An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus[*]

Eckhardt Holz, Ketil Stølen

Abstract

This paper presents a first attempt to embed a restricted version of SDL as a target language in Focus. Brief introductions to both Focus and SDL are given, and it is shown how both methods can be assigned a denotational semantics based on streams and stream processing functions. A set of Focus specifications, referred to as F-SDL, is characterized whose elements structurally and semantically match SDL specifications to such a degree that an automatic translation is almost straightforward. Finally it is outlined how Focus can be used to develop an SDL specification of a protocol.

1 Introduction

Focus [BDD$^+$92], [BS94] is a methodology, in the tradition of [Kah74], [Kel78] for the formal specification and development of distributed systems. A system is modeled by a network of components working concurrently and communicating asynchronously via unbounded FIFO channels. A number of reasoning styles and techniques is supported. Focus provides mathematical formalisms which support the formulation of highly abstract, not necessarily executable specifications with a clear semantics. Moreover, Focus offers powerful refinement calculi which allow distributed systems to be developed in the same style as for example VDM and Z allow for the development of sequential programs. Finally, Focus is modular in the meaning that design decisions can be checked at the point where they are taken, that component specifications can be developed in isolation, and that already completed developments can be reused in new program developments.

SDL [CCI93] has been developed by CCITT and was initially intended for the description of telecommunication systems. However, SDL is also well-suited for more general specification tasks. In SDL the behavior of a system is equal to the combined behavior of its processes. A process is basically a communicating, extended, finite-state machine. The processes communicate asynchronously by sending signals via signal routes. SDL provides both a textual and a graphical specification formalism. SDL has received considerable interest from industry and is supported by a large number of tools and environments.

In some sense Focus and SDL are orthogonal approaches. Because of its very mathematical and abstract notation, Focus has its strength in the area of formal refinement and

[*]This work is supported by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen".

verification. SDL, on the other hand, due to its graphical notation and many structuring constructs, is very well-suited for the formulation of large and complicated "real-life" specifications. It is therefore tempting to try to combine these two approaches into one methodology inheriting the strength of both. This is our motivation!

SDL offers a large number of specification and structuring constructs, and it is important to realize that it is not our intention to transform Focus into a method, which allows for the development of any SDL specification. In fact we are only interested in the sublanguage of SDL, which in a natural way corresponds to Focus developments. For example, Focus is not well-suited for the description of dynamic networks — networks where processes can be created and interfaces may change during execution. Thus the full generality of the SDL process creation mechanism is not very relevant in connection with a Focus development. This does not mean that we consider these additional features of SDL to be of little value. On the contrary, we rather see Focus as a tool or facility, which can be used to formally develop and verify certain restricted, critical parts of an SDL specification. Typical examples would be communication protocols, mutual exclusion algorithms or some complicated sorting algorithm.

Section 2 describes the underlying formalism. Then Focus and SDL are introduced in Sections 3 and 4, respectively. It is shown how both Focus and a restricted version of SDL can be assigned the same type of denotational semantics. In Section 5 F-SDL is syntactically characterized, and Section 6 outlines how an SDL specification of a protocol can be formally developed employing the proposed technique. Finally, Section 7 gives a brief summary and discusses possible extensions.

2 Underlying Formalism

$\mathbb{N}$ denotes the set of positive natural numbers. We assume the availability of the standard logical operators. As usual, $\Rightarrow$ binds weaker than $\wedge, \vee, \neg$ which again bind weaker than all other operators and function symbols.

A stream is a finite or infinite sequence of actions. It models the history of a communication channel by representing the sequence of messages sent along the channel. Given a set of actions D, D^* denotes the set of all finite streams generated from D; D^∞ denotes the set of all infinite streams generated from D, and D^ω denotes $D^* \cup D^\infty$.

Let $d \in D$, $r, s \in D^\omega$, and j be a natural number, then:

- ϵ denotes the empty stream;
- $\#r$ denotes the length of r, which is equal to ∞ if r is infinite, and is equal to the number of elements in r otherwise;
- $r|_j$ denotes the prefix of r of length j if $j < \#r$, and r otherwise;
- $d \,\&\, s$ denotes the result of appending d to s;
- $r \frown s$ denotes r if r is infinite and the result of concatenating r with s, otherwise;
- $r \sqsubseteq s$ holds if r is a prefix of s.

The stream operators defined above are overloaded to tuples of streams in a straightforward way. ϵ will also be used to denote tuples of empty streams when the size of the tuple is clear from the context. If d is an n-tuple of actions, and r, s are n-tuples of streams, then $\#r$ denotes the length of the shortest stream in r; $d \,\&\, s$ denotes the result of applying $\&$ pointwisely to the components of d and s; $r \frown s$ and $r \sqsubseteq s$ are generalized in the same pointwise way.

A chain c is an infinite sequence of stream tuples $c_1, c_2, \ldots$ such that for all $j \geq 1$, $c_j \sqsubseteq c_{j+1}$. $\sqcup c$ denotes c's least upper bound. Since streams may be infinite such least upper bounds always exist.

A Boolean function $P : (D^\omega)^n \rightarrow \mathsf{B}$ is admissible iff whenever it yields true for each element of a chain, then it yields true for the least upper bound of the chain. P is prefix-closed iff whenever it yields true for a stream tuple, then it also yields true for any prefix of this stream tuple. P is safe iff it is admissible and prefix-closed. We write $\mathsf{safe}(P)$ iff P is safe.

A function $\tau \in (D^\omega)^n \rightarrow (D^\omega)^m$ is called a ((n,m)-ary) stream processing function iff it is prefix monotonic and continuous:

$$\text{for stream tuples } i \text{ and } i' \text{ in } (D^\omega)^n : i \sqsubseteq i' \Rightarrow \tau(i) \sqsubseteq \tau(i'),$$

$$\text{for all chains } c \text{ generated from } (D^\omega)^n : \ \tau(\sqcup c) = \sqcup\{\tau(c_j) | j \in \mathsf{N}\}.$$

That a function is prefix monotonic implies that if the input is increased then the output may at most be increased. Thus what has already been output can never be removed later on. Prefix continuity, on the other hand, implies that the function's behavior for infinite inputs is completely determined by its behavior for finite inputs.

A stream processing function $\tau \in (D^\omega)^n \rightarrow (D^\omega)^m$ is pulse-driven iff:

$$\text{for all stream tuples } i \text{ in } (D^\omega)^n : \ \#i \neq \infty \Rightarrow \#i < \#\tau(i).$$

That a function is pulse-driven means that the length of the shortest output stream is infinite or greater than the shortest input stream. This property is interesting in the context of feedback constructs because it guarantees that the least fixpoint is always infinite for infinite input streams.

The arrows $\rightarrow$, $\overset{c}{\rightarrow}$ and $\overset{cp}{\rightarrow}$ are used to tag domains of ordinary functions, domains of monotonic, continuous functions, and domains of monotonic, continuous, pulse-driven, functions, respectively.

To model time-outs we need a special action $\surd$, called "tick". There are several ways to interpret streams with ticks. In this paper, all actions should be understood to represent the same time interval — the least observable time unit. $\surd$ occurs in a stream whenever no ordinary message is sent within a time unit. A stream or a stream tuple with occurrences of $\surd$'s is said to be timed. Similarly, a stream processing function is said to be timed when it operates on domains of timed streams. Observe that in the case of a timed, pulse-driven, stream processing function the output during the first $n+1$ time intervals is completely determined by the input during the first n time intervals. For any stream or stream tuple i, $\diamond i$ denotes the result of removing all occurrences of $\surd$ in i.

In the more theoretical parts of this paper, to avoid unnecessary complications, we distinguish between only two sets of actions, namely the set D denoting the set of all actions minus $\surd$, and $D_{\surd}$ denoting $D \cup \{\surd\}$. However, the proposed formalism can easily be generalized to deal with more general sorting, and this is exploited in the examples.

We use two additional functions in our examples: a projection function ⓒ and a function $\propto$ which eliminates repetitions. More explicitly, if A is a set of n-tuples of actions, d, e are n-tuples of actions, and r is an n-tuple of streams, then Aⓒ and $\propto$ are stream processing functions such that the following axioms hold:

$$A \copyright \epsilon = \epsilon,$$
$$d \in A \Rightarrow A \copyright d \,\&\, r = d \,\&\, A \copyright r,$$
$$d \notin A \Rightarrow A \copyright d \,\&\, r = A \copyright r,$$

$$\propto(\epsilon) = \epsilon,$$
$$\propto(\langle d \rangle) = \langle d \rangle,$$
$$d = e \Rightarrow \propto(d \,\&\, e \,\&\, r) = \propto(d \,\&\, r),$$
$$d \neq e \Rightarrow \propto(d \,\&\, e \,\&\, r) = d \,\&\, \propto(e \,\&\, r).$$

Note that these axioms together with the monotonicity and continuity constraints determine the semantics also for stream tuples whose stream components are not of the same length. For example $\propto(a \,\&\, b, \epsilon) = \epsilon$. When $A = \{d\}$ we write $d \copyright r$ instead of $\{d\} \copyright r$.

3 Focus and Its Stream Semantics

Depending upon the logical concepts they employ, Focus specifications can be divided into a number of subclasses. For example, Focus distinguishes between trace specifications, equational specifications, functional specifications, assumption/commitment specifications, state-oriented specifications, relational specifications, etc. — each of these alternatives having special problem areas or stages in a system development where they are particularly suited. The use of tables and diagrams is also supported.

In this paper, we employ only so-called relational specifications. However, the proposed approach can easily be combined with the other specification techniques in Focus. A relational specification of a component with n input channels and m output channels is written in the form

$$S(i_1 \in D^\omega, \ldots, i_n \in D^\omega \rhd o_1 \in D^\omega, \ldots, o_m \in D^\omega) \equiv R,$$

where S is the specification's name; $i_1, \ldots, i_n$ and $o_1, \ldots, o_m$ are disjoint, repetition free lists of identifiers representing n respectively m streams; R is a formula with the elements of $i_1, \ldots, i_n$ and $o_1, \ldots, o_m$ as its only free variables. Each stream models the communication history of a channel, and R characterizes the allowed relation between the histories of the input and the output channels. We therefore refer to R as the input/output relation. For any specification S, R_S represents its input/output relation.

In Focus specifications are modeled by sets of timed, pulse-driven, stream processing functions. In real-time specifications the ticks occur also at the syntactic level. In other specifications they are abstracted away in the sense that they are not allowed to occur explicitly in the specifications. In this paper we consider only specifications of the latter type.

