $\forall$ x: g.x = f(x, s, d)

and f is specified by:

f(A_OFF_HOOK & x, d, A_IDLE) =
if connected.d **then** BUSY_SUB & CALL & CONNECT_DIG_REC & f(x, d, AWAIT_CONN)
else f(x, d, A_IDLE)
fi

$f(A\_ON\_HOOK^{k}\hat{}CONGESTION \& x, d, AWAIT\_CONN) =$
CONG_TONE & CONG_CALL & $f(A\_ON\_HOOK^{k}\hat{}x, d, AWAIT\_A\_ON\_HOOK)$

$f(A\_ON\_HOOK^{k}\hat{}CONNECTION \& x, d, AWAIT\_CONN) = DIAL\_TONE \& f(A\_ON\_HOOK^{k}\hat{}x, d, AWAIT\_FIRST\_DIGIT)$

f(A_ON_HOOK & x, d, AWAIT_FIRST_DIGIT) = DIAL_TONE_OFF & A(x, d)

f(DIALLED_DIGIT & x, d, AWAIT_FIRST_DIGIT) = DIAL_TONE_OFF & DIGIT & f(x, d, AWAIT_ANALYSIS)

$f(DIALLED\_DIGIT^{k} \& FETCH\_NEXT\_DIGIT \& x, d, AWAIT\_ANALYSIS) = f(x, d, AWAIT\_DIGIT)$

f(DIALED_DIGIT & x, d, AWAIT_DIGIT) = DIGIT & f(x, d, AWAIT_ANALYSIS)

$f(DIALLED\_DIGIT^{k} \& SEND\_RING\_TONE \& x, d, AWAIT\_ANALYSIS) =$
RING_TONE_TO_A & DISCONN_DIG_REC & f(x, d, A_RINGING)

f(B_ANSWER & x, d, A_RINGING) = RING_TONE_A_OFF & f(x, d, CONVERSATION)

f(A_ON_HOOK & x, d, A_RINGING) = RING_TONE_A_OFF & A_ON & IDLE_SUB & f(x, d, A_IDLE)

A(x, d) = DISCONN_DIC_REC & IDLE_SUB & f(x, d, A_IDLE)

f(A_ON_HOOK & x, d, CONVERSATION) = A_ON & f(x, d, AWAIT_DISC)

f(A_OFF_HOOK & x, d, AWAIT_DISC) = A_OFF & f(x, d, CONVERSATION)

f(DISC_A &x,d, AWAIT_DISC) = IDLE_SUB & f(x, d, A_IDLE)

f(DISC_A & x, d, CONVERSATION) = f(x, d, AWAIT_A_ON_HOOK)

f(A_ON_HOOK & x, d, AWAIT_A_ON_HOOK) = IDLE_SUB & f(x, d, A_IDLE)

f(A_ON_HOOK & x, d, AWAIT_A_ON_HOOK_2) = CONG_TONE_OFF & f(x, d, IDLE_SUB)

The translation of the SDL diagram into a system of equations for stream processing functions is completely schematic and can be done mechanically. Note that some of the equations could be simplified and some of the states could be eliminated. However, we tried to be as close as possible to the graphical representation.

We obtain the following equations for the function g that is taken as the external behaviour of the process A_subscriber_actions (let d_0: be the initial value of the variable d)

g: $M^{\omega} \rightarrow N^{\omega}$

with

g.x = f(x, d_0, A_IDLE)

Next we show for that example how sequence charts can be related to block diagrams and thus can be verified.

