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

Eckhardt Holz[a] and Ketil Stølen[b]

[a]Fachbereich Informatik, Humboldt Universität zu Berlin, Lindenstrasse 54a, D-10099 Berlin, Germany

[b]Fakultät für Informatik, TU München, Postfach 20 24 20, D-80290 München, Germany

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 [1], [2] is a methodology 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, directed 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 sense 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 [3] has been developed by ITU-TSS 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 two orthogonal approaches. Because of its very mathematical and abstract notation, Focus has its strength in the area of formal refinement

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

and verification. SDL, on the other hand, due to its graphical notation and many structuring constructs, is 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 important 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.

A development of an SDL specification in Focus is split into three phases: first a requirement specification is formulated in Focus; then in a step-wise fashion this requirement specification is refined into an F-SDL specification; finally the F-SDL specification is translated into SDL.

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

N denotes the set of positive natural numbers. 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 $d \in D$, $r, s \in D^\omega$, $A \subseteq D$ 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; $A©r$ denotes the result of filtering away all actions not in A (the projection of r on A); $d \& s$ 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 a tuple of empty streams when the size of this 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 function $\tau \in (D^\omega)^n \to (D^\omega)^m$ 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 $\tau \in (D^\omega)^n \to (D^\omega)^m$ is pulse-driven iff:

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

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

To model timeouts 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 are 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 one additional function in our examples, namely a function $\propto$ which eliminates repetitions. More explicitly, if d, e are n-tuples of actions, and r is an n-tuple of streams, then $\propto$ is a stream processing function such that the following axioms hold:

$$\propto(d \,\&\, \epsilon) = d \,\&\, \epsilon, \quad \propto(d \,\&\, e \,\&\, r) = \mathsf{if}\ d = e\ \mathsf{then}\ \propto(d \,\&\, r)\ \mathsf{else}\ d \,\&\, \propto(e \,\&\, r).$$

Note that these axioms together with the monotonicity and continuity constraints deter-

mine the semantics also for stream tuples whose stream components are not of the same length.

3. FOCUS AND ITS STREAM SEMANTICS

Depending on the logical concepts they employ, Focus specifications can be divided into a number of subclasses. 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.

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:

$$[\![\ S\ ]\!] \stackrel{\mathsf{def}}{=} \{\tau \in (D^\omega_{\surd})^n \stackrel{cp}{\rightarrow} (D^\omega_{\surd})^m \mid \forall r \in (D^\infty_{\surd})^n : (\diamond r, \diamond\tau(r)) \models R\},$$

where $(\diamond r, \diamond\tau(r)) \models R$ 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 $\diamond\tau(r)$.

Networks of specifications are expressed in an equational style. For example, the network S pictured in Figure 1, consisting of 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, $[\![\ S\ ]\!] \stackrel{\mathsf{def}}{=} \{\tau_1 \otimes \tau_2 \mid \tau_1 \in [\![\ S_1\ ]\!] \wedge \tau_2 \in [\![\ S_2\ ]\!]\}$, where for any pair of timed stream tuples i and r, $(\tau_1 \otimes \tau_2)(i, r) \stackrel{\mathsf{def}}{=} (o, s)$ iff (o, s) is the least fixpoint solution with respect to i and r. 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 [4].

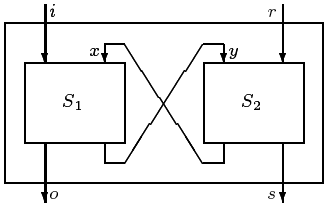

Figure 1. The Network S

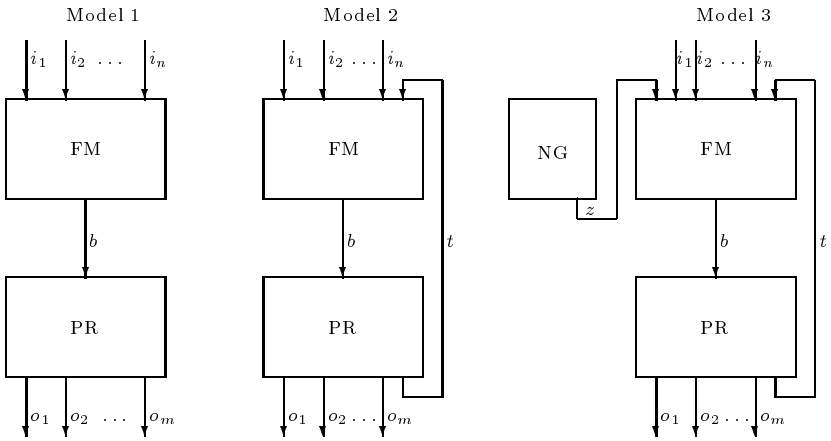

Figure 2. Three Models of an SDL Process

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 $[\![S_2]\!] \subseteq [\![S_1]\!]$.