The denotation of the specification S is the set of all (n,m)-ary, timed, pulse-driven, stream processing functions which fulfill R when time-signals are abstracted away and only complete inputs are considered:

$$[\![S]\!] \overset{\mathsf{def}}{=} \{\tau \in (D_{\surd}^{\omega})^n \overset{cp}{\to} (D_{\surd}^{\omega})^m \,|\, \forall r \in (D_{\surd}^{\infty})^n : (\diamond r, \diamond\tau(r)) \models R\},$$

where $(\diamond r, \diamond\tau(r)) \models R$ holds iff R evaluates to true when each input identifier i_j is interpreted as the j'th element of the n-tuple $\diamond r$, and each output identifier o_j is interpreted as the j'th element of the m-tuple $\diamond\tau(r)$.

Due to the time abstraction, at the syntactic level the streams are untimed. At the semantic level the time-ticks allow complete input histories (infinite inputs) to be distinguished from partial input histories (finite inputs). The additional expressiveness resulting from this makes it possible to specify fair merge components. Such components cannot be modeled by sets of untimed stream processing functions [Kel78]. See [BS94] for a more detailed discussion.

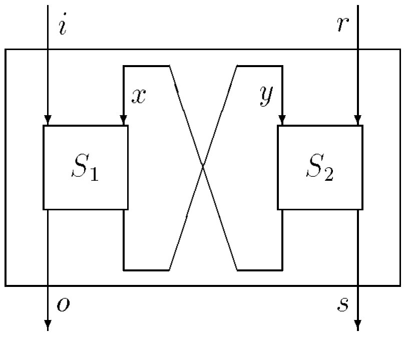

Figure 1: Network Consisting of the Specifications S_1 and S_2.

Networks of specifications are expressed in an equational style. For example, the network S pictured in Figure 1, consisting of the the two specifications S_1 and S_2, is characterized as below:

$$S(i \in D^{\omega}, r \in D^{\omega} \rhd o \in D^{\omega}, s \in D^{\omega}) \equiv (o,y) = S_1(i,x),\ (x,s) = S_2(y,r)$$

The channels represented by x and y are now hidden in the sense that they represent local channels. The comma separating the two equations can be read as an "and".

More formally, $[\![S]\!] \overset{\mathsf{def}}{=} \{\tau_1 \otimes \tau_2 \,|\, \tau_1 \in [\![S_1]\!] \wedge \tau_2 \in [\![S_2]\!]\}$, where for any pair of timed stream tuples i and r, $(\tau_1 \otimes \tau_2)(i,r) \overset{\mathsf{def}}{=} (o,s)$ iff (o,s) is the least fixpoint solution with respect to i and r. This is logically expressed by the following formula:

$$\begin{aligned}
&\exists x, y \in D_{\surd}^{\omega} : \forall o', y', x', s' \in D_{\surd}^{\omega} : \\
&\qquad \tau_1(i,x) = (o,y) \wedge \tau_2(y,r) = (x,s)\ \wedge \qquad (1)\\
&\qquad \tau_1(i,x') = (o',y') \wedge \tau_2(y',r) = (x',s') \Rightarrow (o,y,x,s) \sqsubseteq (o',y',x',s'). \qquad (2)
\end{aligned}$$

(1) requires (o, y, x, s) to represent a fixpoint; (2) requires this fixpoint to be the least.

In fact any dataflow network can be expressed in this equational style. We have already seen an example of a finite network. Infinite networks are expressed using recursion. For a more detailed syntactic and semantic treatment, see [Ded92].

Focus offers a number of refinement concepts with corresponding refinement calculi. The most basic of these is behavioral refinement, which at the semantic level corresponds to set inclusion: a specification S_2 refines a specification S_1, written $S_1 \rightsquigarrow S_2$, iff the denotation of S_2 is equal to or contained in the denotation of S_1. More formally: $S_1 \rightsquigarrow S_2$ iff $[\![S_2]\!] \subseteq [\![S_1]\!]$.

The refinement relation $\rightsquigarrow$ is reflexive, transitive and a congruence with respect to the composition operators. Hence, $\rightsquigarrow$ allows compositional system development: once a specification is decomposed into a network of subspecifications, each of these subspecifications can be further refined in isolation.

Refinement calculi for relational specifications can be found in [SDW93], [BS94]. In the former the relational specifications are written in a so-called assumption/commitment style. See [Bro93] for an overview of refinement concepts supported by Focus.

4 SDL and Its Stream Semantics

An SDL specification defines a system behavior in a stimulus/response fashion, assuming that both stimuli and response are discrete and carry information. The specification model is based on the concept of communicating, extended, finite-state machines. SDL provides structuring concepts which facilitate the specification of large and/or complex systems. Specifications can be expressed both in a textual and a graphical formalism.

An SDL system specification is a container for a set of blocks. It is separated from its environment by a system boundary. The blocks are connected to one another and to the system environment by channels. Each communication between blocks respectively between blocks and the environment takes place using signals, which are conveyed by the channels. The transmission of signals can be delaying or non-delaying and uni- or bidirectional. A block can be a container for a set of blocks (block substructure) or it can be a container for a set of processes.

Processes are interconnected by non-delaying signalroutes. Signalroutes are also used to connect processes to the block boundary. A process definition defines a set of processes. Several instances of the same process set may exist concurrently and execute asynchronously and in parallel with each other and with instances of other processes in the system. A process instance is a communicating, finite-state machine extended to allow for:

- a secondary state, represented by local variables, in addition to the ordinary control state,
- explicit nondeterminism in terms of spontaneous input and nondeterministic decision,
- deferred consumption of input signals.

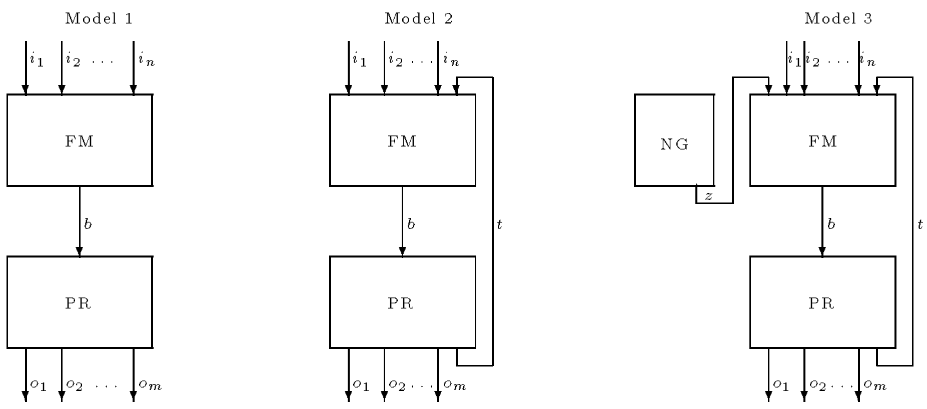

Figure 2: Three Models of an SDL Process.

Each process has an internal, unbounded buffer in which all incoming signals are inserted in the order of their arrival and thereafter processed. Simultaneously arriving signals are arbitrarily ordered. The set of valid state-transitions is described by a process graph or a service decomposition.

SDL offers many other facilities and structuring concepts. For example, the latest version, SDL92, is object-oriented. It is beyond the scope of this paper to give a more detailed description.

There is a close relationship between a functional core of the SDL language and lazy functional programming languages based on stream communication. It is this functional core that interests us in this paper. Inspired by [Bro91], we sketch how this restricted version of SDL can be given a denotational semantics in terms of streams and stream processing functions. Since Focus is based upon such a semantics this can be achieved by specifying the different SDL constructs in Focus. We consider only the time independent part of SDL. This means that all channels are declared as non-delaying and that only some restricted aspects of the SDL facilities for timers are modeled. Moreover, with the exception of SDL92's statements for explicit nondeterminism, all the constructs considered by us are contained in what [BHS91] calls Basic-SDL.

The semantics of SDL systems and blocks can easily be expressed as finite Focus networks given that the behavior of SDL processes can be described in Focus. This is explained in more detail when we later characterize F-SDL. In this section we concentrate on the modeling of the SDL process construct.

As indicated by Model 1 in Figure 2, if we ignore facilities for process creation, the timer constructs, explicit nondeterminism and that an SDL process can send signals to itself, a Basic-SDL process, with n input signal routes and m output signal routes, can be modeled as the sequential composition of two components, namely a fair merge component FM, which merges the streams of signals received on the n input signal routes into a stream b modeling the internal, unbounded buffer of an SDL process, and a processing component PR, which carries out the actual processing.

The component FM is characterized by the following relational Focus specification:

$$\mathrm{FM}(i_1 \in D^\omega, \ldots, i_n \in D^\omega \rhd b \in D^\omega) \equiv \exists o \in \{1, \ldots, n\}^\omega : \wedge_{j=1}^{n} split_j(b, o) = i_j.$$

Based on an oracle (its second argument) the auxiliary function $split_j$ extracts (from its first argument) the stream of signals received on the j'th input signal route — mathematically expressed:

$$j = y \Rightarrow split_j(x \,\&\, b, y \,\&\, p) = x \,\&\, split_j(b, p),$$

$$j \neq y \Rightarrow split_j(x \,\&\, b, y \,\&\, p) = split_j(b, p).$$

FM is a typical example of a non-executable Focus specification — non-executable in the sense that the input/output relation is characterized without giving any algorithm for its realization. In some sense the fair merge component is specified in terms of its inverse. Clearly, if the process has only one input channel, the fair merge component is not needed.

When we later define F-SDL, FM is assumed to be a specification constant with exactly the above characterized semantics. Moreover, FM is overloaded to deal with any number of input channels of any signal sorts[1].

In an SDL process specification the fair merge component FM is hidden in the sense that it is only a part of the semantics of the process. After all, since it is always the case that all incoming input signals are passed on to the internal buffer of the process, this does not have to be stated explicitly at the syntactic level. The visible part of an SDL process specification is basically a (possibly nondeterministic) functional program corresponding to the component PR.

As explained in [Bro91], with respect to the internal, unbounded buffer, the behavior of a deterministic SDL processing component can be modeled by a function

$$g \in D^q \to D^\omega \xrightarrow{c} (D^\omega)^m,$$

which for any q-tuple of secondary state variables l, returns a stream processing function $g(l)$, which characterizes the behavior of the processing component PR. This means that in the deterministic case the behavior of the component PR can be characterized by a relational Focus specification of the following form:

$$\mathrm{PR}(b \in D^\omega \rhd o_1 \in D^\omega, \ldots, o_m \in D^\omega) \equiv$$

$$\exists l_1, \ldots, l_q \in D : \exists g \in D^q \to D^\omega \xrightarrow{c} (D^\omega)^m : g(l_1, \ldots, l_q)(b) = (o_1, \ldots, o_m) \text{ where } Q$$

The variables $l_1, \ldots, l_q$ represents the secondary state of an SDL process. The existentially quantified function variable g (actually $g(l_1, \ldots, l_q)$) models the behavior of the processing component. The formula Q gives the actual definition of g. Section 5 explains in more detail how Q can be expressed.

[1]Such a specification constant can for example be expressed using polymorphic, object-dependent types.