6.2 Sequence Charts for the Service Interaction

The sequence chart for the service interaction in [CCITT Z.100] Annex D User Guide-lines Fig. D-10.2.7 describes the service interaction in the normal case. It contains the necessary signals for the A_subcriber_actions. We obtain the subtrace t as a local trace for this service of the process SUBSCRIBER.LINE (Fig.D-10.2.4):

in.A_OFF_HOOK &
out.BUSY_SUB &
out.CALL &
out.CONNECT_DIG_REC &
in.CONNECTION &
out.DIAL_TONE &
in.DIALLED_DIGIT &
out.DIAL_TONE_OFF &
out.DIGIT &
in.FETCH_NEXT_DIGIT &
in.DIALLED_DIGIT &
out.DIGIT &
in.SEND_RING_TONE &
out.RING_TONE_TO_A &
out.DISCONN_DIG_REC &
in.B_ANSWER &
out.RING_TONE_OFF &
in.A_ON_HOOK &
out.A_ON &
in.DISC_A &
out.IDLE_SUB & ‹›

We now can verify our function representing the behaviour of the service A_subscriber_actions by showing that t is a local trace for it. We have to prove

trace(g, t) = true

with the function trace as specified in section 3.3. We obtain by applying the equations listed in the formal specification of the behaviour function for the service A_subscriber_actions (here for simple notation we identify the names of the control states with the resp. functions):

trace(g, t) =
trace(A_IDLE, in.A_OFF_HOOK& out.BUSY_SUB& out.CALL& out.CONNECT_DIG_REC & rt^4.t) =
trace(AWAIT_CONN, in.CONNECTION & out.DIAL_TONE & rt^6.t) =
trace(AWAIT_FIRST_DIGIT, in.DIALlED_DIGIT & out.DIAL_TONE_OFF & out.DIGIT & rt^9.t) =
trace(AWAIT_ANALYSIS, in.FETCH_NEXT_DIGIT & rt^{10}.t) =
trace(AWAIT_DIGIT, in.DIALLED_DIGIT & out.DIGIT & rt^{12}.t) =
trace(AWAIT_ANALYSIS, in.SEND_RING_TONE & out.RING_TONE_TO_A & out.DISCONN_DIG_REC & rt^{15}.t) =
trace(A_RINGING, in.B_ANSWER & out.RING_TONE_OFF & rt^{17}.t) =
trace(CONVERSATION, in.A_ON_HOOK & out.A_ON & rt^{19}.t) =
trace(AWAIT_DISC, in.DISC_A & out.IDLE_SUB & ‹›) =

trace(A_IDLE, ‹›) =

true

This is a very simple form of verification, but it nevertheless allows to show that SDL sequence charts are fulfilled by the process diagrams. Thus it allows to check some consistency for a SDL design.

However, there are many other possibilities for verification for SDL system descriptions after translating them to the functional model besides that simple ones such as

- verification of properties,
- generation of sequence charts from the process descriptions.

We will come back to these possibilities in the next chapter where we discuss further methods and tools that can be based on our formal model.

7. Formal Methods and Tools for SDL based on the Semantic Framework

In chapter 3, we introduced the formal foundation of functional system descriptions used in chapter 4 and 5 to build a semantic framework for SDL. In this chapter we discuss several possibilities for tools which could both support the semantic framework and build further capabilities upon it.

The formal foundation is based upon the concept of streams and stream processing functions. We have shown that sequence charts in a SDL network can be mapped onto streams and also that processes can be mapped onto the specific functions or more precisely sets of functions. Finally, we have shown how the SDL notion of system construction, based upon creating a network of interconnected components, where each component is derived from a specific process, can be represented in terms of compositions of processes, where each process is an instantiation of the corresponding function.

7.1 Formal Representation of Semantics

With all SDL graphical forms semantics can be associated in terms of functional models of distributed systems and these have been identified in the framework introduced in the previous chapter. In the traditional use of SDL, the specific requirements are defined with varying degrees of formality during the design derivation process and only become fully formally represented when a graphical form is actually expressed in programming language terms (assuming a fixed semantics for SDL programs). Given the formal framework which we have established in this study, it is now possible to improve upon this situation by suggesting that the functions (and hence the semantics) of each graphical form in SDL should be defined in terms of a functional language.

