Development of a Distributed Min/Max Component*

Max Fuchs, Ketil Stølen

Abstract

We introduce a specification technique and a refinement calculus for networks of components communicating asynchronously via unbounded FIFO channels. Specifications are formulated in a relational style. The given refinement rules allow network decomposition and interface refinement. We employ the proposed formalism to specify a so-called Min/Max Component. In a step-wise fashion we refine this specification into a functional program. Finally we outline how this program can be translated into SDL.

1 Introduction

Focus [BDD+92a] is a general framework, 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 are 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 the methods presented in [Jon90], [Bac88], [Mor90] 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.

This paper presents a new style of reasoning inside the Focus framework. The objective of this paper is to explain how the proposed formalism can be employed in practical program design. It is shown in detail how an abstract requirement specification can be refined into a concrete implementation using compositional refinement techniques. After briefly introducing the main features of the method, a so-called Min/Max Component is specified and developed. The example has been carefully chosen: first of all, the resulting network is small enough to allow us to go through the whole development cycle in detail, and secondly, since it is generally accepted that program development is much simpler in the sequential than in the concurrent case, a network with a lot of communication (concurrent interference), but with almost trivial basic components (sequential processes) has been chosen. For a formal foundation of the proposed methodology see [SDW93], [BS93], [Bro92d], [Bro92a].

To show that our formalism can be combined with other methods and techniques for system development, we briefly outline a general strategy for the translation of a certain subclass of Focus specifications into SDL [CCI89]. We then apply this strategy to the low-level specification of the Min/Max Component.

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

The next section of the paper, Section 2, introduces the underlying formalism. In Section 3 it is explained what we mean by a specification. Moreover, a number of composition operators is defined and a refinement calculus is formulated. The Focus related part of the Min/Max development is handled in Section 4. Section 5 relates SDL to the Focus framework. In Section 6 it is shown how the specification of the Min/Max component can be translated into SDL. Section 7 gives a brief summary.

2 Underlying Formalism

N denotes the set of natural numbers, N^+ denotes $\mathsf{N} \setminus \{0\}$, and B denotes the set $\{true, false\}$. For any set S, $\wp(S)$ denotes the set of all nonempty subsets of S. We assume the availability of the usual logical operators and the standard set operators including min and max for sets of natural numbers. 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, i.e. it represents 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 $D^* \cup D^\infty$.

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

- ϵ denotes the empty stream;
- $\#r$ denotes the length of r, i.e. ∞ if r is infinite, and the number of elements in r otherwise;
- $\mathsf{dom}.r$ denotes N^+ if $\#r = \infty$, and $\{1, 2, \ldots, \#r\}$ otherwise;
- $r[j]$ denotes the j'th element of r if $j \in \mathsf{dom}.r$;
- $\mathsf{rng}.r$ denotes $\{r[j] | j \in \mathsf{dom}.r\}$;
- $r|_j$ denotes the prefix of r of length j if $j < \#r$, and r otherwise;
- $d \,\&\, s$ denotes the result of appending d to s;
- $r \sqsubseteq s$ denotes that r is a prefix of s.

Some of 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 j is a natural number, 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; $r[j]$ denotes the tuple consisting of the j'th action of each stream in r if $j \leq \#r$; $r|_j$ denotes the result of applying $|_j$ to each component of r; $d \,\&\, s$ denotes the result of applying $\&$ pointwisely to the components of d and s; $r \sqsubseteq s$ denotes that r is a prefix of s.

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 formula P is a safety formula iff it is prefix-closed and admissible, i.e. whenever it holds for a stream tuple s, then it also holds for any prefix of s, and whenever it holds for each element of a chain, then it also holds for the least upper bound of the chain. $\mathsf{sft}(P)$ holds iff P is a safety formula.

For formulas we need a substitution operator. Given a variable a and term t, then $P[^a_t]$ denotes the result of substituting t for every free occurrence of a in P. The operator is generalized in an obvious way in the case that a and t are lists.

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.

In this paper we will use $I_\surd$ and $O_\surd$ to range over domains of timed stream tuples, and I and O to denote their untimed counterparts. Thus I and O are the sets of untimed stream tuples generated from the timed stream tuples in $I_\surd$ and $O_\surd$, respectively, by removing all occurrences of $\surd$. For any stream i, $\diamond i$ denotes the result of removing all occurrences of $\surd$ in i. For example $\diamond(a \,\&\, b \,\&\, \surd \,\&\, a \,\&\, s) = a \,\&\, b \,\&\, a \,\&\, \diamond\, s$.

A function $\tau \in I_\surd \to O_\surd$ is called a timed stream processing function iff it is prefix continuous:

for all chains c generated from $I_\surd$: $\tau(\sqcup c) = \sqcup\{\tau(c_j) | j \in \mathsf{N}^+\}$,

and pulse driven:

for all stream tuples i in $I_\surd$: $\#i \neq \infty \Rightarrow \#i < \#\tau(i)$.

That a function is prefix continuous implies first of all that the function's behavior for infinite inputs is completely determined by its behavior for finite inputs. Secondly, prefix continuity implies prefix monotonicity which basically means 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.

That a function is pulse driven means that the length of the shortest output stream is infinite or greater than the shortest input stream. Said in a different way: the function's behavior during the first $n+1$ time intervals is completely determine by the input during the first n time intervals. This property is important for feedback constructs because it guarantees that the least fixpoint is always infinite for infinite input streams. For a detailed discussion of timed stream processing functions, see [Bro92b].

To distinguish domains of timed stream processing functions from ordinary function domains we will use $\xrightarrow{p}$ instead of $\to$ in their type declarations. We will use $\xrightarrow{c}$ to characterize domains of continuous functions.

3 Specification and Refinement

A specification of a component with n input channels and m output channels is an expression of the form

$\mathsf{spec}\ \mathrm{S} :: i : I \rhd o : O \equiv R$

where S is the specification's name; $i : I$ is a list of input identifiers (with corresponding types) representing n streams — each modeling the history of messages sent along the corresponding input channel; $o : O$ is a list of output identifiers (with corresponding types) representing m streams — each modeling the history of messages sent along the corresponding output channel; and R is a formula

with the elements of i and o as its only free variables. It is assumed that i and o are disjoint and without repetitions. We will use $\cdot$ to concatenate lists of identifiers.

The denotation of the specification S, written $[\![\text{S}]\!]$, is a set of timed stream processing functions:

$$\{\tau \in I_{\surd}^{\infty} \xrightarrow{p} O_{\surd}^{\infty} | \forall s \in I_{\surd}^{\infty} : \#s = \infty \Rightarrow R[{}^{i}_{\diamond s}\ {}^{o}_{\diamond \tau(s)}]\}$$

Thus a timed stream processing function fulfills S iff it behaves in accordance with S, for any complete input (infinite input), when ticks are abstracted away. Ticks occur only in the denotation. At the syntactic level streams are untimed, which means that they do not have occurrences of $\surd$'s. From a user's point of view our specifications are completely untimed. The reason why we use sets of timed stream processing functions instead untimed stream processing functions in the denotation is the well-known fair merge problem [Kel78]: a certain class of weakly time dependent components, like fair merge, cannot be modeled by a set of untimed stream processing functions. With the denotation used here, where we only constrain the behavior for complete (infinite) input, this is no longer a problem. The ticks allow us to distinguish the function's behavior for incomplete (finite) input from the function's behavior for complete (infinite) input. See [BS93] for a detailed discussion.

For any specification with name S we refer to its corresponding formula as R_{S}.

The operator $\otimes$ can be used to compose two specifications by connecting any output channel (strictly speaking: output stream) of the former to an identically named input channel of the latter, and by connecting any output channel of the latter to an identically named input channel of the former. For example, if $\otimes$ is used to compose the specifications S_1 and S_2 with the elements of respectively $i{\cdot}x/o{\cdot}y$ and $y{\cdot}r/x{\cdot}s$ as input/output identifiers, then the output channels denoted by y of S_1 are connected to the identically named input channels of S_2, and the output channels denoted by x of S_2 are connected to the identically named input channels of S_1, as indicated in Figure 1. The composite specification has $i{\cdot}r/o{\cdot}s$ as input/output identifiers. Thus the identifiers of the lists x and y are now hidden in the sense that they represent local channels.