Based on this, we may define the semantics of a simple SDL process as $[\![\text{ SDL\_PROC }]\!]$, where

$$\text{SDL\_PROC}(i_1 \in D^\omega, \ldots, i_n \in D^\omega \rhd o_1 \in D^\omega, \ldots, o_m \in D^\omega) \equiv$$

$$\begin{aligned}&(b) = \text{FM}(i_1, \ldots, i_n),\\ &(o_1, \ldots, o_m) = \text{PR}(b)\end{aligned}$$

As already mentioned, in this paper we are only interested in the time independent part of SDL — time-independent in the sense that all channels are declared as nondelaying. Nevertheless, SDL timers are needed to allow certain weakly time dependent components to be expressed. An example of such a component is the sender specified in Section 6. To allow for the specification of such components, we extend our restricted SDL language with a set-timer command of the following form: $\mathsf{set}(\mathsf{now}, timer(n))$. The first parameter is fixed as now.

Since the first parameter is fixed as now, according to the SDL semantics the time-out signals are placed in the unbounded, internal buffer in the same order as they are sent. Moreover, they are fairly interleaved with the signals received on the other input channels. Clearly, since we are only interested in the time-independent part of SDL, we do not need reset signals, nor the SDL constructs for checking whether a timer is active or idle.

As indicated by Model 2 in Figure 2, under these restrictions we may model an SDL-process with timers by adding an additional feedback channel t, which allows the processing component PR to send its timer signals back to FM. The latter merges the stream of timer signals with the other streams of input signals in the same way as before.

Unfortunately, as someone familiar with SDL may have observed, a problem has been brushed under the carpet. In SDL a timer signal remains active until it is consumed by the processing component PR. When the processing component sends a timer signal for which there is already an active copy in b, then the already active copy is deleted at the very same moment as the new copy is placed in b. Thus to make sure that we get the intended effect when our Focus specifications are translated into SDL, this problem must somehow be taken into consideration. There are at least two straightforward solutions.

The first alternative is to handle it directly in the specification of PR, namely by for each timer $timer(n)$, to add an additional parameter $m_{timer(n)}$ keeping track of the difference between the number of times $timer(n)$ has been output along t, and the number of times $timer(n)$ has been input from b. Then, whenever a timer signal $timer(n)$ is input from b, if $m_{timer(n)} > 1$, this signal is ignored — otherwise it is processed in the usual SDL way.

Given that T is the set of all timers, then the second alternative is to impose an additional proof-obligation which must be satisfied by the function

$$g \in D^q \to (D \cup T)^\omega \overset{c}{\to} (D^\omega)^m \times T^\omega$$

characterizing the behavior of the processing component PR. More explicitly, to require that

$$g(l_1,\ldots,l_q)(b) = (o_1,\ldots,o_m,t) \Rightarrow \#timer(n)\copyright t \leq \#timer(n)\copyright b + 1 \qquad (*)$$

for all timer signals $timer(n) \in T$. Note that g is monotonic and continuous.

Clearly, the first alternative allows us to model a larger class of SDL-specifications. However, the additional expressiveness we then get is not particular interesting from a pragmatic point of view. Moreover, it is expensive in the sense that our Focus specifications become more complicated. For this reason we decide in favor of the second alternative. Thus an SDL-process, which behaves in accordance with the additional proof obligation $(*)$, can be modeled by the set of timed, pulse-driven, stream processing functions characterized by $[\![$ SDL_PROC $]\!]$, where

$$\text{SDL\_PROC}(i_1 \in D^\omega,\ldots,i_n \in D^\omega \rhd o_1 \in D^\omega,\ldots,o_m \in D^\omega) \equiv$$

$$(b) = \text{FM}(i_1,\ldots,i_n,t),$$
$$(o_1,\ldots,o_m,t) = \text{PR}(b)$$

This format can of course easily be generalized to also allow the process to send ordinary signals to itself. It is enough to define t to be of type $(D \cup T)^\omega$.

So-far we have considered deterministic processing components only. However, in SDL92 nondeterminism can be expressed explicitly using the constructs for spontaneous input and nondeterministic decision. We now extend our semantic model to deal with these two constructs.

Spontaneous input is in SDL specified using an input symbol with the keyword none. A spontaneous input attached to a state means that the actual transition can be initiated at any time nondeterministically. This construct can for example be used to model unreliable behavior.

To handle spontaneous input, we add an additional component NG to our model, as indicated by Model 3 in Figure 2. The component NG is supposed to output nondeterministically some stream of none's, and is specified by:

$$\text{NG}(\ \rhd z \in \{\textsf{none}\}^\omega) \equiv \textsf{true}$$

The fair merge component must now also take the input from NG into consideration when it generates the stream modeling the internal unbounded buffer b — the signals received along z are of course treated as ordinary input signals.

As a consequence, the denotation of an SDL process with timers (given the stated restrictions) and spontaneous input is characterized by $[\![$ SDL_PROC $]\!]$, where

$$\text{SDL\_PROC}(i_1 \in D^\omega, \ldots, i_n \in D^\omega \rhd o_1 \in D^\omega, \ldots, o_m \in D^\omega) \equiv$$

$$\begin{aligned}
&(z) = \text{NG},\\
&(b) = \text{FM}(z, i_1, \ldots, i_n, t),\\
&(o_1, \ldots, o_m, t) = \text{PR}(b)
\end{aligned}$$

The other way of expressing nondeterminism explicitly in SDL — so-called nondeterministic decision — is represented by the keyword **any** inside a decision symbol with no values attached to its outlets. This indicates that the alternative chosen by the process cannot be forecast. To build this into our model we introduce a prophecy-variable p in the specification of PR:

$$\text{PR}(b \in (D \cup T)^\omega \rhd o_1 \in D^\omega, \ldots, o_m \in D^\omega, t \in T^\omega) \equiv$$

$$\begin{aligned}
&\exists p \in \mathsf{N}^\infty : \exists l_1, \ldots, l_q \in D : \exists g \in \mathsf{N}^\infty \to D^q \to (D \cup T)^\omega \xrightarrow{c} (D^\omega)^m \times T^\omega :\\
&\quad g(p)(l_1, \ldots, l_q)(b) = (o_1, \ldots, o_m, t) \ \textsf{where}\ Q
\end{aligned}$$

The existentially quantified p represents some infinite stream of positive natural numbers and can be thought of as a seed. If the k'th nondeterministic decision executed by the process has 5 outlets, then outlet $(p[k]\ \textsf{mod}\ 5) + 1$ is chosen where $p[k]$ denotes the k'th natural number in p.

5 The Syntax of F-SDL

So-far we have concentrated on mapping the syntactic expressions of the two formalisms into the very same semantics. In this section we work in the opposite direction. Based on the proposed SDL semantics expressed in Focus, a language called F-SDL is defined. This language characterizes a set of Focus specifications whose elements structurally and semantically matches SDL specifications to such a degree that an automatic translation into SDL is almost straightforward.

This section outlines the context independent syntax of F-SDL (we write "outlines" because the objective of this paper is to explain the central ideas — syntactic details will first be fixed in connection with the implementation of the F-SDL-to-SDL translator). The presentation of the syntax is interleaved with a number of examples showing the relationship to SDL. There is of course a large number of additional context dependent constraints. Since many of those are either obvious or correspond straightforwardly to similar constraints in SDL, only the most important are mentioned below. No translation algorithm is given. However, the examples indicate how the translation should be conducted.

We use the following EBNF convention: **[**expr**]** — expr is optional; $\{\text{expr}\}^*$ — zero or more repetitions of expr; $\{\text{expr//sep}\}^*$ — zero or more repetitions of expr separated by sep; $\{\text{expr}\}^+$ — one or more repetitions of expr; $\{\text{expr//sep}\}^+$ — one or more repetitions of expr separated by sep; $\text{expr}_1 \| \text{expr}_2$ — choice; ( ) — grouping.

We start by explaining how SDL system specifications are expressed in F-SDL:

$\langle\textsf{sys\_spec}\rangle$ ::= system $\langle\textsf{head}\rangle$ [$\langle\textsf{data\_def}\rangle$] $\langle\textsf{sys\_body}\rangle$ where $\langle\textsf{sys\_defs}\rangle$ end

$\langle\textsf{head}\rangle$ ::= $\langle\textsf{id}\rangle$ ($\{ \langle\textsf{ch\_decl}\rangle //, \}^*$ $\rhd$ $\{ \langle\textsf{ch\_decl}\rangle //, \}^*$) $\equiv$

$\langle\textsf{data\_def}\rangle$ ::= SDL_DATA_DEF

$\langle\textsf{sys\_body}\rangle$ ::= $\{ \langle\textsf{equation}\rangle //, \}^+$

$\langle\textsf{sys\_spec}\rangle$ characterizes a finite Focus network of blocks (i.e. as a set of equations) whose definitions are given in $\langle\textsf{sys\_defs}\rangle$. $\langle\textsf{data\_def}\rangle$ is used to define new datatypes, signals etc. as in SDL. Strictly speaking, the keywords system, where and end have no influence on the stream semantics. They have been included to simplify the implementation of the translation algorithm and to increase the readability of the specifications.

$\langle\textsf{ch\_decl}\rangle$ ::= $\{ \langle\textsf{ch\_id}\rangle //, \}^+ \in \langle\textsf{sig\_set}\rangle^{\omega}$

$\langle\textsf{equation}\rangle$ ::= $\langle\textsf{left\_side}\rangle = \langle\textsf{right\_side}\rangle$

$\langle\textsf{left\_side}\rangle$ ::= $\langle\textsf{ch\_tuple}\rangle$

$\langle\textsf{ch\_tuple}\rangle$ ::= ($\{ \langle\textsf{ch\_id}\rangle //, \}^*$)

$\langle\textsf{right\_side}\rangle$ ::= $\langle\textsf{id}\rangle$ $\langle\textsf{ch\_tuple}\rangle$

$\langle\textsf{sys\_defs}\rangle$ ::= $\{ \langle\textsf{block\_spec}\rangle //, \}^+$

Additional Constraints: Any block identifier occurring in $\langle\textsf{sys\_body}\rangle$ must also be defined in $\langle\textsf{sys\_defs}\rangle$. A system specification is required to satisfy a number of constraints with respect to its channel identifiers:

- the lists of input and output identifiers are disjoint and without repetitions;
- the same identifier occurs in maximum one $\langle\textsf{left\_side}\rangle$ and not more than once;
- the same identifier occurs in maximum one $\langle\textsf{right\_side}\rangle$ and not more than once;
- an input identifier cannot occur in a $\langle\textsf{left\_side}\rangle$;
- an output identifier cannot occur in a $\langle\textsf{right\_side}\rangle$;
- an identifier, which is no input identifier and occurs in a $\langle\textsf{right\_side}\rangle$, must also occur in a $\langle\textsf{left\_side}\rangle$.