In order to ease the task of defining the function of each of the graphical forms, it is suggested that a special functional form of the SDL textual design representation is developed. This representation would be based upon the existing textual representations, as defined in the recommendations of SDL and also used in this report. The objective would be to make the best possible use of existing SDL notation and to construct the representation in such a way that a tool could make all the necessary transformations into the full formal framework. This would therefore be consistent with one of our initial objectives of supporting a formal way of using SDL without necessarily forcing designers or specifiers to be completely familiar with all the underlying mathematics (in the same way, for example, as the use of the Laplace transform in electrical engineering). The functional language for describing the semantics of SDL graphs is called FDL (Functional Description Language) in the following. FDL could be understood as a specification language along the lines of [Broy 88] or as a functional programming language along the lines of [Broy 82] for which an implementation is available by a graph reduction based term evaluating scheme.

We in particular get a very simple and basic notion of correctness for SDL process and block diagrams. Both diagrams are semantically connected to predicates

$$Q: (M^{\omega} \rightarrow N^{\omega}) \rightarrow \mathbb{B}$$

A process diagram or block diagram with behaviour predicate Q is called correct w.r.t. a specifying predicate R (which may be the behaviour predicate w.r.t. process diagram or block diagram), if:

$$Q \Rightarrow R$$

This way we get a very simple and tractable logic for verifying process diagrams or block diagrams.

7.2 Tools based upon the Textual Representation

In this section, we introduce some ideas for tools based upon a textual description of SDL in terms of FDL.

7.2.1 SDL Design Checker

The first tool that might be considered is a tool to design and store descriptions of SDL graphical forms and specifications and to check that they conform to standard SDL syntax. Such a tool is currently under development by Siemens and does not have any specific characteristics relating to this study. However, the tool being developed does store the resulting graphical forms, including the design interactions between them, in a database, and this information can be used as a basis for other tools described below. It is hoped that more advanced support tools can be obtained by extending the existing tool.

7.2.2 Transformer from SDL to Functional Descriptions and FDL Checker

Given that a database can be set up which describes an SDL design in terms of a set of interrelated graphical forms as indicated above, it is possible to produce a tool which could transform the information content of one or more SDL graphical forms into its equivalent FDL representation. The FDL specification would consist of a number of data type specifications, functionalities and equations. For process diagrams the transformation would be complete. For block diagrams without process diagrams the transformation would produce only the outline of the function, but not the details. The user could then be invited to edit the FDL to add the functional detail of so far unspecified subblocks.

Given that a user had produced an SDL diagram, perhaps using the tools described earlier, it would be necessary to check that the consistence of the semantics of SDL had been adhered to, within the functional descriptions. For instance it could be checked that the sequence charts actually are examples of possible runs. A tool to perform this check is quite straightforward provided, of course.

7.2.3 FDL Builder and FDL Evaluator Description

Once a database has been established containing all the functional descriptions induced by a particular SDL system, it is then possible to consider evaluating such a system. However, before it can be evaluated, it is necessary to build the complete system into a form suitable for evaluation. An FDL building tool could be produced to achieve this objective. It would be given the name of the SDL system module and then, by consulting the database, trace which processes were required, create the appropriate components from them and finally produce a single composite image ready for evaluation. In particular the evaluation could be based on the already mentioned implementation of a function system design language as described in [Broy 82] by a graph reduction term evaluation procedure.

There are two possibilities envisaged for the evaluator. Firstly, a basic version could be produced which merely operated upon a set of streams and produced the set of output streams determined by the set of functions contained in the system when connected together in the specific way determined by the system. Such an evaluator could be based on a diploma thesis finished at the University of Passau recently which gives a graph reduction implementation for a functional language that is powerful enough to include FDL.

A more sophisticated and, we hope, more useful version would, in addition to the facilities of the basic version, add the capability to determine several metrics about the operation of the system. Such metrics would be concerned with the 'operational' behaviour of the system. The sort of metrics we have in mind are the following:

i) total number of objects in each stream,
ii) total number of items of data passed through each channel,
iii) mean and peak rates of flow through each channel,
iv) deadlock etc.

Here there are many possibilities to consider special interactive support tools for SDL/FDL designs.