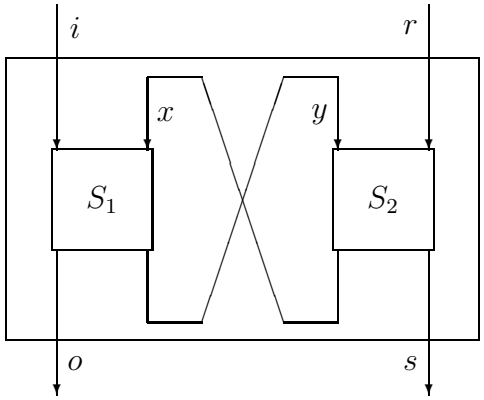

Figure 1: $S_1 \otimes S_2$.

More formally,

$$[\![S_1 \otimes S_2]\!] \stackrel{\mathsf{def}}{=} \{\tau_1 \otimes \tau_2 | \tau_1 \in [\![S_1]\!] \wedge \tau_2 \in [\![S_2]\!]\},$$

where for any pair of timed stream tuples i and r, $\tau_1 \otimes \tau_2(i, r) = (o, s)$ iff (o, s) is the least fixpoint solution with respect to i and r:

$$\begin{array}{l} \exists x, y : \forall o', y', x', s' : \\ \quad \tau_1(i, x) = (o, y) \wedge \tau_2(y, r) = (x, s) \wedge \qquad (1) \\ \quad \tau_1(i, x') = (o', y') \wedge \tau_2(y', r) = (x', s') \Rightarrow (o, y, x, s) \sqsubseteq (o', y', x', s') \qquad (2). \end{array}$$

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

When using $\otimes$ to build networks of specifications one will often experience that the operator needed is not $\otimes$, but a slight modification of $\otimes$, where for example there are no input channels corresponding to r, no output channels corresponding to o, or the channels represented by x are not hidden. Instead of introducing a new operator (and a new refinement rule) for each possible variation, we overload and use $\otimes$ for all of them, with two exceptions. To increase the readability we use $\|$ instead of $\otimes$ when there are no feedback channels, i.e. no channels corresponding to x and y in Figure 1, and ; instead of $\otimes$ in the case of sequential composition, i.e. when there are no channels corresponding to o, r and x. Whenever $\otimes$ is used it will always be clear from the context which version is the intended one. We will refer to ;, $\|$ and $\otimes$ as sequential composition, parallel composition and mutual feedback, respectively. Observe that what we call sequential composition is based on functional composition. In contrast to for example CSP the component characterized by the first specification do not have to terminate before the component characterized by the second specification starts to work. Instead, the two components work in a pipelined manner. As pictured in Figure 2 we use $\otimes_{j=1}^{n} S_j$ as a short-hand for $(\ldots((S_n \otimes S_{n-1}) \otimes S_{n-2}) \otimes \ldots \otimes S_1)$. $\|_{j=1}^{n} S_j$ is defined accordingly.

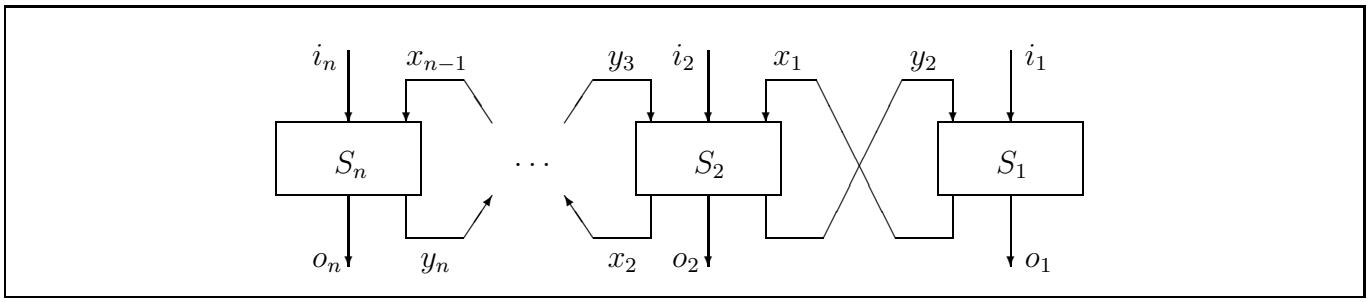

Figure 2: $\otimes_{j=1}^{n} S_j$.

A specification S_2 refines another specification S_1, written

$$S_1 \rightsquigarrow S_2$$

iff $[\![S_2]\!] \subseteq [\![S_1]\!]$. Given a requirement specification *Spec*, the goal of a system development is to construct a network of components A such that $Spec \rightsquigarrow A$ holds. The refinement relation $\rightsquigarrow$ is reflexive, transitive and a congruence w.r.t. 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.

The next step is to explain how refinements can be proved correct. We will present 9 rules altogether. All rules should be understood as follows: whenever each premise is valid, then the conclusion is valid. Thus, there is no binding between the input/output observables of two different premises.

The first three rules are easy to understand:

$$\textsf{Rule 1}: \quad \frac{S_1 \rightsquigarrow S_2 \quad S_2 \rightsquigarrow S_3}{S_1 \rightsquigarrow S_3} \qquad \textsf{Rule 2}: \quad \frac{S_1 \rightsquigarrow S_2}{S \rightsquigarrow S(S_2/S_1)} \qquad \textsf{Rule 3}: \quad \frac{R_{S_2} \Rightarrow R_{S_1}}{S_1 \rightsquigarrow S_2}$$

Rule 1 and 2 state that $\rightsquigarrow$ is transitive and a congruence. $S(S_2/S_1)$ denotes the result of substituting S_2 for one occurrence of S_1 in the network of specifications S. Rule 3 is a traditional consequence rule. It is assumed that the two specifications have the same input/output identifiers.

If S_1 and S_2 have lists of input/output identifiers as in Figure 1, then the rule for mutual feedback looks as follows:

$$\textsf{Rule 4}: \quad \frac{\begin{array}{l}\textsf{sft}(I_1) \wedge \textsf{sft}(I_2) \\ I_1[{}^{x}_{\epsilon}] \wedge I_2[{}^{y}_{\epsilon}] \\ I_1 \wedge R_{S_1} \Rightarrow I_2 \\ I_2 \wedge R_{S_2} \Rightarrow I_1 \\ I_1 \wedge R_{S_1} \wedge I_2 \wedge R_{S_2} \Rightarrow R_S\end{array}}{S \rightsquigarrow S_1 \otimes S_2}$$

Recall that $\textsf{sft}(P)$ holds if P is a safety formula. I_1 and I_2 are formulas with the elements of i, r, x and i, r, y as the only free variables (see Figure 1), respectively. This rule is closely related to the while-rule of Hoare-logic. I_1 and I_2 can be thought of as invariants. It is a well-known result that the least fixpoint of a feedback construct is equal to the least upper bound of the corresponding Kleene-chain [Kle52]. This is what fixpoint induction is based on, and this is also the idea behind Rule 4. The first, third and fourth premise imply that when the invariants hold for one element of the Kleene-chain then the invariants also hold for the next element of the Kleene chain. (Note that since our specifications only constrain the behavior for infinite inputs, this does not follow without the first premise, i.e. without the fact that I_1 and I_2 are safety formulas.) The second premise then implies that the invariants hold for all the elements of the Kleene-chain, in which case the first premise can be used to infer that the invariants hold for the least upper bound of the Kleene chain. The conclusion can now be deduced from premise five. See [SDW93], [BS93] for a more detailed discussion.

In the case of parallel and sequential composition the five premises degenerate to one premise (i.e. the special case where $I_1 = I_2 = \textsf{true}$):

$$R_{S_1} \wedge R_{S_2} \Rightarrow R_S.$$

This simplified version of Rule 4 is sufficient also in the case of the $\otimes$ operator if the overall specification holds for all fixpoints of the component specifications. If this is not the case, the full version is needed. Even in those situations where the simplified rule is sufficiently strong, the full version of Rule 4 may be helpful in the sense that it makes it easier to carry out the proof. However, this strongly depends upon the property that is to be proved.