Example 1 System Specification:
The SDL system diagram in Figure 3 corresponds to the following F-SDL specification:

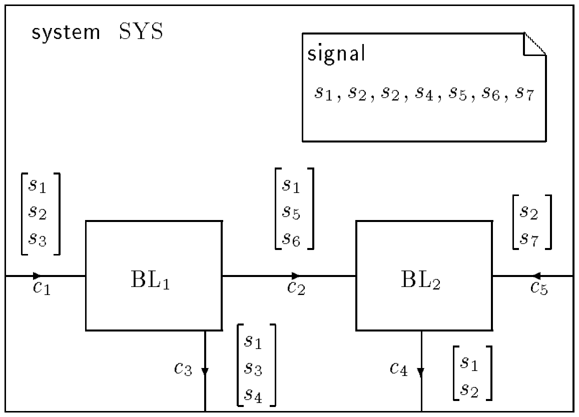

Figure 3: SDL System Diagram

system $\mathrm{SYS}(c_1 \in \{s_1, s_2, s_3\}^\omega, c_5 \in \{s_2, s_7\}^\omega \rhd c_3 \in \{s_1, s_3, s_4\}^\omega, c_4 \in \{s_1, s_2\}^\omega) \equiv$

$(c_3, c_2) = \mathrm{BL}_1(c_1),$
$(c_4) = \mathrm{BL}_2(c_2, c_5)$

where

block BL_1 ... end,

block BL_2 ... end

end

The type of the internal channel c_2 can be deduced from the specifications of the blocks.
□

F-SDL block specifications are expressed in almost the same way as systems:

⟨block_spec⟩ ::= block ⟨head⟩ **[** ⟨data_def⟩ **]** ⟨block_body⟩ where ⟨block_defs⟩ end

⟨block_body⟩ ::= **[** **{** ⟨alias⟩ //,**}**$^+$, **]** **{** ⟨equation⟩ //,**}**$^+$

⟨alias⟩ ::= ⟨ch_id⟩ = ⟨ch_id⟩

⟨block_defs⟩ ::= **{** ⟨block_spec⟩ //,**}**$^+$ **|** **{** ⟨proc_spec⟩ //,**}**$^+$

⟨proc_spec⟩ ::= ⟨ord_proc_spec⟩ **|** ⟨rec_proc_spec⟩

⟨block_body⟩ differs from ⟨sys_body⟩ in that in addition to the equations characterizing a finite network there are also equations modeling the renaming of channels/signal-routes

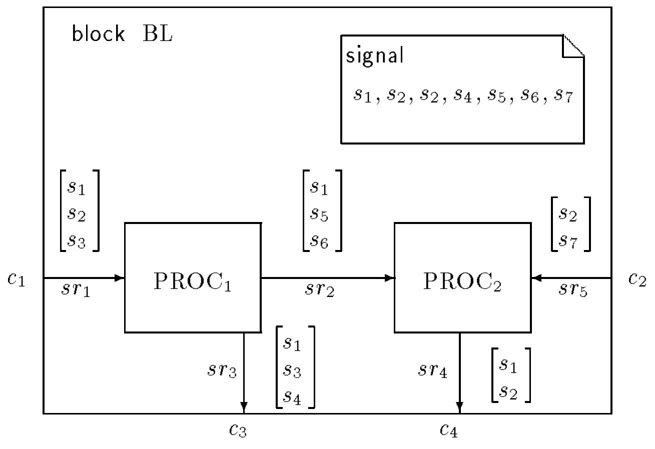

Figure 4: SDL Block Diagram

that can be necessary on the borderline between an SDL block and its surrounding system.

By ordinary processes ⟨ord_proc_spec⟩ we characterize the SDL processes which do not create other processes. A recursive process ⟨rec_proc_spec⟩, on the other hand, has process creation and communication as its only responsibility. The reason why we make this distinction is that in F-SDL only some rather restricted aspects of the SDL process creation facilities are modeled, namely those needed to express infinite Focus networks.

Clearly, a block or a process identifier occurring in ⟨block_body⟩ must also be defined in ⟨block_defs⟩. Moreover, we have the same constraints on channel identifiers as for F-SDL system specifications, with the additional requirements that the aliasing is conducted in accordance with standard SDL rules.

Example 2 Block Specification:
The SDL block diagram of Figure 4 corresponds to the following F-SDL specification:

block $\mathrm{BL}(c_1 \in \{s_1, s_2, s_3\}^\omega, c_2 \in \{s_2, s_7\}^\omega \rhd c_3 \in \{s_1, s_3, s_4\}^\omega, c_4 \in \{s_1, s_2\}^\omega) \equiv$

$sr_1 = c_1,$
$sr_5 = c_2,$
$c_3 = sr_3,$
$c_4 = sr_4,$
$(sr_3, sr_2) = \mathrm{PROC}_1(sr_1),$
$(sr_4) = \mathrm{PROC}_2(sr_2, sr_5)$

where

ord_process PROC_1 ... end,

ord_process PROC_2 ... end

end

□

We now characterize the syntactic structure of ordinary processes.

⟨ord_proc_spec⟩ ::= ord_process ⟨head⟩ **[** ⟨data_def⟩ **]** ⟨proc_body⟩ where ⟨pr_spec⟩ end

⟨proc_body⟩ ::= **[** ⟨ng_eq⟩ , **]** **[** ⟨fm_eq⟩ , **]** ⟨pr_eq⟩

⟨ng_eq⟩ ::= ⟨ch_unit_tuple⟩ = NG

⟨ch_unit_tuple⟩ ::= (⟨ch_id⟩)

⟨fm_eq⟩ ::= ⟨ch_unit_tuple⟩ = FM ⟨ch_tuple⟩

⟨pr_eq⟩ ::= ⟨ch_tuple⟩ = PR **[** ⟨ch_unit_tuple⟩ **]**

⟨ng_eq⟩, ⟨fm_eq⟩ and ⟨pr_eq⟩ model respectively the NG-equation, the FM-equation and the PR-equation, as explained on Page 11. NG and FM are specification constants and do not have to be explicitly defined in F-SDL. ⟨pr_spec⟩ gives the F-SDL specification of the processing component PR.

The finite network characterized by an ordinary process specification is required to satisfy the very same constraints with respect to stream identifiers as system and block specifications. In addition the different components are required to be connected as explained on Page 11.

Example 3 Ordinary Process Specification:
An F-SDL process specification is for example of the following form:

ord_process $\mathrm{PROC}(sr_1 \in \{s_1, s_2\}^\omega, sr_2 \in \{s_3\}^\omega \rhd sr_3 \in \{s_1, s_4\}^\omega, sr_4 \in \{s_1, s_2\}^\omega) \equiv$

$(z) = \mathrm{NG},$
$(b) = \mathrm{FM}(z, sr_1, sr_2, t),$
$(sr_3, sr_4, t) = \mathrm{PR}(b)$

where

PR ... end

end

In an SDL process specification, the structure characterized above is to a large extent hidden in its semantics. What we see in an SDL process diagram is basically the specification of the processing component PR. Note that the types of z, b and t can be deduced from the declarations in the PR specification.

□

The processing component PR represents the visible part of an SDL process specification. Its syntactic structure minus the state transitions is fixed below:

⟨pr_spec⟩ ::= ⟨pr_head⟩ ⟨pr_body⟩ where ⟨proc_graph⟩ end

⟨pr_head⟩ ::= PR (**[** ⟨ch_decl⟩ **]** $\rhd$ **{** ⟨ch_decl⟩ //,**}***) $\equiv$

⟨pr_body⟩ ::= **[** ⟨proph_decl⟩ **]** **{** ⟨loc_st_decl⟩ //,**}*** ⟨func_decl⟩ ⟨func_call⟩ = ⟨ch_tuple⟩

⟨proph_decl⟩ ::= $\exists$ ⟨proph_id⟩ $\in \mathsf{N}^\infty$:

⟨loc_st_decl⟩ ::= $\exists$ **{** ⟨loc_st_id⟩ //,**}**$^+$ $\in$ ⟨loc_st_type⟩ :

⟨func_decl⟩ ::= $\exists$ ⟨func_id⟩ $\in$ ⟨func_type⟩ :

⟨func_call⟩ ::= ⟨func_id⟩ **[** (⟨proph_id⟩) **]** **[** ⟨loc_st_tuple⟩ **]** **[** (⟨ch_id⟩) **]**

⟨loc_st_tuple⟩ ::= (**{** ⟨loc_st_id⟩ //,**}**$^+$)

Note the close correspondance between the structure of ⟨pr_body⟩ and the corresponding fragment of the specification PR on Page 11. The process graph ⟨proc_graph⟩ (the formula Q on Page 11) is basically a functional program expressed in terms of pattern matching.

Example 4 Processing Component Specification:
The SDL process diagram of Figure 5 corresponds to the F-SDL specification of Example 3, where PR is specified as below:

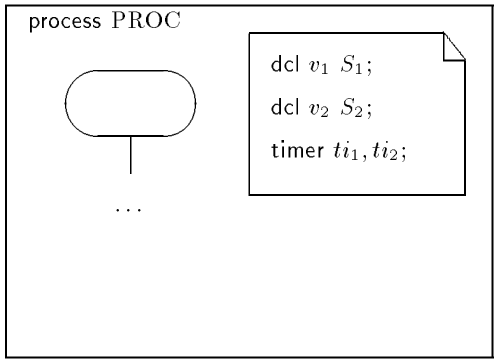

Figure 5: Process Specification in SDL

$$\mathrm{PR}(b \in \{s_1, s_2, s_3, ti_1, ti_2\}^\omega \triangleright sr_3 \in \{s_1, s_4\}^\omega, sr_4 \in \{s_1, s_2\}^\omega, t \in \{ti_1, ti_2\}^\omega) \equiv$$

$$\begin{aligned} &\exists p \in \mathsf{N}^\infty : \exists v_1 \in S_1 : \exists v_2 \in S_2 : \\ &\quad \exists g \in \mathsf{N}^\infty \to S_1 \times S_2 \to \{s_1, s_2, s_3, ti_1, ti_2\}^\omega \xrightarrow{c} \{s_1, s_4\}^\omega \times \{s_1, s_2\}^\omega \times \{ti_1, ti_2\}^\omega : \\ &\qquad g(p)(v_1, v_2)(b) = (sr_3, sr_4, t) \end{aligned}$$

where

$\vdots$

end

□

$\langle$proc_graph$\rangle$ ::= **[** $\langle$var_decls$\rangle$ **]** $\langle$i_tran$\rangle$ **[**, **{** $\langle$tran$\rangle$ //,**}**$^+$ **]**

$\langle$var_decls$\rangle$ allows us to introduce universally quantified variables needed for the pattern matching. $\langle$i_tran$\rangle$ is used to model the initialization transition — the transition from the start symbol to the first control state. $\langle$tran$\rangle$ models any other state transitions of an SDL process graph. $\langle$itran$\rangle$ is just a special case of $\langle$tran$\rangle$, namely a transition which does not consume input signals. Hence, its syntactic definition follows straightforwardly from the definition of $\langle$tran$\rangle$. Examples of complete F-SDL process graphs can be found in Section 6.