Another possibility is given by the RAP system as being developed at the University of Passau. Since SDL graphical forms are basically translated into equations in FDL the system RAP can be applied, in principle. It can be used for prototyping and simulating SDL designs.

7.2.4 Graphics Based Tools

Given the correspondence between the textual and graphical forms of SDL notation, a tool can be developed in a relatively straightforward manner to convert a SDL design from one form to the other, and to check the correspondence between the two forms of the design. The graphical part of this tool could be developed from the existing tool at Siemens.

Further possible tools, which might be not completely formal with respect to semantic properties, but nevertheless could be of considerable assistance to the user, could be mixtures of graphical formalisms like SDL, which allow the description of the overall structure of a system, and logical formalisms like special versions of specifications of stream processing functions, that allow the description of the behaviour of the various components of a system.

More advanced tools include design support tools that provide a number of support functions for top-down and bottom-up design of SDL diagrams and also for stepwise transformation of those designs.

7.3 Algebraic and Logical Reasoning

Using only the functional stream processing concept it is rather difficult to do any formal reasoning on the system within this model. Here more diversified formalisms have to be developed, if the stream model is to be the basis for formal reasoning. Special forms of logic, perhaps including forms of modal logic and temporal logic, have to be considered. While these forms of logic increase the complexity of the foundational model, they are likely to make the form of reasoning easier (although without increasing the total reasoning power available). These forms of logic could be incorporated into a verification and reasoning tool.

The most advanced design support system could provide an integrated framework for graphical representations of SDL designs, corresponding specifications in terms of logical formulae in terms of FDL declarations, reasoning support tools and design (decomposition, composition and transformation) support tools.

In particular notations from [Broy 88] can be considered for deriving a specification language for formulating requirements for SDL components. These requirements can be formulated for processes as well as for general blocks (and blocks formed by block diagrams). The requirements can be formulated at the level of traces (as a logical counterpart to sequence charts) or at the level of stream processing functions.

Often it is from a methodological point of view appropriate to distinguish between *safety* and *liveness* properties of system components. For our example of the process A_subscribe_actions we may formulate the safety property that a signal CALL is only send if a signal A_OFF_HOOK was received before. This can be expressed by the following formula on the local trace t:

$$t' \sqsubseteq t \Rightarrow \#(\{out(\text{CALL})\} \copyright t'\} \leq \#(\{in(\text{A\_OFF\_HOOK})\} \copyright t')$$

or by the equivalent formula for the external function f associated with the process diagram

$$\#(\{\text{CALL}\} \copyright f.x) \leq \#(\{\text{A\_OFF\_HOOK}\} \copyright x)$$

A typical liveness property is given by the following requirement: If the first input signal is A_OFF_HOOK and the first output signal is BUSY_SUB then the second output signal is CALL. This is expressed by the trace assertion for trace t:

$$ft.t = in(\text{A\_OFF\_HOOK}) \wedge ft(rt.t) = out(\text{BUSY\_SUB}) \Rightarrow ft(rt^2.t) = out(\text{CALL})$$

or on the functional level

$$ft.x = \text{A\_OFF\_HOOK} \wedge ft.f.x = \text{BUSY\_SUB} \Rightarrow ft.rt.f.x = \text{CALL}$$

This is of course only a very simple example that is very close to the defining equations. However, much more sophisticated specifications can be written by using the introduced formalism. Examples are provided in [Broy 88].

7.4 Towards a Development Method for SDL

Two basic requirements for a 'rigourous' development method for SDL system descriptions are as follows:

(i) a formal concept of the equivalence between representations, and

(ii) calculable criteria for the evaluation of particular representations.

Requirement (i) is sketched in the current study. Requirement (ii) may be met by the proposed specification formalism. We believe that a true development method can be produced to support SDL.

Here it is appropriate to mention a further aspect. Obviously the specifications for a stream processing function derived from a SDL diagram often do not cover all cases. For instance in our example of the A_subscriber_actions process diagram many cases are not covered. For instance nothing is said about the output behaviour of the considered process if it gets DIALLED_DIGIT as first signal. Thus nothing is specified about the stream produced by:

f(DIALLED_DIGIT & x, d, A–IDLE) .