Rule 4 can be generalized in a straightforward way to deal with more than two component specifications:

Rule 5 :

$$\frac{\begin{array}{l}\wedge_{j=1}^{n}\,\mathsf{sft}(I_j)\\ I_1[\frac{y_2}{\epsilon}] \wedge (\wedge_{j=1}^{n} I_j[\frac{x_{j-1}\ \ y_{j+1}}{\epsilon}]) \wedge I_n[\frac{x_{n-1}}{\epsilon}]\\ \wedge_{j=1}^{n} (\wedge_{k=1}^{j-1} I_k \wedge R_{S_k}) \wedge (\wedge_{k=j+1}^{n} I_k \wedge R_{S_k}) \Rightarrow I_j\\ (\wedge_{j=1}^{n} I_j \wedge R_{S_j}) \Rightarrow R_S\end{array}}{S \rightsquigarrow \otimes_{j=1}^{n} S_j}$$

The component specifications are assumed to have lists of input/output observables as in Figure 2. I_j is a formula with the elements of $i_1, \ldots, i_n$ and x_{j-1}, y_{j+1} as its only free variables. If $n = 2$ then the first premise of Rule 5 corresponds to the first premise of Rule 4, the second corresponds to the second, the third corresponds to the third and fourth, and the fourth corresponds to the fifth.

To relate the interface of one specification with the interface of another specification we also need another refinement concept, namely interface refinement, which is basically refinement modulo a representation function m, as indicated in Figure 3. This concept of refinement allows the number of input/output channels to be changed. It also supports action refinement.

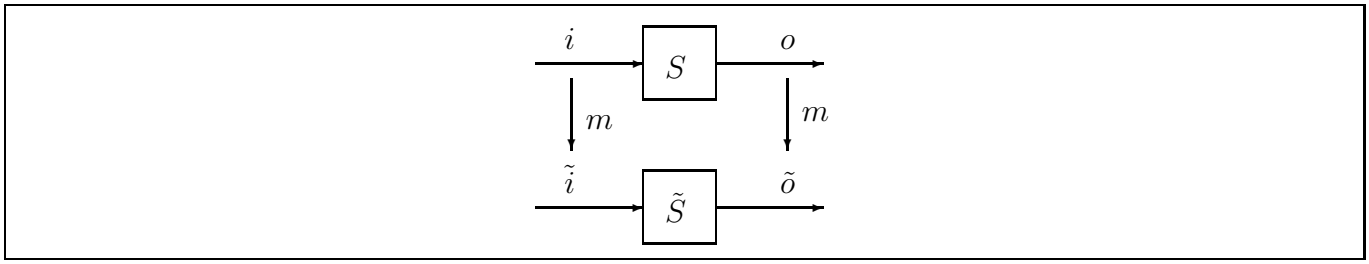

Figure 3: $S \overset{m}{\rightsquigarrow} \tilde{S}$.

We restrict ourselves to the case that all the channels at the same abstraction level are of the same type. Thus with respect to Figure 3 the components of the stream tuples i and o are of the same type, and so are also the components of the stream tuples $\tilde{i}$ and $\tilde{o}$. The rules and the refinement concept given below can easily be generalized to handle streams of different types. We also make another simplification: only the refinement of streams into tuples of streams is considered. See [Bro92d], [Bro92a] for more general refinement concepts.

A continuous function m, which maps streams to n-tuples of streams (for some $n \geq 1$) and has a continuous inverse, is said to be a representation. Any representation m is overloaded to stream tuples in a straightforward way: the result of applying a representation m to a stream tuple i is equal to the result of applying m to each component of i.

By $S \overset{m}{\rightsquigarrow} \tilde{S}$ we denote that $\tilde{S}$ is a refinement of S with respect to the representation m. More formally:

$$S \overset{m}{\rightsquigarrow} \tilde{S}$$

iff

$$\forall \tilde{q} \in [\![\ \tilde{S}\ ]\!] : \exists q \in [\![\ S\ ]\!] : \forall i : \#i = \infty \Rightarrow \diamond\tilde{q}([\![\ m\ ]\!](i)) = \diamond[\![\ m\ ]\!](q(i)),$$

where $[\![m]\!]$ is a timed stream processing function such that for all i, $m(\diamond i) = \diamond[\![m]\!](i)$.
The following rule is then sound:

Rule 6 :

$$\frac{m(i) = \tilde{i} \wedge R_{\tilde{S}} \Rightarrow \exists o \in O : R_S \wedge m(o) = \tilde{o}}{S \overset{m}{\rightsquigarrow} \tilde{S}}$$

S and $\tilde{S}$ are assumed to have i/o and $\tilde{i}/\tilde{o}$ as input/output observables, respectively. This concept of interface refinement is compositional in the sense that the following three rules are sound:

Rule 7 :

$$\frac{\begin{array}{l} S_1 \overset{m}{\rightsquigarrow} \tilde{S}_1 \\ S_2 \overset{m}{\rightsquigarrow} \tilde{S}_2 \\ \text{for all inputs } \tilde{S}_1 \otimes \tilde{S}_2 \text{ has a unique solution} \end{array}}{S_1 \otimes S_2 \overset{m}{\rightsquigarrow} \tilde{S}_1 \otimes \tilde{S}_2}$$

Rule 8 :

$$\frac{\begin{array}{l} S_1 \rightsquigarrow S_2 \\ S_2 \overset{m}{\rightsquigarrow} S_3 \end{array}}{S_1 \overset{m}{\rightsquigarrow} S_3}$$

Rule 9 :

$$\frac{\begin{array}{l} S_1 \overset{m}{\rightsquigarrow} S_2 \\ S_2 \rightsquigarrow S_3 \end{array}}{S_1 \overset{m}{\rightsquigarrow} S_3}$$

In the case of sequential and parallel composition the third premise of Rule 7 is not needed. This special case is referred to as the degenerated version of Rule 7. The concept of interface refinement and the given rules can be generalized in the style of [Bro92d], [Bro92a].

4 Design of a Min/Max Component

We want to specify and formally develop a component with two input channels ia and ib, and two output channels mn and mx, as shown in Figure 4.

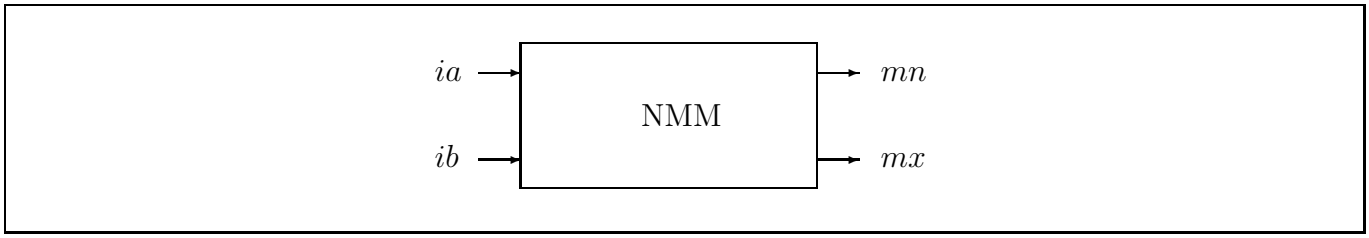

Figure 4: Min/Max Component.

For each natural number the component reads from one of its input channels, it is required to output the minimum and the maximum received so-far along mn and mx, respectively. There are no constraints on the order in which the component switches from processing inputs received on ia to processing inputs received on ib, and back again. However, it is required that all input messages eventually are read and processed. We will refer to this component as NMM (for Nondeterministic Min/Max).

To allow for an implementation where each channel is refined by a tuple of channels all of type Bit, we restrict the natural numbers received on the input channels to be less than 2^{BW}, where BW is a constant representing the band width.