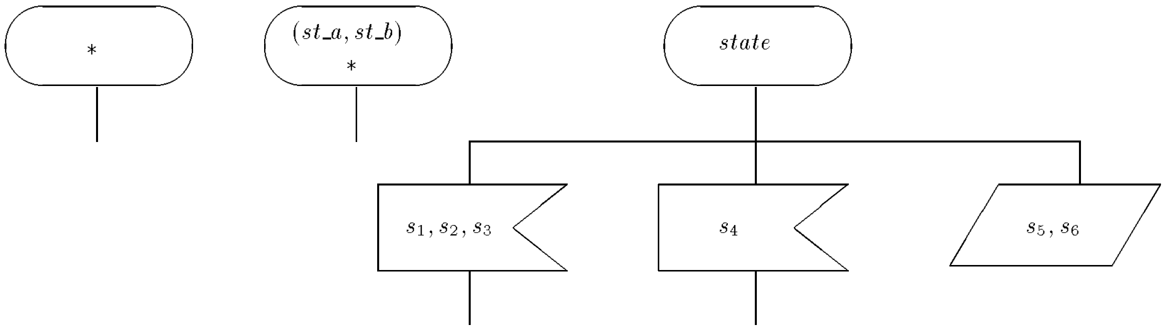

Figure 6: Three State-Transition Fragments

$\langle$tran$\rangle$ ::= **[** $\langle$func_id$\rangle$ $\notin$ $\langle$func_set$\rangle$ $\Rightarrow$ **]** ($\langle$h_tran$\rangle$ **|** $\langle$v_tran$\rangle$)

$\langle$h_tran$\rangle$ ::= $\langle$sig_id$\rangle$ $\notin$ $\langle$sig_set$\rangle$ $\Rightarrow$ $\langle$h_left$\rangle$ = $\langle$h_right$\rangle$

$\langle$h_left$\rangle$ ::= $\langle$func_id$\rangle$ **[** ($\langle$proph_id$\rangle$) **]** **[** $\langle$loc_st_tuple$\rangle$ **]** **[** ($\langle$signal$\rangle$ & $\langle$ch_id$\rangle$) **]**

$\langle$h_right$\rangle$ ::= $\langle$func_call$\rangle$

$\langle$v_tran$\rangle$ ::= $\langle$s_tran$\rangle$ **|** $\langle$o_tran$\rangle$

The optional part of $\langle$tran$\rangle$ is used to model an SDL transition from an asterisk state (see Example 5). A hidden transition $\langle$h_tran$\rangle$ models the way an SDL process consumes input signals for which there is no input- or save symbol (see Example 6). A visible transition $\langle$v_tran$\rangle$ models any other transition, namely a save transition $\langle$s_tran$\rangle$ or an ordinary state transition $\langle$o_tran$\rangle$.

Example 5 Asterisk State Transition:
The two F-SDL fragments

$$st\_var \notin \{\} \Rightarrow st\_var(\ldots) = \ \ldots,$$

$$st\_var \notin \{st\_a, st\_b\} \Rightarrow st\_var(\ldots) = \ \ldots.$$

correspond to the two first state-transition fragments of Figure 6. The first transition can be performed in any state; the second can be performed in any state different from *st_a* and *st_b*. *st_var* is a function variable, and *st_a* and *st_b* are functions representing SDL states *st_a* and *st_b*, respectively. Thus in the F-SDL process graph, we basically define one function for each SDL state we want to model.
□

$\langle \mathsf{s\_tran} \rangle$ ::= $\langle \mathsf{sig\_id} \rangle \notin \langle \mathsf{sig\_set} \rangle \wedge \langle \mathsf{sig\_set} \rangle \copyright \langle \mathsf{ch\_id} \rangle = \langle \mathsf{ch\_id} \rangle \Rightarrow \langle \mathsf{s\_left} \rangle = \langle \mathsf{s\_right} \rangle$

$\langle \mathsf{s\_left} \rangle$ ::= $\langle \mathsf{func\_id} \rangle$ **[** ($\langle \mathsf{proph\_id} \rangle$) **]** **[** $\langle \mathsf{loc\_st\_tuple} \rangle$ **]** ($\langle \mathsf{ch\_id} \rangle \frown (\langle \mathsf{sig\_id} \rangle \;\&\; \langle \mathsf{ch\_id} \rangle))$

$\langle \mathsf{s\_right} \rangle$ ::= $\langle \mathsf{func\_id} \rangle$ **[** ($\langle \mathsf{proph\_id} \rangle$) **]** **[** $\langle \mathsf{loc\_st\_tuple} \rangle$ **]** (($\langle \mathsf{sig\_id} \rangle \;\&\; \langle \mathsf{ch\_id} \rangle) \frown \langle \mathsf{ch\_id} \rangle)$

$\langle \mathsf{o\_tran} \rangle$ ::= **[** $\langle \mathsf{sig\_id} \rangle \in \langle \mathsf{sig\_set} \rangle \Rightarrow$ **]** $\langle \mathsf{left} \rangle = \langle \mathsf{right} \rangle$

$\langle \mathsf{left} \rangle$ ::= $\langle \mathsf{func\_id} \rangle$ **[** ($\langle \mathsf{proph\_pat} \rangle$) **]** **[** $\langle \mathsf{loc\_st\_tuple} \rangle$ **]** ($\langle \mathsf{sig\_pat} \rangle$)

$\langle \mathsf{sig\_pat} \rangle$ ::= **[** $\langle \mathsf{signal} \rangle \;\&$ **]** $\langle \mathsf{ch\_id} \rangle$

$\langle \mathsf{proph\_pat} \rangle$ ::= $\{ \langle \mathsf{id} \rangle \;\& \}^* \langle \mathsf{proph\_id} \rangle$

A save transition $\langle \mathsf{s\_tran} \rangle$ allows the order of the signals in the internal, unbounded buffer to be permuted. See Example 6. Ordinary state-transitions $\langle \mathsf{o\_tran} \rangle$ model the "real" state-transitions of an SDL process graph. The optional part is used to model that input symbols may contain lists of signals.

Example 6 Input Consumption:
Assume that the actual process has $\{s_1, s_2, \ldots, s_7\}$ as its input signal set. Then, the third SDL fragment of Figure 6 corresponds to the following F-SDL fragment:

$$a \in \{s_1, s_2, s_3\} \Rightarrow state(a \,\&\, in) = \;\ldots$$

$$state(s_4 \,\&\, in) = \;\ldots$$

$$a \notin \{s_5, s_6\} \wedge \{s_5, s_6\} \copyright in = in \Rightarrow state(in \frown (a \,\&\, in')) = state((a \,\&\, in) \frown in')$$

$$a \notin \{s_1, s_2, s_3, s_4, s_5, s_6\} \Rightarrow state(a \,\&\, in) = state(in)$$

The first two "conditional equations" model the ordinary transitions; the third models the save-transition; the fourth models a hidden transition, namely the implicit consumption of any occurrence of s_7. In the save transition the antecedent makes sure that a represents the first signal that is not to be saved. The "task of the consequence" is then to move a to the front of the internal buffer and thereafter continue the processing. The behavior for the missing cases (like $state(\epsilon)$) follows implicitly from the monotonicity and continuity constraint imposed on $state$.

□

$\langle$right$\rangle$ ::= $\langle$if_then_else$\rangle$ | $\langle$let_in$\rangle$ | $\langle$output$\rangle$ | $\langle$next_call$\rangle$

$\langle$if_then_else$\rangle$ ::= if $\langle$test$\rangle$ then $\langle$right$\rangle$ **{** elseif $\langle$test$\rangle$ then $\langle$right$\rangle$ **}*** [else $\langle$right$\rangle$ **]** fi

$\langle$test$\rangle$ ::= SDL_EXPR = SDL_EXPR | $\langle$id$\rangle$ mod $\langle$id$\rangle$ = $\langle$id$\rangle$

$\langle$let_in$\rangle$ ::= let **{** $\langle$assignment$\rangle$ //,**}**$^+$ in $\langle$right$\rangle$

$\langle$assignment$\rangle$ ::= $\langle$loc_st_id$\rangle$ = SDL_EXPR

$\langle$output$\rangle$ ::= $\langle$signal$\rangle$ $\langle$selector$\rangle$ $\langle$right$\rangle$

$\langle$selector$\rangle$ ::= $\&_{\langle \text{output\_route\_number} \rangle}$

$\langle$next_call$\rangle$ ::= $\langle$func_id$\rangle$ **[**($\langle$proph_id$\rangle$)**]** **[**(**{** SDL_EXPR //,**}**$^+$)**]** $\langle$ch_id$\rangle$

The different types of decisions are expressed by the $\langle$if_then_else$\rangle$ construct. There are two alternative tests. The left-hand side alternative models ordinary SDL decisions; the right-hand side alternative models nondeterministic decisions (this means that its mod operator is assumed to be syntactically different from SDL's mod operator.

Example 7 Decisions:
The SDL fragment pictured in Figure 7 corresponds to the following F-SDL expression:

```
if $n = 1$ then
   if $b =$ true then ...
   elseif $b =$ false then ...
   fi
elseif $n = 2$ then
   if $k$ mod $2 = 0$ then ...
   else ...
   fi
else
   if $a = 1$ then ...
   elseif $a = 2$ then ...
   elseif $a = 3$ then ...
   elseif $a = 4$ then ...
   fi
fi
```

k is a variable representing the next element of the prophecy stream (the existentially quantified p in the specification of PR on Page 11).
□

In an obvious way, $\langle$let_in$\rangle$ is employed to model assignments to the secondary state-variables. The $\langle$output$\rangle$ construct is used to output signals. $\langle$selector$\rangle$ chooses the right

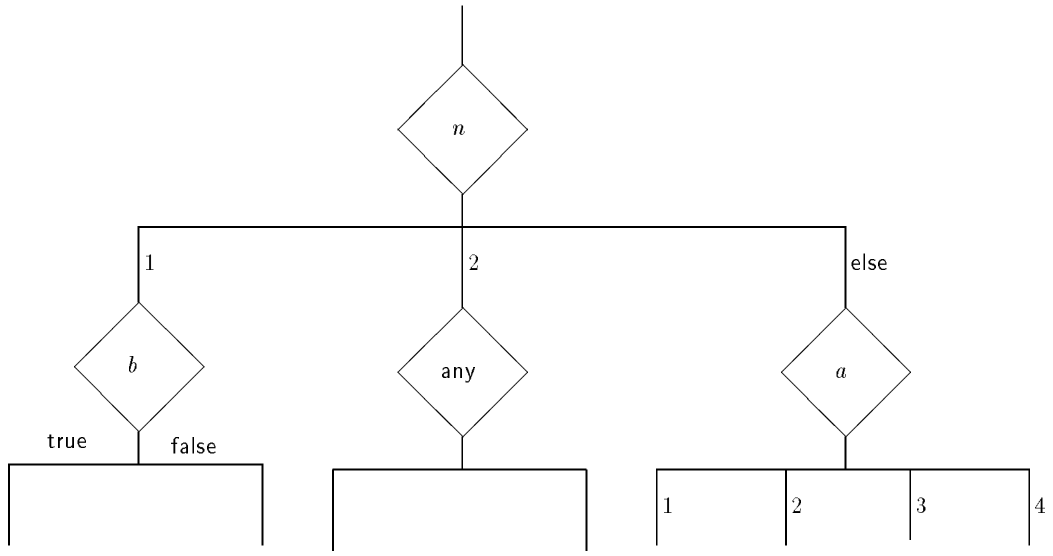

Figure 7: SDL Decisions Fragment

output signal-route. The next control state is characterized by the ⟨next_call⟩ construct. Due to ⟨expr_tuple⟩ it can also be used to model certain assignments to the secondary state variables. See the specifications $\mathrm{PR}_{\mathrm{S}_1}$ and $\mathrm{PR}_{\mathrm{R}_1}$ in Section 6.