4. SDL AND ITS STREAM SEMANTICS

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 two blocks or between a block 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 sets of processes.

Sets of 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 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 non-deterministic decision; deferred consumption of input signals.

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.

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 [5], 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 [6] 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:

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

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

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

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. 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 [5], with respect to the internal, unbounded buffer, the behavior of a deterministic SDL processing component can be modeled by a function

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

which for any p-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:

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

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

The variables $l_1, \ldots, l_p$ represent the secondary state of an SDL process. The existentially quantified function variable g 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. Based on this outline, we may define the semantics of a simple SDL process as $[\![$ SDL_PROC $]\!]$, where

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

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

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 $g \in D^p \to (D \cup$

$T)^\omega \xrightarrow{c} (D^\omega)^m \times T^\omega$ characterizing the behavior of the processing component PR. More explicitly, to require that

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

for all timer signals $timer(n) \in T$.

Clearly, the first alternative allows us to model more SDL-specifications. However, the additional expressiveness we then get is not particularly interesting from a practical 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:

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

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

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

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

The other way of expressing nondeterminism explicitly in SDL — so-called nondeter-

ministic 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 an oracle q in the specification of PR:

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

$$\exists q \in \mathsf{N}^\infty : \exists l_1, \ldots, l_p \in D : \exists g \in \mathsf{N}^\infty \to D^p \to (D \cup T)^\omega \xrightarrow{c} (D^\omega)^m \times T^\omega :$$
$$g(q)(l_1, \ldots, l_p)(b) = (o_1, \ldots, o_m, t) \text{ where } Q$$

The existentially quantified q 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 $(q[k] \bmod 5) + 1$ is chosen where $q[k]$ denotes the k'th natural number in q.

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 match SDL specifications to such a degree that an automatic translation into SDL is almost straightforward. Because of the space constraints only certain restricted aspects can be discussed here. The reader is referred to [7] for more details.

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

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

⟨sys_spec⟩ ::= system ⟨head⟩ **[** ⟨data_def⟩ **]** ⟨sys_body⟩ where ⟨sys_defs⟩ end

⟨head⟩ ::= ⟨id⟩ (**{** ⟨ch_decl⟩ //,**}*** ▷ **{** ⟨ch_decl⟩ //,**}***) ≡

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

⟨sys_defs⟩ ::= **{** ⟨block_spec⟩ //,**}**$^+$

⟨sys_spec⟩ characterizes a finite Focus network of blocks (i.e. as a set of equations) whose definitions are given in ⟨sys_defs⟩. ⟨data_def⟩ 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. There are of course a number of constraints. For example any block identifier occurring in the the right-hand side of an equation must also be defined in ⟨sys_defs⟩.

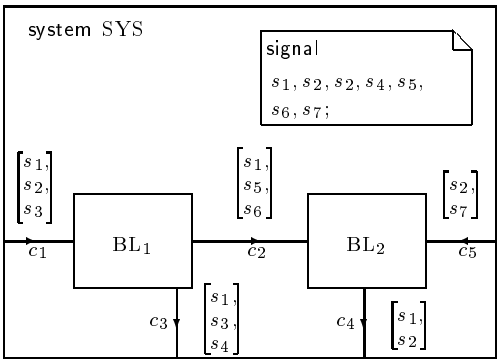

Figure 3. An SDL System Diagram

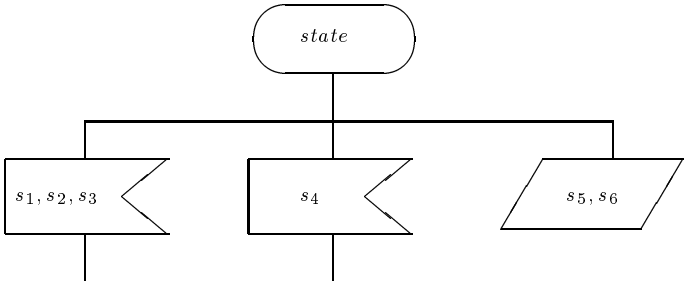

Figure 4. A State Transition Fragment

The SDL system diagram in Figure 3 corresponds to the following F-SDL specification:

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

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

$$\textsf{where} \quad \textsf{block BL}_1 \ldots, \textsf{block BL}_2 \ldots \quad \textsf{end}$$

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

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

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

$$\langle \textsf{proc\_spec} \rangle ::= \langle \textsf{ord\_proc\_spec} \rangle \parallel \langle \textsf{rec\_proc\_spec} \rangle$$

By ordinary processes $\langle$ord_proc_spec$\rangle$ we characterize the SDL processes which do not create other processes. A recursive process $\langle$rec_proc_spec$\rangle$, 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.

We now characterize the syntactic structure of ordinary processes.

$$\langle \textsf{ord\_proc\_spec} \rangle ::= \textsf{ord\_process } \langle \textsf{head} \rangle\ [\ \langle \textsf{data\_def} \rangle\ ]\ \langle \textsf{proc\_body} \rangle \textsf{ where } \langle \textsf{pr\_spec} \rangle \textsf{ end}$$

$$\langle \textsf{proc\_body} \rangle ::= [\ \langle \textsf{ng\_eq} \rangle\ ,\ ]\ [\ \langle \textsf{fm\_eq} \rangle\ ,\ ]\ \langle \textsf{pr\_eq} \rangle$$

$\langle$ng_eq$\rangle$, $\langle$fm_eq$\rangle$ and $\langle$pr_eq$\rangle$ model respectively the NG-equation, the FM-equation and the PR-equation, as explained in Section 4. NG and FM are specification constants and do not have to be explicitly defined in F-SDL. $\langle$pr_spec$\rangle$ gives the F-SDL specification of the processing component PR.

$$\langle \mathsf{pr\_spec} \rangle \ ::= \ \mathrm{PR}\,(\,[\,\langle \mathsf{ch\_decl} \rangle\,]\ \rhd\ \{\,\langle \mathsf{ch\_decl} \rangle\,//,\}^{*}\,) \equiv \langle \mathsf{pr\_body} \rangle\ \mathsf{where}\ \langle \mathsf{proc\_graph} \rangle\ \mathsf{end}$$

The detailed structure of $\langle \mathsf{pr\_body} \rangle$ corresponds to the similar fragment of the specification PR in Section 4. The process graph $\langle \mathsf{proc\_graph} \rangle$ (the formula Q in Section 4) is basically a functional program based on pattern matching.

$$\langle \mathsf{proc\_graph} \rangle \ ::= \ [\,\langle \mathsf{var\_decls} \rangle\,]\ \langle \mathsf{i\_tran} \rangle\ [\,,\,\{\,\langle \mathsf{tran} \rangle\,//,\}^{+}\,]$$

$\langle \mathsf{var\_decls} \rangle$ allows us to introduce universally quantified variables needed for the pattern matching. $\langle \mathsf{i\_tran} \rangle$ is used to model the initialization transition — the transition from the start symbol to the first control state. $\langle \mathsf{tran} \rangle$ models any other state transitions of an SDL process graph. $\langle \mathsf{i\_tran} \rangle$ is just a special case of $\langle \mathsf{tran} \rangle$. For an example of a complete F-SDL process graph, see the specification PR in Section 6.

$$\langle \mathsf{tran} \rangle \ ::= \ [\,\langle \mathsf{func\_id} \rangle \ \notin \ \langle \mathsf{func\_set} \rangle \ \Rightarrow\ ]\,(\,\langle \mathsf{h\_tran} \rangle \,\|\, \langle \mathsf{v\_tran} \rangle\,)$$

The optional part of $\langle \mathsf{tran} \rangle$ is used to model an SDL transition from an asterisk state.