Given that

$$Q \stackrel{\mathsf{def}}{=} \{0, \ldots, 2^{\mathrm{BW}} - 1\}$$

$$read : Q^\omega \times \{l, r\}^\omega \times \{l, r\} \stackrel{c}{\to} Q^\omega$$

$$read(o \,\&\, op, y \,\&\, hp, x) = \mathsf{if}\ y = x\ \mathsf{then}\ o \,\&\, read(op, hp, x)\ \mathsf{else}\ read(op, hp, x)$$

the component NMM can be formally specified as follows:

$$\mathsf{spec}\ \mathrm{NMM} :: ia, ib : Q^\omega \rhd mn, mx : Q^\omega \equiv$$

$$\begin{aligned}
&\exists h \in \{l, r\}^\omega : \exists o \in Q^\omega : \\
&\quad ia = read(o, h, l) \wedge ib = read(o, h, r) \wedge \\
&\quad \#mx = \#mn = \#ia + \#ib \wedge \\
&\quad \forall j \in \mathsf{dom}.mn : mn[j] = \mathsf{min}(\mathsf{rng}.o|_j) \wedge mx[j] = \mathsf{max}(\mathsf{rng}.o|_j)
\end{aligned}$$

The existentially quantified h is used to model the order in which the input messages are read. The existentially quantified o can be thought of as representing an internal channel in which ia and ib are fairly merged together in accordance with h (this fact is exploited when MNN is decomposed in the next section). The first two conjuncts make sure that the input channels are read fairly. The third conjunct constrains the component to process all its inputs, and the fourth conjunct requires the minimum and the maximum to be output along mn and mx, as described above.

This specification is clearly nondeterministic since the order in which the inputs are read is not determined, i.e. h is not fixed. One might think that the third conjunct is a consequence of the fourth. However, this is not the case. Without the third conjunct, the specification is for example also satisfied by a component, which as soon as it inputs a 0, outputs infinitely many 0's along mn.

Note that without the antecedent $\#s = \infty$ in the definition of $[\![\ \ ]\!]$ on Page 4, the denotation of this specification would be empty because there is no untimed stream processing function which satisfies this specification, i.e. the specification would be inconsistent in the sense that no implementation could be found.

4.1 Refinement of NMM

We start by decomposing NMM into four component specifications, as shown in Figure 5, namely into:

- FM, which fairly merges the two input streams represented by ia and ib into an output stream represented by o;
- COPY, which, as its name indicates, sends copies of the input received on o onto ri and le (for right and left);
- two specifications FILTER$_{(\mathsf{min},\mathsf{ub})}$ and FILTER$_{(\mathsf{max},\mathsf{lb})}$, where $\mathsf{ub} = 2^{\mathrm{BW}} - 1$ and $\mathsf{lb} = 0$, characterizing respectively a Min and a Max component.

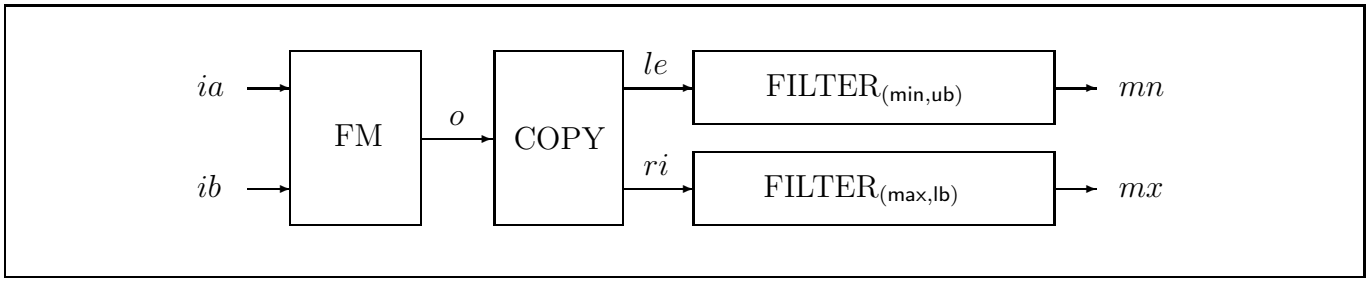

Figure 5: Refinement of the Min/Max Component.

The first one, FM, can be specified as follows:

$$\mathsf{spec}\ \mathrm{FM} :: ia, ib : Q^\omega \rhd o : Q^\omega \equiv$$

$$\exists h \in \{l, r\}^\omega : ia = read(o, h, l) \wedge ib = read(o, h, r)$$

The second component specification, COPY, is completely trivial:

$$\mathsf{spec}\ \mathrm{COPY} :: o : Q^\omega \rhd le, ri : Q^\omega \equiv$$

$$le = ri = o$$

The other two can be seen as instances of a more general formula, which we call FILTER:

$$\mathsf{spec}\ \mathrm{FILTER} :: ((m : \wp(\mathsf{N}) \to \mathsf{N}) \times init : Q) \times nw : Q^\omega \rhd out : Q^\omega \equiv$$

$$\#out = \#nw \wedge \forall j \in \mathsf{dom}.out : out[j] = m(\mathsf{rng}.nw|_j \cup \{init\})$$

FILTER has, in addition to the input observable nw and the output observable out, two additional parameters, namely a function m and a natural number $init$. The parameters m and $init$ are instantiated with min and ub in the specification characterizing the Min component, and with max and lb in the specification characterizing the Max component. The first conjunct in the specification

of FILTER restricts the number of output messages to be equal to the number of input messages. The second conjunct makes sure that the j'th output message is correctly chosen (modulo m) between the j first input messages and *init*.

The correctness of this decomposition, i.e. that

$$\text{NMM} \rightsquigarrow \text{FM}\,;\text{COPY}\,;(\text{FILTER}_{(\mathsf{min},\mathsf{ub})} \parallel \text{FILTER}_{(\mathsf{max},\mathsf{lb})}) \qquad (1)$$

follows from Rule 4 (the degenerated version) and straightforward predicate calculus.

4.2 Refinement of FILTER

The FILTER specification can be decomposed into two component specifications, REG and CP, as shown in Figure 6. REG can be interpreted as specifying a register storing the last number received on *bk*. Its initial value is *init*. Thus, REG outputs what it receives on *bk* prefixed with *init*, i.e. the initial value of the register:

$$\mathsf{spec}\ \text{REG} :: init : Q \times bk : Q^\omega \rhd od : Q^\omega \equiv$$

$$od = init \,\&\, bk$$

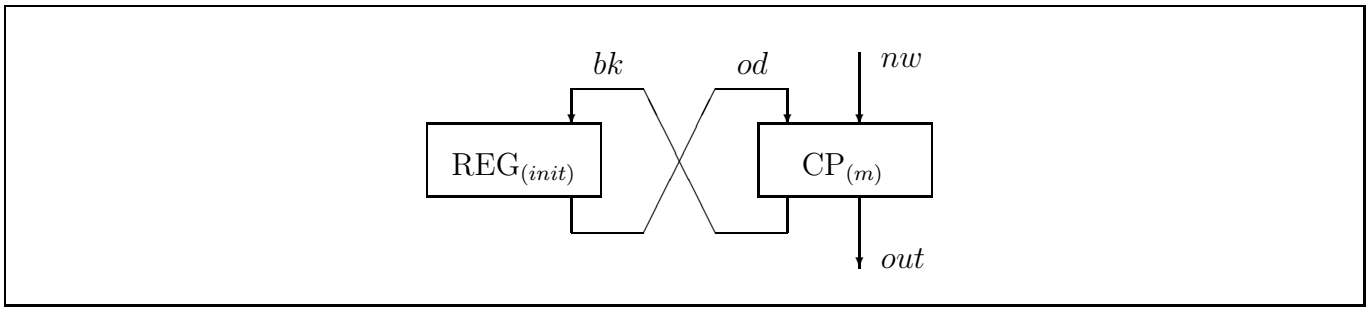

Figure 6: Refinement of the FILTER Component.

CP, on the other hand, compares a number received on *nw* with the corresponding number received on *od*. Depending on m, one of these numbers is chosen and output on both *bk* and *out*.

$$\mathsf{spec}\ \text{CP} :: (m : \wp(\mathsf{N}) \to \mathsf{N}) \times od, nw : Q^\omega \rhd bk, out : Q^\omega \equiv$$

$$bk = out \wedge \#out = \mathsf{min}(\{\#od, \#nw\}) \wedge \forall j \in \mathsf{dom}.out : out[j] = m(\{od[j], nw[j]\})$$

The first conjunct requires a message to be output along *bk* iff it is output along *out*. The second conjunct restricts any implementation to output exactly one message along *out* for each pair of messages it receives on its two input channels. The third conjunct makes sure that the correct number modulo m is chosen.