⟨rec_proc_spec⟩ ::= rec_process ⟨head⟩ ⟨rec_proc_body⟩ end

⟨rec_proc_body⟩ ::= **{** ⟨equation⟩ //,**}**$^+$

The recursive process is constrained to occur at least once at the right-hand side of an equation in ⟨rec_proc_body⟩. All the other processes occurring in ⟨rec_proc_body⟩ are required to have been specified in the surrounding block. Each recursive F-SDL process corresponds to an SDL process which waits until it receives an input signal, then generates a new process for each equation in its body, and thereafter is responsible for the communication between these new processes and the rest of the network. Thus we basically simulates infinite Focus networks in SDL.

6 Development of a Protocol

The objective of this section is to outline how Focus can be used to develop an SDL specification of a protocol inspired from [Ste76]. In this paper we do not give any formal proofs. However, with respect to each (nontrivial) refinement step, the proof obligations are stated.

We refer to the overall protocol network as SP, and not surprisingly, from an external point of view, it is required to behave as an identity component:[2]

[2]SP is here assigned a subscript 0 to allow the different specifications of the overall network to be distinguished. For example, below we give a more refined specification of SP which we refer to as SP_1.

system $\mathrm{SP}_0(i \in \mathrm{DT}^\omega \rhd o \in \mathrm{DT}^\omega) \equiv$

$o = i$

end.

$\mathrm{DT} = \{dt(d) \mid d \in D\}$ is the set of data signals, where D is some nonempty set of data. If we ignore the keywords system and end, which as explained in Section 5 have no semantics, this is an ordinary Focus specification. However, SP is no complete F-SDL specification because its internal structure has not yet been fixed in an F-SDL manner.

As usual in the case of protocols, the system to be developed consists of a sender, a receiver and a communication medium. Thus, it seems natural to decompose our abstract system specification into three blocks: a block called SND specifying the sender, a block called REC specifying the receiver, and a block called MED specifying the medium. As indicated in Figure 8, at the system level there are 8 channels altogether; two of these are external; the other six are internal. Thus we want an F-SDL system specification of the following form:

system $\mathrm{SP}_1(i \in \mathrm{DT}^\omega \rhd o \in \mathrm{DT}^\omega) \equiv$

$(sd, sn) = \mathrm{SND}_0(i, ma),$
$(ma, md, mn) = \mathrm{MED}_0(sd, sn, ra),$
$(ra, o) = \mathrm{REC}_0(md, mn)$

where

block SND_0 ... end,

block MED_0 ... end,

block REC_0 ... end

end

Although the three blocks are unspecified, we already have enough information to generate the corresponding SDL system diagram.

For each data signal $dt(d)$ input on i, the sender SND generates a unique sequence-number signal $snr(n)$ and repeatedly sends these two signals along sd and sn, respectively, until it receives the sequence-number signal $snr(n)$ on ma. Any sequence-number signal input on ma is of course sent by REC to acknowledge that the corresponding data signal has been received. More formally, the sender can be specified as below:

A similar convention is also used with respect to other specifications.

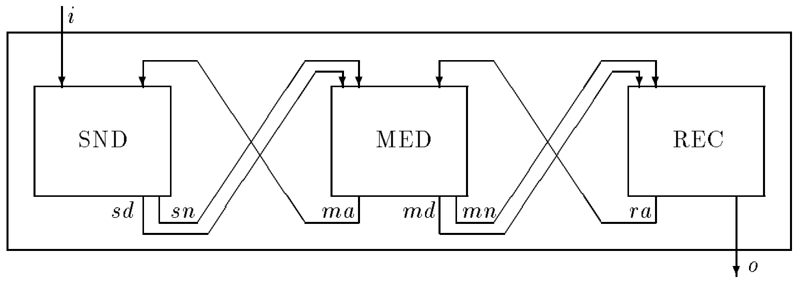

Figure 8: Network for the Protocol.

```
block $\mathrm{SND}_0(i \in \mathrm{DT}^\omega, ma \in \mathrm{SN}^\omega \rhd sd \in \mathrm{DT}^\omega, sn \in \mathrm{SN}^\omega) \equiv$

    let
      $ma' = \propto(ma)$
      $(sd', sn') = \propto(sd, sn)$
    in
      $\#ma' \leq \#i$
          $\Rightarrow$
      $\#sd = \#sn \wedge \forall n \in \mathrm{SN} : \#n©sn' \leq 1 \wedge sd' \sqsubseteq i \wedge$
      if $\#ma' = \#i$ then $\#sd' = \#i$ else $(\#sd = \infty \wedge \#sd' = \#ma' + 1)$

end
```

$\mathrm{SN} = \{snr(n) \,|\, n \in \mathsf{N}\}$ denotes the set of sequence-number signals. The let-construct defines ma' and (sd', sn') to be equal to ma and (sd, sn) minus consecutive repetitions, respectively.

The antecedent states the environment assumption, namely that the length of ma' is less than or equal to the length of i. This is a sensible assumption since the receiver should only acknowledge the data signals it receives.

The first conjunct of the consequence requires sd and sn to be of the same length. This means that SND never sends a data signal without also sending its corresponding sequence-number — and the other way around.

The second conjunct of the consequence makes sure that each sequence-number has maximum one occurrence in sn'. This means that any data signal in sd that is not equal to its predecessor has a corresponding sequence-number that is different from all its predecessors.

The third conjunct of the consequence requires sd' to be a prefix of i. This means that the sender only sends the data-signals input from i, and moreover that they are sent in the order they are received.

The fourth conjunct of the consequence requires that if ma' is of the same length as i then this also holds for sd'; otherwise sd is infinite and the length of sd' is equal to the length of ma' plus 1. This means that the sender does not send more data signals (when repetitions are ignored) than it receives on i, and that when no acknowledgement for a certain data signal is received, then this signal is sent repeatedly forever.

Note that the second conjunct of the else branch together with conjunct two and three of the consequence make sure that each data signal input on i is assigned a unique sequence-number.

The communication medium MED is lossy in the sense that both acknowledgements and data signals (with the corresponding sequence-numbers) can be lost. It is assumed that a data signal is lost iff the corresponding sequence number signal is lost. Although the medium is lossy, it is "friendly" in the sense that if the same signal pair is repeatedly sent, then it will eventually get through, and similar for acknowledgements. This can be formalized as follows:

$$\mathsf{block}\ \mathrm{MED}_0(sd \in \mathrm{DT}^\omega, sn, ra \in \mathrm{SN}^\omega \rhd ma \in \mathrm{SN}^\omega, md \in \mathrm{DT}^\omega, mn \in \mathrm{SN}^\omega) \equiv$$

$$\begin{array}{l} \exists p_1, p_2 \in \{0,1\}^\omega : \#1 © p_1 = \#1 © p_2 = \infty \wedge \\ \quad \exists d \in \{1\}^\omega : (ma, d) = \{(n, 1) \mid n \in \mathrm{SN}\} © (ra, p_1) \wedge \\ \quad \exists d \in \{1\}^\omega : (md, mn, d) = \{(d, n, 1) \mid d \in \mathrm{DT} \wedge n \in \mathrm{SN}\} © (sd, sn, p_2) \end{array}$$

$$\mathsf{end}$$

Since the medium is not supposed to be translated into SDL, we do not have to worry about that the signals received on ra and sn cannot be syntactically distinguished.

The two existentially quantified, infinite streams p_1 and p_2 model the nondeterminism. When the n'th element of p_1 is a 0 then the n'th pair of signals input from sd and sn is lost; otherwise it is transmitted properly. Since p_1 has infinitely many 1's it follows that an input pair eventually will get through if it is sent often enough. p_2 models the lossiness with respect to acknowledgements in a similar way.

The receiver REC is supposed to output any sequence-number signal it receives on mn along ra, and any data-element minus repetitions along o:

$$\mathsf{block}\ \mathrm{REC}_0(md \in \mathrm{DT}^\omega, mn \in \mathrm{SN}^\omega \rhd ra \in \mathrm{SN}^\omega, o \in \mathrm{DT}^\omega) \equiv$$

$$\mathsf{let}\ (md', mn') = \propto(md, mn)\ \mathsf{in}\ ra = mn|_{\#md} \wedge o = md'$$

$$\mathsf{end}$$

The next step is to prove that this decomposition is correct — in other words, to verify that

$$\mathrm{SP}_0 \rightsquigarrow \mathrm{SP}_1.$$

To do so it is according to [BS94] enough to formulate three invariants I_1, I_2 and I_3 with the elements of (i, ma), (i, sd, sn, ra) and (i, md, mn) as their only free variables, respectively, and then prove that the following six proof-obligations are fulfilled:

$I_1[^{ma}_{\epsilon}] \wedge I_2[^{sd}_{\epsilon}\ ^{sn}_{\epsilon}\ ^{ra}_{\epsilon}] \wedge I_3[^{md}_{\epsilon}\ ^{mn}_{\epsilon}]$ $\quad$ $I_1 \wedge R_{\mathrm{SND}_0} \wedge I_2 \wedge R_{\mathrm{MED}_0} \Rightarrow I_3$

$I_2 \wedge R_{\mathrm{MED}_0} \wedge I_3 \wedge R_{\mathrm{REC}_0} \Rightarrow I_1$ $\quad$ $\mathsf{safe}(\lambda ma : I_1) \wedge \mathsf{safe}(\lambda sd, sn, ra : I_2) \wedge \mathsf{safe}(\lambda md, mn : I_3)$

$I_1 \wedge R_{\mathrm{SND}_0} \wedge I_3 \wedge R_{\mathrm{REC}_0} \Rightarrow I_2$ $\quad$ $I_1 \wedge R_{\mathrm{REC}_0} \wedge I_2 \wedge R_{\mathrm{MED}_0} \wedge I_3 \wedge R_{\mathrm{SND}_0} \Rightarrow R_{\mathrm{SP}_0}$

Remember that for any specification S, R_S denotes its input/output relation. In the first proof obligation the substitution operators have standard semantics ($[^a_b]$ replaces all occurrences of the variable a with b). In the fifth the lambda-notation is used to say that it is enough to prove that I_1, I_2 and I_3 are safe when i is kept constant.