Obviously such an input of signal DIALLED_DIGIT should not occur in the initial state A_IDLE of the considered process. However, due to failure in the environment those signals might arrive and it is part of a proper design of reliable systems to consider and cover those exceptional cases, too.

8. Conclusions

In this section we draw a number of conclusions from the study.

8.1 Experiences and Evaluation of the Study from an Academic Point of View

From an academic point of view the questions in the study in which one could be interested were concerned with the suitability of the described methods for practical applications. SDL is a rather pragmatic approach to the design of distributed reactive systems with limited real-time aspects. In contrast, functional descriptions of interactive systems are a very academic, theory-oriented vehicle for the study of the semantic structure of reactive systems. However, the functional description of interactive systems proved to be surprisingly well suited to our discussion of the SDL formalism. In addition, the experience of explaining the formal machinery of denotational semantics to people coming from the area of practical applications proved to be less difficult than expected in studies like [Broy et al. 87]. Nevertheless, it is probably here that the most critical point of the study can be found. Although not requiring deep mathematics the denotational functional approach does not seem to be in accordance with the classical views of people dealing with distributed systems.

During the case study it became very clear that the functional view of distributed systems is regarded as unusual by members of the SDL user community, and it brings a number of

conceptual difficulties for those who are used to thinking in terms of imperative and operational state-oriented presentations. The functional view of reactive systems seems to be in contrast to the widely used operational state-oriented view. Although the two views are completely consistent and either view can be transformed into the other, people from the area of practical applications generally seem to have problems in adopting the functional view. The reason is quite simple: they are used to thinking about systems in an operational style and the state-oriented view corresponds to an operational style. Functional views of distributed systems are non-operational, descriptive views which allow one to talk about properties of systems and their behaviour without going into the question of how, operationally, a system is computing. It is not clear at the moment what to conclude from this. One conclusion could be that one should try to obtain a more imperative notation for the stream-processing framework. The other possibility is that it is just a matter of learning and that people should learn the more functional view of concurrent systems. It may not be easy, however, to convince people of the necessity of learning about functional views.

Certainly there are many possibilities for building more tools and methods over and above the attempt to join a rather formal academic approach to distributed reactive systems like stream processing functions and the much more pragmatic application-oriented approach like SDL. However, the first and most interesting observation is that it is at all possible in principle to combine these extreme and very different views. In that respect we consider the study as completely successful.

It is not surprising that the result of this study does not lead immediately to a set of tools which can be used by the designer. Much more work has to be done to come closer to this goal. However, a lot of interesting questions, and also many new aspects and possibilities for investigation have been discovered during the study.

From an academical point of view a number of valuable insights have been obtained, leading to the following conclusions:

1. The denotational framework of stream processing functions is well suited to discuss properties of pragmatic formalisms like SDL.

2. The formalism can be explained to people working in practical applications.

3. The formal framework of denotational semantics has to be explained very carefully in terms of the intuitive understanding of people with practical experience.

4. The relationship between the functional model (provided by the denotational semantics) and the intuitive understanding of distributed systems (in terms of state-oriented views) has to be properly explained to those people who really want to specify their system in terms of stream processing functions.

5. A more finely tuned notation for stream processing functions should be developed, closer to the needs of SDL systems.

These conclusions indicate some areas where further research might be considered and where it will be useful.

In this study SDL was chosen as the specification formalism for which a semantic model was sketched in terms of functional system models. It is an interesting question whether the technique of functional system modelling can also be used for similar formalisms such as Estelle (cf. [Budkowski, Dembinski 87]) or LOTOS (cf. [Brinksma et al. 87]). For SDL the modelling by stream processing functions was especially well-suited, since it is based on asynchronous communication concepts. For languages based on synchronous communication concepts ("handshaking") the modelling is technically more complicated, since the very specific communication protocols have to be made explicit in the semantics, but nevertheless I assume that a semantic model in terms of functions can be given.