To prove that this decomposition is correct, it must be shown that

$$\mathrm{FILTER}_{(m,init)} \rightsquigarrow \mathrm{REG}_{(init)} \otimes \mathrm{CP}_{(m)} \qquad (2)$$

We will use the full version of Rule 4. Let

$$I_1 \stackrel{\mathsf{def}}{=} \forall j \in \mathsf{dom}.bk : bk[j] = m(\mathsf{rng}.nw|_j \cup \{init\}),$$

$$I_2 \stackrel{\mathsf{def}}{=} \forall j \in \mathsf{dom}.od : od[j] = m(\mathsf{rng}.nw|_{j-1} \cup \{init\}).$$

It is easy to prove that I_1 and I_2 are safety formulas. Thus Rule 4 implies that it is enough to show that

$$I_1(m, init)[^{bk}_{\epsilon}] \wedge I_2(m, init)[^{od}_{\epsilon}],$$

$$I_1(m, init) \wedge R_{\mathrm{REG}(init)} \Rightarrow I_2(m, init),$$

$$I_2(m, init) \wedge R_{\mathrm{CP}(m)} \Rightarrow I_1(m, init),$$

$$I_1(m, init) \wedge R_{\mathrm{REG}(init)} \wedge I_2(m, init) \wedge R_{\mathrm{CP}(m)} \Rightarrow R_{\mathrm{FILTER}(m,init)},$$

which follows by straightforward predicate calculus.
From (1), instantiations of (2), Rules 1 and 2 we can deduce

$$\mathrm{NMM} \rightsquigarrow \mathrm{FM}\,;\mathrm{COPY}\,;((\mathrm{REG}_{(\mathsf{ub})} \otimes \mathrm{CP}_{(\mathsf{min})}) \parallel (\mathrm{REG}_{(\mathsf{lb})} \otimes \mathrm{CP}_{(\mathsf{max})})) \qquad (3)$$

4.3 Channel Refinement

We now have a network consisting of eleven channels. Seven of these are internal and four are external. The next step is to use the rules for interface refinement to replace each of these channels with BW channels of type Bit. To relate the new observables, which represent BW-tuples of streams of bits, to the old ones, we need a function

$$ntb : Q^{\omega} \stackrel{c}{\rightarrow} (\mathsf{Bit}^{\omega})^{\mathrm{BW}}$$

defined as follows:

$ntb(\epsilon) = \epsilon,$
$ntb(n \,\&\, ns) = nat\_to\_bin(n) \,\&\, ntb(ns).$

nat_to_bin is an auxiliary function of type $Q \rightarrow \mathsf{Bit}^{\mathrm{BW}}$ such that $nat\_to\_bin(n) = b$ iff $n = \sum_{j=1}^{\mathrm{BW}} b_j * 2^{(j-1)}$, where b_1 represents the least significant bit and b_{BW} represents the most significant bit. It is straightforward to prove that ntb is continuous and has a continuous inverse.
FM can now be refined into BIN_FM, where bin_read is the obvious generalization of $read$:

spec BIN_FM :: $ia, ib : (\mathsf{Bit}^\omega)^{\mathrm{BW}} \rhd o : (\mathsf{Bit}^\omega)^{\mathrm{BW}} \equiv$

$$ia = ia|_{\#ia} \wedge ib = ib|_{\#ib}$$
$$\Rightarrow$$
$$\exists h \in \{l, r\}^\omega : ia = bin\_read(o, h, l) \wedge ib = bin\_read(o, h, r).$$

The antecedent requires the streams in the tuple represented by ia (ib) to be of equal length. Remember that when # is applied to a tuple of streams the length of the shortest stream is returned. Clearly, it follows from Rule 6 that:

$$\mathrm{FM} \overset{ntb}{\rightsquigarrow} \mathrm{BIN\_FM} \qquad (4)$$

There are trivial refinements of COPY and REG such that

$$\mathrm{COPY} \overset{ntb}{\rightsquigarrow} \mathrm{BIN\_COPY} \qquad (5)$$

$$\mathrm{REG}_{(n)} \overset{ntb}{\rightsquigarrow} \mathrm{BIN\_REG}_{(nat\_to\_bin(n))} \qquad (6)$$

CP is refined as follows:

spec BIN_CP :: $(m \in \wp(\mathsf{N}) \to \mathsf{N}) \times od, nw : (\mathsf{Bit}^\omega)^{\mathrm{BW}} \rhd bk, out : (\mathsf{Bit}^\omega)^{\mathrm{BW}} \equiv$

$$bk = out \wedge out = out|_{\#out} \wedge \#out = \mathsf{min}(\{\#od, \#nw\}) \wedge$$
$$\forall j \in \mathsf{dom}.out : out[j] = nat\_to\_bin(m(\{nat\_to\_bin^{-1}(od[j]), nat\_to\_bin^{-1}(nw[j])\}))$$

The first conjunct requires the history of the output tuple bk connecting BIN_CP to BIN_REG to be equal to the "external" output tuple out. This, the second and third conjunct makes sure that the number of messages sent along each output channel is equal to the minimum number of messages received on an input channel. The fourth conjunct takes advantage of that nat_to_bin has an inverse to state that the correct binary number modulo m is output.
As in the previous cases

$$\mathrm{CP}_{(m)} \overset{ntb}{\rightsquigarrow} \mathrm{BIN\_CP}_{(m)} \qquad (7)$$

follows from Rule 6, in which case 3 - 7 and Rule 7-8 allow us to deduce:

$$\mathrm{NMM}$$
$$\overset{ntb}{\rightsquigarrow}$$
$$\mathrm{BIN\_FM}\,;\mathrm{BIN\_COPY}\,;$$
$$((\mathrm{BIN\_REG}_{(nat\_to\_bin(\mathsf{ub}))} \otimes \mathrm{BIN\_CP}_{(\mathsf{min})}) \parallel (\mathrm{BIN\_REG}_{(nat\_to\_bin(\mathsf{lb}))} \otimes \mathrm{BIN\_CP}_{(\mathsf{max})})) \qquad (8)$$

4.4 Further Decompositions

We will now take advantage of the interface refinement of the previous section and split BIN_FM, BIN_COPY, BIN_REG and BIN_CP into networks of BW+1, BW, BW and BW + 2 specifications, respectively.

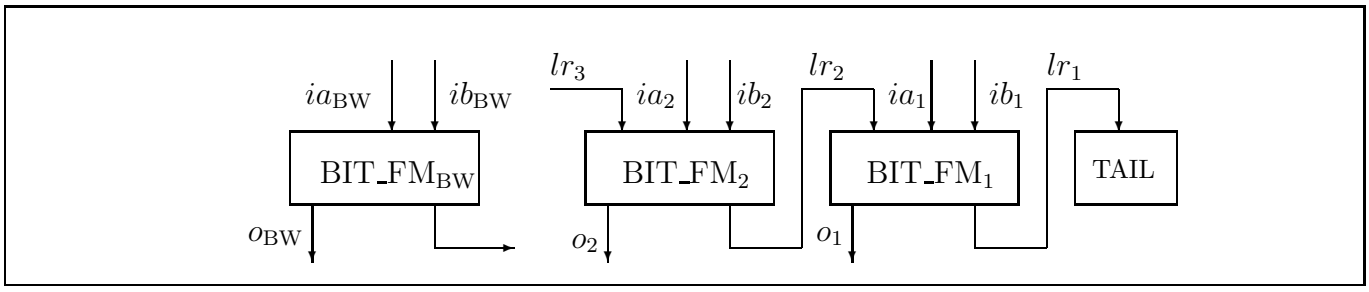

Figure 7: The bitwise Refinement of BIN_FM.