A hidden transition $\langle \mathsf{h\_tran} \rangle$ models the way an SDL process consumes input signals for which there is no input or save symbol. A visible transition $\langle \mathsf{v\_tran} \rangle$ models any other transition, namely a save transition or an ordinary state transition.

$$\langle \mathsf{v\_tran} \rangle \ ::= \ \langle \mathsf{s\_tran} \rangle \,\|\, \langle \mathsf{o\_tran} \rangle$$

A save transition $\langle \mathsf{s\_tran} \rangle$ allows the order of the signals in the internal, unbounded buffer to be permuted. Ordinary state-transitions $\langle \mathsf{o\_tran} \rangle$ model the "real" state-transitions of an SDL process graph.

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

The optional part is used to model that input symbols may contain lists of signals.

Assume that the actual process has $\{s_1, s_2, \ldots, s_7\}$ as its input signal set. Then, the SDL fragment of Figure 4 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.

6. DEVELOPMENT OF A PROTOCOL SPECIFICATION

In this Section we sketch how Focus can be used to develop an SDL specification of a protocol based on [8]. For more details, see [7]. We refer to the overall protocol network as SP (for Stenning Protocol), and not surprisingly, from an external point of view, it is required to behave as an identity component:

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

$\mathrm{DT} = \{dt(d) \mid d \in D\}$ is the set of data signals, where D is some nonempty set of data. If we ignore the keyword system, which as explained in Section 5 has no semantics, this is an ordinary Focus specification. However, SP_0 is not a complete F-SDL specification because its internal structure has not yet been fixed in an F-SDL manner. SP is assigned a subscript to allow the different specifications of the same component to be distinguished.

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. At the block 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:

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

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

$$(ma, mn, md) = \mathrm{MED}(sd, sn, ra),$$

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

$$\textsf{where}\quad \textsf{block}\ \mathrm{SND}\ \ldots, \textsf{block}\ \mathrm{MED}\ \ldots, \textsf{block}\ \mathrm{REC}\ \ldots\quad \textsf{end}$$

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 $sn(n)$ and repeatedly sends these two signals along sd and sn, respectively, until it receives the sequence-number signal $sn(n)$ 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:

$$
\begin{array}{l}
\mathsf{block}\ \mathrm{SND}(i \in \mathrm{DT}^\omega, ma \in \mathrm{SN}^\omega \rhd sd \in \mathrm{DT}^\omega, sn \in \mathrm{SN}^\omega) \equiv \\
\quad \mathsf{let} \\
\quad\quad ma' = \propto(ma) \\
\quad\quad (sd', sn') = \propto(sd, sn) \\
\quad \mathsf{in} \\
\quad\quad \#ma' \leq \#i \\
\quad\quad\quad \Rightarrow \\
\quad\quad \#sd = \#sn \wedge \forall n \in \mathrm{SN} : \#n \copyright sn' \leq 1 \wedge sd' \sqsubseteq i \wedge \\
\quad\quad \mathsf{if}\ \#ma' = \#i\ \mathsf{then}\ \#sd' = \#i\ \mathsf{else}\ \#sd = \infty \wedge \#sd' = \#ma' + 1
\end{array}
$$

$\mathrm{SN} = \{sn(n) \mid n \in \mathsf{N}\}$ denotes the set of sequence-number signals. The let-construct defines ma' and (sd', sn') to be equal to mn 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 consequent 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 consequent 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 consequent 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 consequent 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 it never receives an acknowledgement for a data signal, then this signal is sent repeatedly forever.

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

The medium and the receiver can be specified accordingly. The verification of this decomposition boils down to formulating three invariants and showing that six proof-obligations based on these invariants are valid (see [7]).

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. Thus the verification of this refinement step is trivial. Each process specification is then decomposed

into a FM and a PR component, as explained above. In the case of SND an additional feedback channel is introduced to allow for the use of timers. These two decompositions are verified in the same way as the decomposition of SP_0. Then, the specification of the sender's PR component is refined into:

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

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

where $\forall l, n \in \mathsf{N} : \forall d, d' \in D : \forall s \in \mathrm{SN} \cup T : \forall in, in' \in (\mathrm{DT} \cup \mathrm{SN} \cup T)^\omega :$

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

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

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

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

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

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

end

The translation algorithm then gives the SDL specification of Figure 5. The PR component of the receiver can be refined accordingly.

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.