8.2 Short Evaluation of the Study

Firstly, we have successfully demonstrated the feasibility of describing the semantics of SDL in terms of stream functions processing. However, at present, the problems connected with timeouts need further inspection.

We also noted that the functional description of a subsystem prior to its decomposition into simpler components may be described either as a single function or as a composition of a number of other functions. These two alternatives could represent different stages in the development lifecycle.

It is clear that there are many possible ways to provide a quite powerful environment for the specification, validation, design and verification of systems, the structure of which could be represented by SDL diagrams. However, it is also quite clear that such a framework needs a lot of further work to be done.

The formal description of SDL leads also to a number question that concern design issues for SDL. A particular issue for SDL is nondeterminism and the use of time notions. Both notions are included in SDL in a rather half-hearted way. Nondeterminism arises implicitly in SDL via the scheduling of messages in communication block diagrams and also via timeouts. Since explicit nondeterminism is not included certain nondeterministic behaviours can only be represented very implicitly.

Similar remarks apply for the incorporation of time notions. There is a notion of starting a timer and of a timeout suggesting a quantitative treatment of time. But since nothing is assumed about the relative speed of system components, a quantitative reasoning about time is not possible in SDL.

Acknowledgement

This study was supported by Siemens ZTI SOF. It is a pleasure to thank Dr. Klugmann and Dr. Schmidt for their support and A. Dietl, J. Grabowski, and E. Rudolph for helpful discussions. The work has derived benefit from the study [Broy et al. 87]. I like to thank K. Jackson and R. Pennington for numerous discussions.

References

[Bate 86]
D. G. Bate: Mascot 3 - an introductory tutorial. Software Engineering Journal **1**, 3, pp. 95-102 (1986)

[Brauer 80]
W. Brauer (ed.): Net theory and applications. Lecture Notes in Computer Science **84**, Berlin-Heidelberg-New York-Tokyo: Springer 1980

[Brinksma et al. 87]
E. Brinksma, G. Scollo, C.A. Vissers: Experience and future of LOTOS as a specification language. SDL '87: State of the Art, North Holland 1987

[Brock, Ackermann 81]
J. D. Brock, W. B. Ackermann: Scenarios: A Model of Nondeterminate Computation. In: J. Diaz, I. Ramos(eds): Lecture Notes in Computer Science **107**, Springer 1981, 252-259

[Broy 82]
M. Broy: A theory for nondeterminism, parallelism, communication and concurrency. Habilitation, Fakultät für Mathematik und Informatik der Technischen Universität München, 1982, Revised version in Theoretical Computer Science **45** (1986) 1-61

[Broy 83]
M. Broy: Applicative real time programming. Information Processing 83, IFIP World Congress, Paris 1983, North Holland Publ. Company 1983, 259-264

[Broy 84a]
M. Broy: Denotational semantics of concurrent programs with shared memory. In: M. Fontet, K. Mehlhorn (eds.): STACS 84. Lecture Notes in Computer Science **182**, Berlin-Heidelberg-New York-Tokyo: Springer 1984, 163-173

[Broy 84b]
M. Broy: Semantics of communicating processes. Information & Control **61**:3 (1984), 202-246

[Broy 85a]
M. Broy: Extensional behaviour of concurrent, nondeterministic, communicating systems. In: M. Broy (ed.): Control flow and data flow: Concepts of Distributed Programming, Springer NATO ASI Series.

[Broy 85b]
M. Broy: Specification and top down design of distributed systems (invited talk). In: H. Ehrig et al. (eds.): Formal Methods and Software Development. Lecture Notes in Computer Science 186, Springer 1985, 4-28, Revised version in JCSS **34**:2/3, 1987, 236-264

[Broy 86]
M. Broy: On modularity in programming. In H. Zemanek (ed.): A quarter century of IFIP, North Holland Publ. 1986, 347-362.