The decomposition of BIN_FM is illustrated in Figure 7. TAIL is only required to read what it receives along its input channel, and its specification is left out. $\mathrm{BIT\_FM_{BW}}$ is specified as follows:

$$\mathsf{spec}\ \mathrm{BIT\_FM_{BW}} :: ia_{\mathrm{BW}}, ib_{\mathrm{BW}} : \mathsf{Bit}^\omega \rhd o_{\mathrm{BW}} : \mathsf{Bit}^\omega \times lr_{\mathrm{BW}} : \{l,r\}^\omega \equiv$$

$$\exists h \in \{l,r\}^\omega : ia_{\mathrm{BW}} = read(o_{\mathrm{BW}}, h, l) \wedge ib_{\mathrm{BW}} = read(o_{\mathrm{BW}}, h, r) \wedge lr_{\mathrm{BW}} = h|_{\#o_{\mathrm{BW}}}.$$

On the other hand, $\mathrm{BIT\_FM}_j$, where $1 \leq j < \mathrm{BW}$, is characterized by:

$$\mathsf{spec}\ \mathrm{BIT\_FM}_j :: lr_{j+1} : \{l,r\}^\omega \times ia_j, ib_j : \mathsf{Bit}^\omega \rhd o_j : \mathsf{Bit}^\omega \times lr_j : \{l,r\}^\omega \equiv$$

$$ia_j = read(o_j, lr_{j+1}, l) \wedge ib_j = read(o_j, lr_{j+1}, r) \wedge lr_j = lr_{j+1}.$$

The role of the lr channels is to synchronize the components. When $\mathrm{BIT\_FM_{BW}}$ reads its next bit from channel ia_{BW}, $\mathrm{BIT\_FM}_j$ reads its next bit from channel ia_j.
Since

$$(\wedge_{j=1}^{\mathrm{BW}} R_{\mathrm{BIT\_FM}_j}) \wedge R_{\mathrm{TAIL}} \Rightarrow R_{\mathrm{BIN\_FM}}$$

clearly holds, it follows by the degenerated version of Rule 5 that

$$\mathrm{BIN\_FM} \rightsquigarrow \otimes_{j=1}^{\mathrm{BW}} \mathrm{BIT\_FM}_j \otimes \mathrm{TAIL} \qquad (9).$$

That BIN_CP and BIN_REG can be refined into networks such that

$$\mathrm{BIN\_COPY} \rightsquigarrow \|_{j=1}^{\mathrm{BW}} \mathrm{BIT\_COPY} \qquad (10)$$

$$\mathrm{BIN\_REG}_{(init_{\mathrm{BW}},\ldots,init_1)} \rightsquigarrow \|_{j=1}^{\mathrm{BW}} \mathrm{BIT\_REG}_{(init_j)} \qquad (11)$$

should be obvious. BIN_CP is split into a network of BW+2 specifications as shown in Figure 8.

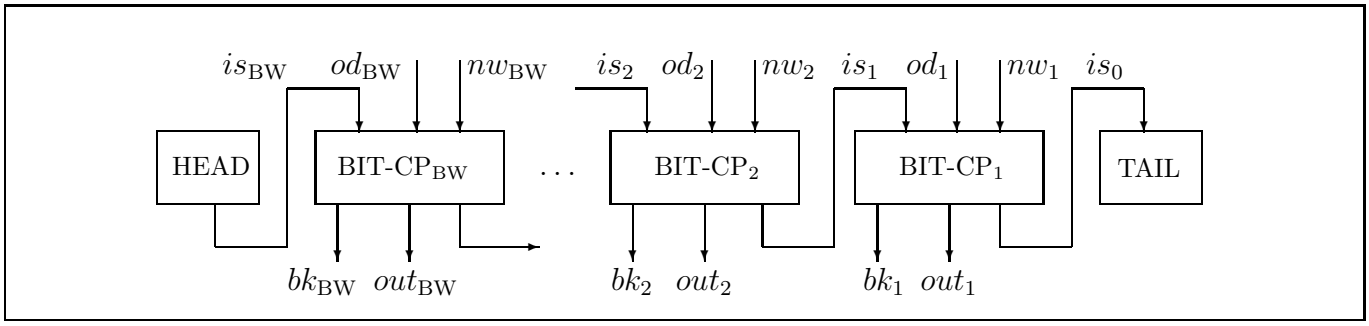

Figure 8: The bitwise refinement of BIN_CP.

If the two binary numbers received on respectively the nw channels and the od channels are equal, then the output is the same as for the previous input. On the other hand, if this is not the case, to decide whether the new binary number received on the nw channels is greater than (or alternatively, smaller than) the old maximum (minimum) received on the od channels, it is necessary to find the most significant position k (from the left to the right) in which the two binary numbers are unequal. If the digit received on this k'th nw channel, nw_k, is greater (less) than the digit received on od_k channel, then the new maximum (minimum) is the binary number received on the nw channels, otherwise the maximum (minimum) is unchanged, so the output is the same as for the previous input.

To carry out this checking from the left (the most significant bit) to the right (the least significant bit), we need an additional datatype St denoting $\{?, \mathrm{old}, \mathrm{new}\}$. ? is used to inform the next BIT_CP that to this point it is not clear whether the old or the new binary number is to be chosen. *old* and *new*, on the other hand, inform the next BIT_CP that the number to be output is the one received on *od* or *nw*, respectively.

HEAD, whose specification is trivial, is supposed to produce infinitely many ?'s.

There are of course other more efficient variants of this implementation, where for example HEAD and the most significant BIT_CP, and TAIL and the least significant BIT_CP are merged into two specifications. However, this possibility is not exploited here.

BIT_CP$_k$ can now be specified as follows:

$$\mathsf{spec}\ \mathrm{BIT\_CP}_k :: (m : \wp(\mathsf{N}) \to \mathsf{N}) \times is_k : \mathrm{St}^\omega \times od_k, nw_k : \mathsf{Bit}^\omega \rhd bk_k, out_k : \mathsf{Bit}^\omega \times is_{k-1} \in \mathrm{St}^\omega \equiv$$

$$\begin{array}{l} bk_k = out_k \wedge \#out_k = \#is_{k-1} = \mathsf{min}(\{\#od_k, \#nw_k, \#is_k\}) \wedge \\ \forall j \in \mathsf{dom}.out : (out_k[j], is_{k-1}[j]) = \\ \quad \mathsf{if}\ is_k[j] = \mathrm{old}\ \mathsf{then}\ (od_k[j], old) \\ \quad \mathsf{else\ if}\ is_k[j] = \mathrm{new}\ \mathsf{then}\ (nw_k[j], new) \\ \quad \mathsf{else\ if}\ od_k[j] = nw_k[j]\ \mathsf{then}\ (od_k[j], ?) \\ \quad \mathsf{else\ if}\ m(\{od_k[j], nw_k[j]\}) = od_k[j]\ \mathsf{then}\ (od_k[j], old) \\ \quad \mathsf{else}\ (nw_k[j], new) \end{array}$$

Since

$$\mathrm{BIN\_CP}_{(m)} \rightsquigarrow \mathrm{HEAD} \otimes (\otimes_{j=1}^{\mathrm{BW}} \mathrm{BIT\_CP}_{j_{(m)}}) \otimes \mathrm{TAIL} \qquad (12)$$

can be proved using Rule 5, i.e. by showing that

$$R_{\mathrm{HEAD}} \wedge (\wedge_{j=1}^{n} R_{\mathrm{BIT\_CP_j}(m)}) \wedge R_{\mathrm{TAIL}} \Rightarrow R_{\mathrm{BIN\_CP}(m)}.$$

The proof is once more straightforward. (8), (9), (10), (11) (12) and Rule 1, 2 and 9 allow us to deduce:

$$\begin{array}{l}
\mathrm{NMM} \\
\quad \overset{ntb}{\rightsquigarrow} \\
(\otimes_{j=1}^{\mathrm{BW}} \mathrm{BIT_FM}_j \otimes \mathrm{TAIL})\,; \\
(\|_{j=1}^{\mathrm{BW}}\ \mathrm{BIT_COPY}_j)\,; \\
(((\|_{j=1}^{\mathrm{BW}}\ \mathrm{BIT_REG}_{j_{(1)}}) \otimes (\mathrm{HEAD} \otimes (\otimes_{j=1}^{\mathrm{BW}}\ \mathrm{BIT_CP}_{j_{(\mathsf{min})}}) \otimes \mathrm{TAIL})) \| \\
\ ((\|_{j=1}^{\mathrm{BW}}\ \mathrm{BIT_REG}_{j_{(0)}}) \otimes (\mathrm{HEAD} \otimes (\otimes_{j=1}^{\mathrm{BW}}\ \mathrm{BIT_CP}_{j_{(\mathsf{max})}}) \otimes \mathrm{TAIL})))
\end{array}$$

The whole network is pictured in Figure 9, where BFM is short for BIT_FM, BCY is short for BIT_COPY, BCP is short for BIT_CP and BRG is short for BIT_REG.