The six proof-obligations follow straightforwardly if the three invariants are defined as below:

$$I_1 \stackrel{\mathsf{def}}{=} \#\!\propto\!(ma) \leq i, \quad I_2 \stackrel{\mathsf{def}}{=} \#\!\propto\!(sn) \leq i \wedge \#\!\propto\!(ra) \leq i, \quad I_3 \stackrel{\mathsf{def}}{=} \#\!\propto\!(mn) \leq i.$$

At the system level our development is now complete. What remains is to transform the two block specifications that we want to implement, namely SND and REC into F-SDL syntax. The specification of the medium is not refined any further. The blocks SND and REC have only one process each, and these processes are therefore constrained to behave in exactly the same way as their respective blocks.

```
block $\mathrm{SND}_1(i \in \mathrm{DT}^\omega, ma \in \mathrm{SN}^\omega \rhd sd \in \mathrm{DT}^\omega, sn \in \mathrm{SN}^\omega) \equiv$

  $(sd, sn) = \mathrm{SND\_PROC}_0(i, ma)$

where

  ord_process $\mathrm{SND\_PROC}_0(i \in \mathrm{DT}^\omega, ma \in \mathrm{SN}^\omega \rhd sd \in \mathrm{DT}^\omega, sn \in \mathrm{SN}^\omega) \equiv R_{\mathrm{SND}_0}$ end

end


block $\mathrm{REC}_1(md \in \mathrm{DT}^\omega, mn \in \mathrm{SN}^\omega \rhd ra \in \mathrm{SN}^\omega, o \in \mathrm{DT}^\omega) \equiv$

  $(ra, o) = \mathrm{REC\_PROC}_0(md, mn)$

where

  ord_process $\mathrm{REC\_PROC}_0(md \in \mathrm{DT}^\omega, mn \in \mathrm{SN}^\omega \rhd ra \in \mathrm{SN}^\omega, o \in \mathrm{DT}^\omega) \equiv R_{\mathrm{REC}_0}$ end

end
```

The verification of this refinement step, namely that

$\mathrm{SND}_0 \rightsquigarrow \mathrm{SND}_1,$

$\mathrm{REC}_0 \rightsquigarrow \mathrm{REC}_1,$

is trivial. Each process specification can then be decomposed into a FM and a PR component, as explained above. In the case of SND_PROC an additional feedback channel is introduced to allow for the use of timers.

```
ord_process $\mathrm{SND\_PROC}_1(i \in \mathrm{DT}^\omega, ma \in \mathrm{SN}^\omega \rhd sd \in \mathrm{DT}^\omega, sn \in \mathrm{SN}^\omega) \equiv$

    $(b) = \mathrm{FM}(i, ma, t),$
    $(sd, sn, t) = \mathrm{PR}_{\mathrm{S}_0}(b)$

where

    $\mathrm{PR}_{\mathrm{S}_0}(b \in (\mathrm{DT} \cup \mathrm{SN} \cup \mathrm{TI})^\omega \rhd sd \in \mathrm{DT}^\omega, sn \in \mathrm{SN}^\omega, t \in \mathrm{TI}^\omega) \equiv$

        let $i = \mathrm{DT}©b, ma = \mathrm{SN}©b$ in
            $R_{\mathrm{SND\_PROC}_0} \wedge \#t = \#sn \wedge \forall n \in \mathsf{N} : \#ti(n)©t = \#snr(n)©sn$

    end

end


ord_process $\mathrm{REC\_PROC}_1(md \in \mathrm{DT}^\omega, mn \in \mathrm{SN}^\omega \rhd ra \in \mathrm{SN}^\omega, o \in \mathrm{DT}^\omega) \equiv$

    $(b) = \mathrm{FM}(md, mn),$
    $(ra, o) = \mathrm{PR}_{\mathrm{R}_0}(b)$

where

    $\mathrm{PR}_{\mathrm{R}_0}(b \in (\mathrm{DT} \cup \mathrm{SN})^\omega \rhd ra \in \mathrm{SN}^\omega, o \in \mathrm{DT}^\omega) \equiv$

        let $md = \mathrm{DT}©b, mn = \mathrm{SN}©b$ in $R_{\mathrm{REC\_PROC}_0}$

    end

end
```

$\mathrm{TI} = \{ti(n) \mid n \in \mathsf{N}\}$ denotes the set of timers. Strictly speaking, the subscripts assigned to the two PR components violates the F-SDL syntax. They have only been introduced to simplify the presentation. Again, the correctness of these two decompositions, namely that

$$\mathrm{SND\_PROC}_0 \rightsquigarrow \mathrm{SND\_PROC}_1,$$

$$\mathrm{REC\_PROC}_0 \rightsquigarrow \mathrm{REC\_PROC}_1$$

follows easily. The final step is to refine the input/output-relations of the two PR components into F-SDL notation:

$$\mathrm{PR}_{\mathrm{S}_1}(b \in (\mathrm{DT} \cup \mathrm{SN} \cup \mathrm{TI})^\omega \rhd sd \in \mathrm{DT}^\omega, sn \in \mathrm{SN}^\omega, t \in \mathrm{TI}^\omega) \equiv$$

$$\exists l \in \mathsf{N} : \exists d \in D : \exists start \in \mathsf{N} \times D \to (\mathrm{DT} \cup \mathrm{SN} \cup \mathrm{TI})^\omega \xrightarrow{c} \mathrm{DT}^\omega \times \mathrm{SN}^\omega \times \mathrm{TI}^\omega : \quad start(l,d)(b) = (sd, sn, t)$$

where $\forall l, n \in \mathsf{N} : \forall d, d' \in D : \forall s \in \mathrm{SN} \cup \mathrm{TI} : \forall in, in' \in (\mathrm{DT} \cup \mathrm{SN} \cup \mathrm{TI})^\omega$:

$$start(l,d)(in) = next(1,d)(in)$$

$$next(l,d)(dt(d') \,\&\, in) = \mathsf{let}\ l = l + 1\ \mathsf{in}\ dt(d') \,\&_1\, snr(l) \,\&_2\, ti(l) \,\&_3\, rep(l,d')(in)$$

$$s \in \{snr(n), ti(k) \mid n, k \in \mathsf{N}\} \Rightarrow next(l,d)(s \,\&\, in) = next(l,d)(in)$$

$$rep(l,d)(ti(n) \,\&\, in) = \mathsf{if}\ n = l\ \mathsf{then}\ dt(d) \,\&_1\, snr(l) \,\&_2\, ti(l) \,\&_3\, rep(l,d)(in)\ \mathsf{else}\ rep(l,d)(in)\ \mathsf{fi}$$

$$rep(l,d)(snr(n) \,\&\, in) = \mathsf{if}\ n = l\ \mathsf{then}\ next(l,d)(in)\ \mathsf{else}\ rep(l,d)(in)\ \mathsf{fi}$$

$$s \notin \{dt(d) \mid d \in D\} \wedge \{dt(d) \mid d \in D\} \copyright in = in \Rightarrow rep(l,d)(in \frown (s \,\&\, in')) = rep(l,d)((s \,\&\, in) \frown in')$$

end

$$\mathrm{PR}_{\mathrm{R}_1}(b \in (\mathrm{DT} \cup \mathrm{SN})^\omega \rhd ra \in \mathrm{SN}^\omega, o \in \mathrm{DT}^\omega) \equiv$$

$$\exists l \in \mathsf{N} : \exists d \in D : \exists start \in \mathsf{N} \times D \to (\mathrm{DT} \cup \mathrm{SN})^\omega \xrightarrow{c} \mathrm{SN}^\omega \times \mathrm{DT}^\omega : \\ start(l,d)(b) = (ra, o)$$

where $\forall l, n \in \mathsf{N} : \forall d, d' \in D : \forall s \in \mathrm{DT} \cup \mathrm{SN} : \forall in, in' \in (\mathrm{DT} \cup \mathrm{SN})^\omega :$

$$start(l,d)(in) = next\_dt(1,d)(in)$$

$$next\_dt(l,d)(dt(d') \,\&\, in) = next\_sn(l,d')(in)$$

$$s \notin \{snr(n) \mid n \in \mathsf{N}\} \wedge \{snr(n) \mid n \in \mathsf{N}\} \copyright in = in \Rightarrow \\ next\_dt(l,d)(in \frown (s \,\&\, in')) = next\_dt(l,d)((s \,\&\, in) \frown in')$$

$$next\_sn(l,d)(snr(n) \,\&\, in) = snr(n) \,\&_1 \text{ if } n = l \text{ then} \\ next\_dt(l,d)(in) \text{ else } dt(d) \,\&_2\, next\_dt(n,d)(in)$$

$$s \notin \{dt(d) \mid d \in D\} \wedge \{dt(d) \mid d \in D\} \copyright in = in \Rightarrow \\ next\_sn(l,d)(in \frown (s \,\&\, in')) = next\_sn(l,d)((s \,\&\, in) \frown in')$$

end

To verify the correctness of these decompositions, namely that

$$\mathrm{PR}_{\mathrm{S}_0} \leadsto \mathrm{PR}_{\mathrm{S}_1}, \qquad \mathrm{PR}_{\mathrm{R}_0} \leadsto \mathrm{PR}_{\mathrm{R}_1},$$

it must be shown that

$$R_{\mathrm{PR}_{\mathrm{S}_1}} \Rightarrow R_{\mathrm{PR}_{\mathrm{S}_0}}, \qquad R_{\mathrm{PR}_{\mathrm{R}_1}} \Rightarrow R_{\mathrm{PR}_{\mathrm{R}_0}},$$

which follows by an inductive argumentation.

We now have a complete F-SDL specification if the medium is ignored. This specification can easily be translated into SDL. SDL versions of the two process specifications are pictured in Figure 9 and 10, respectively. Note the almost one-to-one relationship. The only "real" difference is that in both cases an additional local state variable n has been introduced. Moreover, as a simplification d' has been replaced by d.

7 Conclusions

It has been outlined how Focus and a restricted version of SDL can be assigned the same kind of stream-based, denotational semantics. Based on the proposed SDL semantics expressed in Focus, a language called F-SDL has been defined. It characterizes a set of Focus specifications whose elements allow an automatic "one-to-one" translation into SDL. The proposed technique was demonstrated on hand of a protocol.