The merge 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 as the SDL protocol specification was developed in Section 6. The SDL specification was formally refined from an abstract Focus specification stating that the overall network should behave as an identity component — a

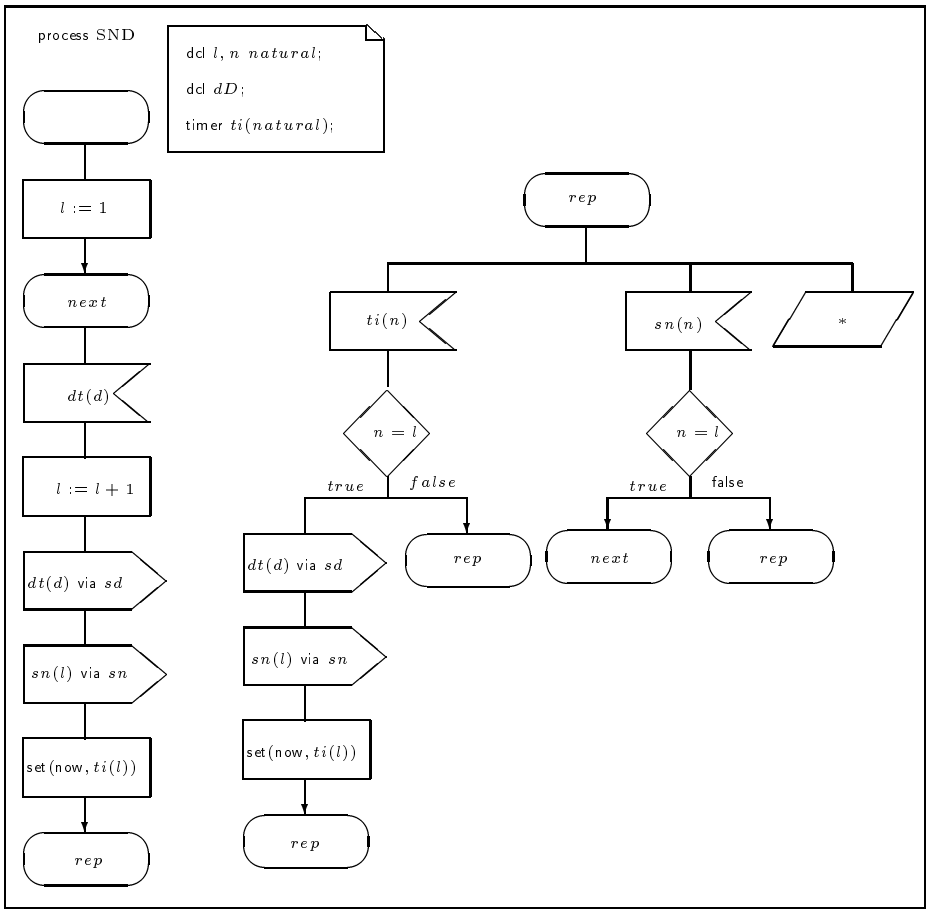

Figure 5. The Sender Process in SDL

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 example, in the specification SND the antecedent can be seen as an environment assumption, and the consequent 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 is often of crucial importance. 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^{++} [9]. 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 seems to be relatively easy to extend the proposed approach to handle a much richer part of SDL, including the the 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 difficult.

8. ACKNOWLEDGEMENTS

Manfred Broy, Christian Facchi, Max Fuchs and Rainer Weber have read preliminary versions of this paper and provided valuable feedback.

REFERENCES

1. 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.
2. M. Broy and K. Stølen. Specification and refinement of finite dataflow networks — a relational approach. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proc. FTRTFT'94, Lecture Notes in Computer Science 863*, pages 247–267, 1994.
3. ITU-TSS, editor. *Functional Specification and Description Language (SDL), Recommendation Z.100*. International Telecommunication Union, Geneva, 1993.
4. F. Dederichs. *Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen*. PhD thesis, Technische Universität München, 1992. Also available as SFB-report 342/17/92 A, Technische Universität München.
5. M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.
6. F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall, 1991.
7. E. Holz and K. Stølen. An attempt to embed a restricted version of SDL as a target language in Focus. Technical Report SFB 342/11/94 A, Technische Universität München, 1994.
8. V. Stenning. A data transfer protocol. *Computer Networks*, 1:98–110, 1976.
9. 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.