4.5 Further Refinements

The final step is to transform our specifications into a more state-machine oriented form suitable for translation into SDL. We will only give two specifications, namely $\mathrm{IMP\_FM_{BW}}$ and $\mathrm{IMP\_CP}_j$ which are refinements of $\mathrm{BIT\_FM_{BW}}$ and $\mathrm{BIT\_CP}_j$, respectively. The other specifications can be refined in a similar way.

$$\begin{array}{l}
\mathsf{spec}\ \mathrm{IMP_FM_{BW}} :: ia_{\mathrm{BW}}, ib_{\mathrm{BW}} : \mathsf{Bit}^\omega \rhd o_{\mathrm{BW}} : \mathsf{Bit}^\omega \times lr_{\mathrm{BW}} : \{l,r\}^\omega \equiv \\
\quad \exists h \in \{l,r\}^\omega : \exists m \in \mathsf{Bit}^\omega : \\
\qquad ia_{\mathrm{BW}} = read(m,h,l) \wedge ib_{\mathrm{BW}} = read(m,h,r) \wedge f(m,h) = (o_{\mathrm{BW}}, lr_{\mathrm{BW}}) \\
\quad \mathsf{where}\ \forall b_1, b_2, i_1, i_2 : \\
\qquad f(b_1 \,\&\, i_1, b_2 \,\&\, i_2) = (b_1, b_2) \,\&\, f(i_1, i_2)
\end{array}$$

The function variable f occurring in the where-clause ranges over a domain of continuous functions.

$$\begin{array}{l}
\mathsf{spec}\ \mathrm{IMP_CP_k} :: (m : \wp(\mathsf{N}) \to \mathsf{N}) \times is_k : \mathrm{St}^\omega \times od_k, nw_k : \mathsf{Bit}^\omega \rhd bk_k, out_k : \mathsf{Bit}^\omega \times is_{k-1} \in \mathrm{St}^\omega \equiv \\
\quad (bk_k, out_k, is_{k-1}) = f(od_k, nw_k, is_k) \\
\quad \mathsf{where}\ \forall b_1, b_2, b_3, i_1, i_2, i_3 : \\
\qquad f(b_1 \,\&\, i_1, b_2 \,\&\, i_2, b_3 \,\&\, i_3) = \mathsf{if}\ b_3 = \mathrm{old}\ \mathsf{then}\ (b_1, b_1, old) \,\&\, f(i_1, i_2, i_3) \\
\qquad\quad \mathsf{else\ if}\ b_3 = \mathrm{new}\ \mathsf{then}\ (b_2, b_2, new) \,\&\, f(i_1, i_2, i_3) \\
\qquad\quad \mathsf{else\ if}\ b_1 = b_2\ \mathsf{then}\ (b_1, b_1, ?) \,\&\, f(i_1, i_2, i_3) \\
\qquad\quad \mathsf{else\ if}\ m(\{b_1, b_2\}) = b_1\ \mathsf{then}\ (b_1, b_1, old) \,\&\, f(i_1, i_2, i_3) \\
\qquad\quad \mathsf{else}\ (b_2, b_2, new) \,\&\, f(i_1, i_2, i_3)
\end{array}$$

Basically, this step corresponds to traditional, sequential program decomposition, and can be verified employing well-known induction techniques. For example to prove that $\mathrm{IMP\_CP}_k$ refines $\mathrm{BIT\_CP}_k$ one basically has to find an admissible formula P such that

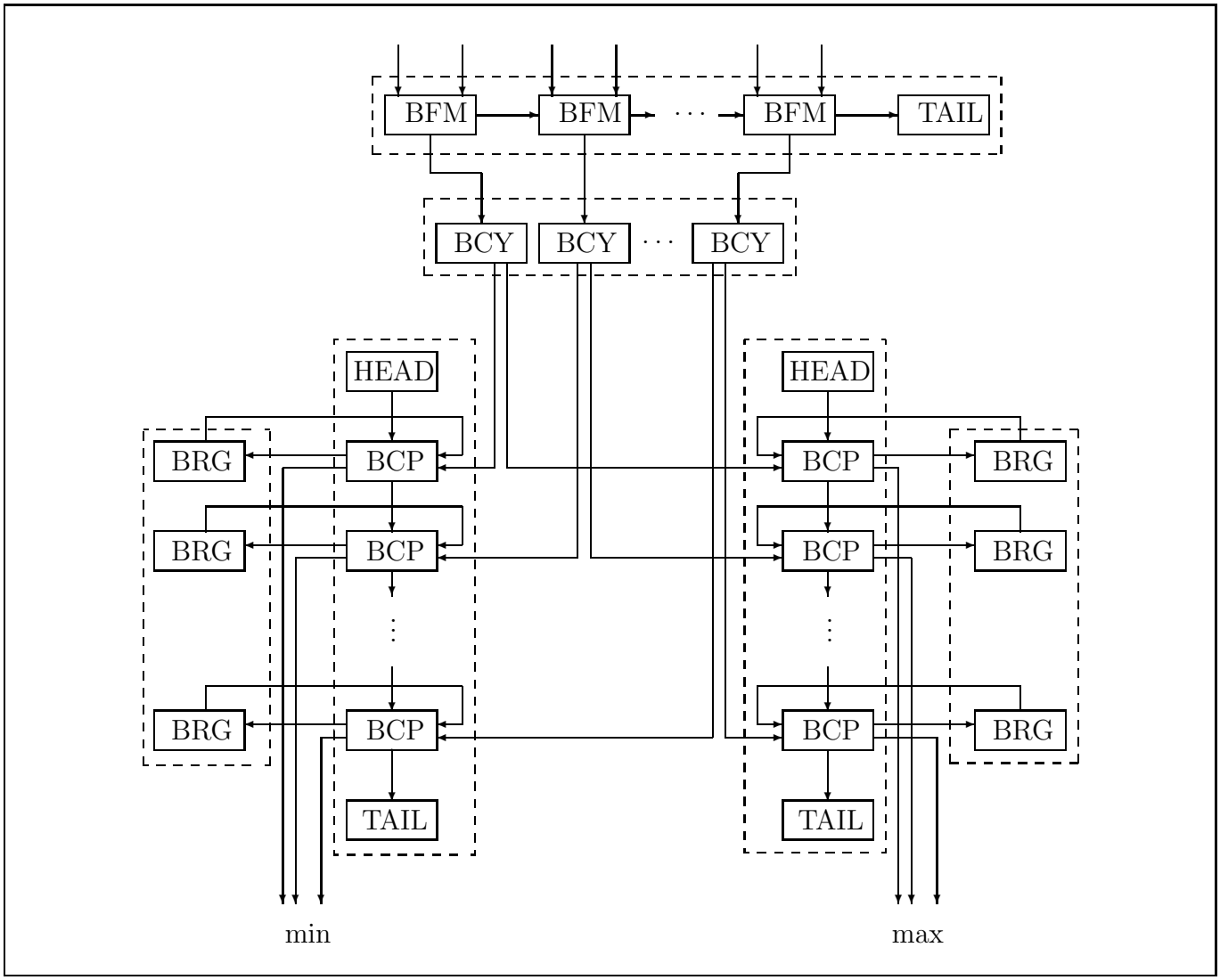

Figure 9: The bitwise Min/Max component.