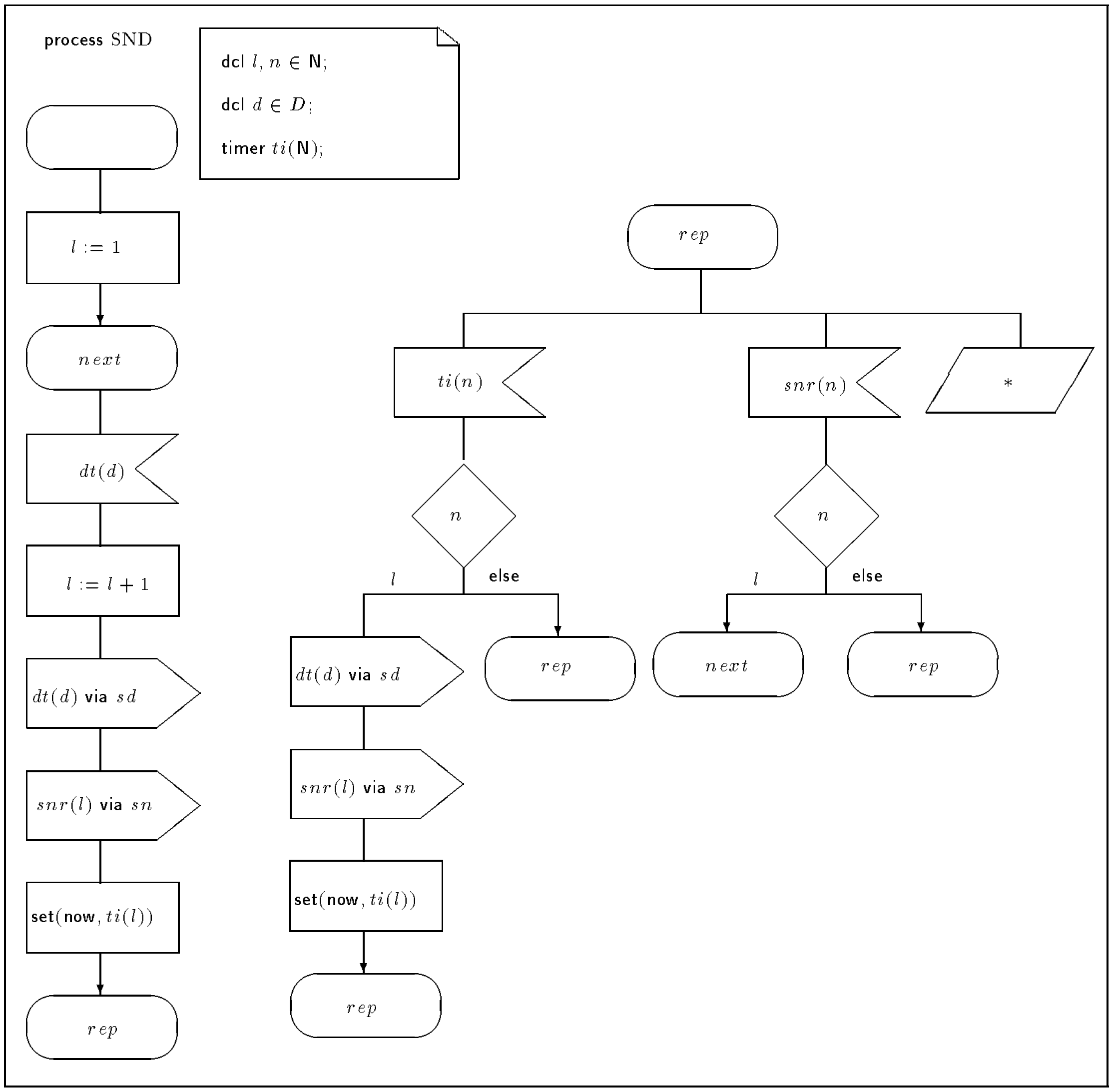

Figure 9: The Sender Process in SDL.

The merger of Focus and SDL into one methodology offers several advantages. Focus with its well-defined formal semantics and very abstract nature allows the use of formal proof techniques for the validation, verification and development of specifications. In particular, employing Focus a certain class of SDL specifications can be developed in a top-down fashion — in other words, in the same style the SDL specification of the protocol was developed in Section 6. The SDL specification was refined from an abstract Focus specification stating that the overall network should behave as an identity component — a requirement specification whose correctness is obvious! Thus the protocol example shows how one can proceed from a simple Focus specification to a non-trivial SDL specification. More complicated protocols can of course be developed accordingly. Another advantage of using Focus is the flexibility with respect to environment assumptions and the treatment of specifications that are not supposed to be implemented. An example of the latter is the medium MED of Section 6. Assumptions about a component's environment can be stated by splitting the specification into an assumption and a commitment part. For

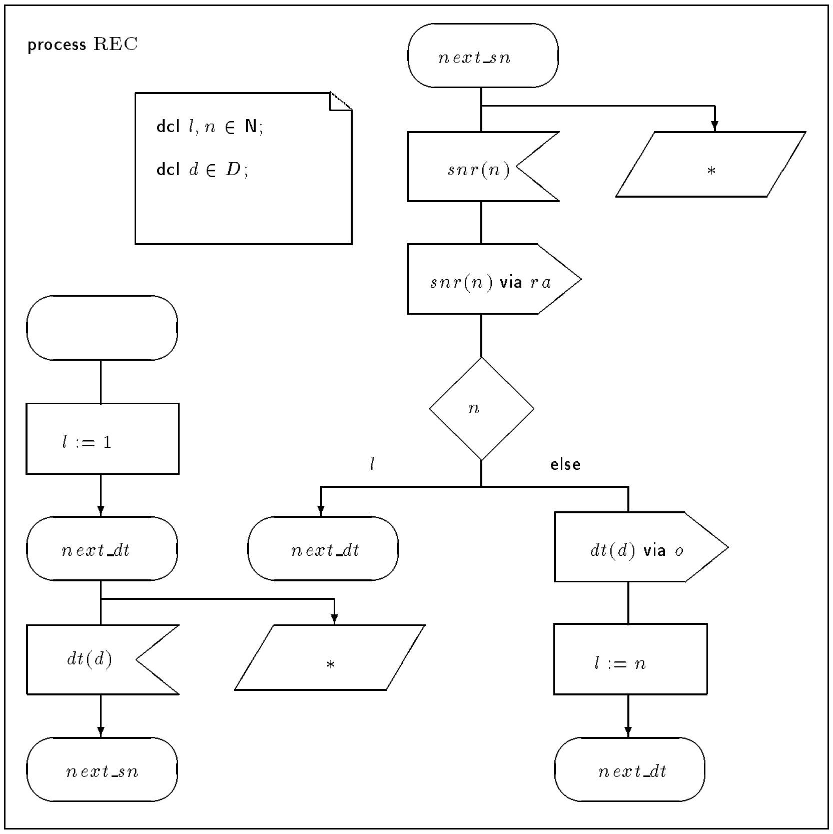

Figure 10: The Receiver Process in REC.

example, in the specification SND_0 on Page 23, the antecedent can be seen as an environment assumption, and the consequence is a commitment which must be fulfilled by the component whenever the environment behaves in accordance with the environment assumption.

SDL is a specification language. This means that the readability and structure of SDL specifications often is of crucial importants. For this reason, F-SDL has been designed in such a way that there is a straightforward mapping into SDL. This means that the user has full control of the syntactic structure of the SDL specification he is developing. In SDL there is a lot of syntactic sugar, which can be hard to model directly in Focus. This can be dealt with by defining and implementing user/controlled transformation rules which allow the user to transform the generated SDL specification into its optimal form.

From the Focus user's point of view, the embedding of SDL as a target language means that the many tools and environments already designed for SDL can be used to transform a developed specification into the chosen target architecture. For example, there are SDL tools which allow an automatic translation into C^{++} [FHvLW93]. Moreover, since a completely formal development is very resource demanding, the Focus user may concentrate his efforts on the critical parts of a system description and specify the less essential aspects directly in SDL.

Although the considered sublanguage is sufficiently expressive to deal with non-trivial applications, many SDL facilities have been ignored. However, it is relatively easy to extend the proposed approach to handle a much richer part of SDL, including the complete SDL timer constructs, procedures (not remote call), and services. On the other hand, the treatment of the more OO-related facilities of SDL, including the full generality of the constructs for process creation, is difficult if at all possible in the context of Focus. This is of course not very surprising since the formal treatment of object-oriented languages is known to be very difficult. In fact for the time being there is to our knowledge no compositional and relative complete proof method for an object-oriented programming language.

Some case-studies based on the proposed technique has been carried out. In [FS93] a Min/Max component has been developed, and [Phi93] designs an SDL specification of a production cell using the assumption/commitment calculus of [SDW93].

8 Acknowledgements

We would like to thank M. Broy, C. Facchi and M. Fuchs who have read earlier drafts of this paper and provided valuable feedback.

References

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems — an introduction to Focus. Technical Report SFB 342/2/92 A, Technische Universität München, 1992.

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall, 1991.

[Bro91] M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.

[Bro93] M. Broy. (Inter-) Action refinement: The easy way. In M. Broy, editor, *Proc. Program Design Calculi*, pages 121–158. Springer, 1993.

[BS94] M. Broy and K. Stølen. Specification and refinement of finite dataflow networks — a relational approach. Technical Report SFB 342/7/94 A, Technische Universität München, 1994.

[CCI93] CCITT, editor. *Functional Specification and Description Language (SDL), Recommendation Z.100*. International Telecommunication Union, Geneva, 1993.

[Ded92] F. Dederichs. *Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen*. PhD thesis, Technische Universiät München, 1992. Also available as SFB-report 342/17/92 A, Technische Universiät München.

[FHvLW93] J. Fischer, E. Holz, M. van Löwis, and D. Witaszek. A run time library for the simulation of SDL'92-specifications. In O. Færgemand and A. Sarma, editors, *Proc. 6'th SDL Forum*, pages 105–118. North-Holland, 1993.

[FS93] M. Fuchs and K. Stølen. Development of a distributed min/max component. Technical Report SFB 342/18/93 A, Technische Universität München, 1993.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Proc. Information Processing 74*, pages 471–475. North-Holland, 1974.

[Kel78] R. M. Keller. Denotational models for parallel programs with indeterminate operators. In E. J. Neuhold, editor, *Proc. Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[Phi93] J. Philipps. Spezifikation einer Fertigungszelle — eine Fallstudie in Focus. Master's thesis, Technische Universiät München, 1993.

[SDW93] K. Stølen, F. Dederichs, and R. Weber. Assumption/commitment rules for networks of asynchronously communicating agents. Technical Report SFB 342/2/93 A, Technische Universität München, 1993.

[Ste76] V. Stenning. A data transfer protocol. *Computer Networks*, 1:98–110, 1976.