[Broy 87a]
M. Broy: Semantics of finite or infinite networks of communicating agents. Distributed Computing **2** (1987), 13-31

[Broy 87b]
M. Broy: Predicative specification for functional programs describing communicating networks. Information Processing Letters **25** (1987) 93-101

[Broy 87c]
M. Broy: Views of Queues. Technische Berichte der Fakultät für Mathematik und Informatik, Universität Passau, 1987, MIP-8704, also in Science of Computer Programming

[Broy 87e]
M. Broy: Algebraic and functional specification of a serializable database interface. Technische Berichte der Fakultät für Mathematik und Informatik, Universität Passau, 1987, MIP-8718

[Broy et al. 87]
M. Broy, K. Jackson, R. Pennington: A Stream Function Definition of MASCOT. System Designers, Software Technology Centre, Final Technical Report 1987

[Broy 88]
M. Broy: Towards a design methodology for distributed systems. International Summer School on Constructive Methods in Computing Science, Marktoberdorf 1988

[Budkowski, Dembinski 87]
S. Budkowski, P. Dembinski: An introduction to Estelle. Computer Networks 14:1, 1987

[CCITT Z.100]
CCITT: Specification and Design Language SDL. Blue Book

[Hehner 84]
E.C.R. Hehner: Predicative Programming. Part I+II. CACM **27**:2 (1984) 134-151

[Hoare 78]
C.A.R. Hoare: Communicating sequential processes, Comm. ACM **21** (8) (1978) 666-667.

[Hoare et al. 81]
C.A.R: Hoare, S.D. Brookes and A.W. Roscoe: A theory of communicating sequential processes.

Oxford University Computing Laboratory, Programming Research Group, Technical Monograph PRG-21, Oxford (1981). Also in: J. ACM **31** (1984) 560-599

[Hoare 85]
C.A.R. Hoare: Communicating Sequential Processes. Prentice Hall 1985

[Jackson 86]
K. Jackson: Mascot 3 and Ada. Software Engineering Journal **1**, 3, pp. 121-135 (1986)

[Kahn, MacQueen 77]
G. Kahn and D. MacQueen, Coroutines and networks of processes, Proc. IFIP Congress 1977,

[MASCOT 85]
JIMCOM: The Official Handbook of Mascot. RSRE, St. Andrews Road, Malvern, Worcs. (1985)

[Mazurkiewicz 85]
A. Mazurkiewicz: Traces, histories, graphs: instances of a process monoid. In: M.P. Chytil, V. Koubek (eds.): MFCS 1984, Lecture Notes in Computer Science **92**, Berlin-Heidelberg-New York-Tokyo: Springer 1985, 115-133

[MacQueen 79]
D.B. MacQueen, Models for distributed computing, IRIA RR No. 351 (1979)

[Milner 80]
R. Milner: A calculus for communicating systems, Lecture Notes in Computer Science **92**, Berlin-Heidelberg-New York-Tokyo: Springer 1980

[Nielsen et al. 81]
M. Nielsen, G. Plotkin, G. Winskel: Petri nets, event structures, and domains. Part 1. Theoretical Computer Science **13**, 1981, 85-108

[Olderog, Hoare 82]
E.-R. Olderog, C.A.R. Hoare: Specification-oriented semantics for communicating processes. In: Diaz: International Colloquium on Automata, Languages and Programming 83, Lecture Notes in Computer Sciences, Berlin-Heidelberg-New York: Springer 1983

[Park 80]
D. Park: On the semantics of fair parallelism. In: D. Björner (ed.): Abstract Software Specification. Lecture Notes in Computer Science **86**, Berlin-Heidelberg-New York: Springer 1980, 504-526

[Rozenberg 85]
G. Rozenberg: Advances in Petri-nets. Lecture Notes in Computer Science **188**, Berlin-Heidelberg-New York-Tokyo: Springer 1985

[Simpson 86]

H. R. Simpson: The Mascot method. Software Engineering Journal **1**, 3, pp. 103-120 (1986)

[Winkowski 80]
J. Winkowski: Behaviors of concurrent systems. Theoretical Computer Science **11**, 1980, 39-60.