$P \Rightarrow R_{\mathrm{BIT\_CP}_k}$,

and it can be proved that the function f defined in the where part of $\mathrm{IMP\_CP}_k$ is such that for any finite input tuple (i_1, i_2, i_2):

$$(\forall i'_1, i'_2, i'_3 : (i_1, i_2, i_3) \succ (i'_1, i'_2, i'_3) \Rightarrow P(i'_1, i'_2, i'_3, f(i'_1, i'_2, i'_3))) \\ \Rightarrow \\ P(i_1, i_2, i_3, f(i_1, i_2, i_3)))$$

$r \succ s$ holds if there is a v such that $v \neq \epsilon$ and $r = v^\frown s$. In fact since $R_{\mathrm{BIT\_CP}_k}$ is admissible we can define P to be equal to $R_{\mathrm{BIT\_CP}_k}$, in which case the two proof obligations follow straightforwardly. The inductive argumentation is needed in order to prove that the recursively defined $\mathrm{IMP\_CP}_k$ is a refinement of $\mathrm{BIT\_CP}_k$ which has been specified without the use of recursion.

That $\mathrm{IMP\_FM}_{\mathrm{BW}}$ refines $\mathrm{BIT\_FM}_{\mathrm{BW}}$ can be verified in a similar way.

Thus, we end up with a network of the following form:

$$
\begin{array}{l}
\mathrm{NMM} \\
\quad \overset{ntb}{\rightsquigarrow} \\
(\otimes_{j=1}^{\mathrm{BW}} \mathrm{IMP\_FM}_j \otimes \mathrm{IMP\_TAIL})\,; \\
(\|_{j=1}^{\mathrm{BW}}\ \mathrm{IMP\_COPY}_j)\,; \\
(((\|_{j=1}^{\mathrm{BW}}\ \mathrm{IMP\_REG}_{j_{(1)}}) \otimes (\mathrm{IMP\_HEAD} \otimes (\otimes_{j=1}^{\mathrm{BW}}\ \mathrm{IMP\_CP}_{j_{(\mathsf{min})}}) \otimes \mathrm{IMP\_TAIL})) \| \\
\ ((\|_{j=1}^{\mathrm{BW}}\ \mathrm{IMP\_REG}_{j_{(0)}}) \otimes (\mathrm{IMP\_HEAD} \otimes (\otimes_{j=1}^{\mathrm{BW}}\ \mathrm{IMP\_CP}_{j_{(\mathsf{max})}}) \otimes \mathrm{IMP\_TAIL})))
\end{array}
$$

5 Relating Focus to SDL

SDL [CCI89] has been developed by CCITT and was initially intended for the description of telecommunication systems. However, SDL is also suited for the description of more general systems. In SDL the behavior of a system is equal to the combined behavior of its processes. A process is an extended finite state machine, i.e. a finite state machine which, in addition to the usual control state, also has a local state which can be used to store and manipulate data. The processes communicate asynchronously by sending signals via signal routes. Each process has an infinite FIFO buffer for the storage of incoming signals. 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 (see [ST87], [FM89], [FR91] and [FS93] for an overview).

In SDL nondeterminism is not included as an explicit concept. However, as pointed out in [Bro91], there is an implicit nondeterminism in the behavior of a process caused by the interleaving of signals sent from different processes. We will call an SDL process deterministic if the interleaving of its incoming signals does not influence its behavior, and nondeterministic otherwise.

As explained in [Bro91], at the semantic level the behavior of a state in an SDL process is characterized by a function

$$\tau : loc\_state \rightarrow (I_{\surd} \xrightarrow{p} O_{\surd})$$

which for a given local state, returns a set of timed stream processing functions. The behavior of a nondeterministic process is the set of timed stream processing functions $\tau_{init}(state)$, where τ_{init} is the function characterizing the initial control state, and *state* is the initial local state.

Using this paradigm, SDL can be assigned a semantics based on stream processing functions. See [Bro91] for more details. One obvious advantage of such a semantics is that the Focus framework can be used for the development of SDL specifications. Such a development can be split into three main phases. A requirement specification is formulated in the Focus framework. Then, in a number of refinement steps the specification is refined into a low-level, executable Focus specification of a certain form. Finally, the low-level, executable Focus specification is mapped into SDL.

It is beyond the scope of this paper to syntactically characterize a sublanguage in the Focus framework which is appropriate for a mapping into SDL. However, a brief sketch will be given. Let us first relate some of the most central SDL concepts to our notation:

- **Signals:** Correspond to actions in Focus;
- **Channels, Signal Routes:** Correspond to channels modeled by streams (or by pairs of streams in the bidirectional case) in Focus;

- **Processes:** Correspond to components in Focus;
- **Timers:** Correspond to timer components in Focus (one such component for each time-out signal);
- **Blocks:** Correspond to subsystems of components in Focus;
- **Systems:** Correspond to systems of components in Focus;
- **Data definitions:** Correspond to abstract data types in Focus;
- **Decisions:** Correspond to if ... then ... else constructs in Focus;
- **Inputs:** Correspond to input actions in Focus;
- **Outputs:** Correspond to output actions in Focus.

A deterministic SDL process corresponds to a formula of the form:

$$\textsf{spec SS} :: in_1 : R_1^\omega \times \ldots \times in_n : R_n^\omega \rhd out_1 : S_1^\omega \times \ldots \times out_m : S_m^\omega \equiv$$

$$f(state)(in_1, \ldots, in_n) = (out_1, \ldots, out_m) \textsf{ where } func\_prog(f)$$

$f(state)$ models the behavior of the initial control state given that *state* is the initial local state. func_prog is an executable specification of f written in a certain normal form — basically as a set of equations with some additional constructs like if then else, let in etc. The specification IMP_CP$_k$ is for example of this structure.

With respect to the specification SS, if we restrict ourselves to the case that $m = n = 1$, such an equation would typically be of the form:

$$f_1(v)(a \,\&\, i) = \textsf{let } s = h_1(a, v) \textsf{ in } s \,\&\, f_2(h_2(a, v))(i).$$

This equation models a state transition in the following sense:

- f_1 is the name of the old control state;
- f_2 is the name of the new control state;
- v is a local state variable;
- in is the input signal, a is the current signal value and i represents the future signal values;
- h_1 is an expression with a and v as free variables which is used to assign a value to s;
- s is the signal value of the output signal *out*;
- h_2 is an expression with a and v as free variables which is used to assign a new value to the state variable v.

The left-hand side of Figure 11 shows what this equation corresponds to in SDL.

In the nondeterministic case, a specification would be of the form:

$$\textsf{spec SS} :: in_1 : R_1^\omega \times \ldots \times in_n : R_n^\omega \rhd out_1 : S_1^\omega \times \ldots \times out_m : S_m^\omega \equiv$$

$$\exists h : \exists in : \text{fair\_merge}(in_1, \ldots in_n, h, in) \wedge$$
$$f(state)(in, h) = (out_1, \ldots, out_m) \textsf{ where } func\_prog(f)$$

fair_merge is a formula characterizing that $in_1, \ldots, in_n$ are fairly merged into in in accordance with $h \in H^\omega$ where there is a one-to-one mapping from H to the identifiers $in_1, in_2, \ldots, in_n$. Strictly speaking h is not needed if the action sets $R_1, \ldots, R_n$ are disjoint. Thus fair_merge is used to model the implicit interleaving of input signals in SDL. func_prog is as before an executable specification of f. The specification $\text{IMP\_FM}_{\text{BW}}$ is of this form.

A mapping from Focus to SDL can be conducted using transformation rules. An automatic translation is also possible.

6 Translation of the Min/Max Component into SDL

Based on the outline above, we will now sketch how our executable Min/Max component can be translated into SDL.

An SDL specification consists of a system diagram, block diagrams and process diagrams. Each sequential component of our low-level specification corresponds quite naturally to a process. Here we have decided to map our specification into four block diagrams.

The system diagram is sketched in Figure 10. The channel names have been left out. Obviously, BIN_FM, BIN_COPY, BIN_MAX and BIN_MIN are blocks corresponding to the similarly named subsystems in our Focus development.

A so called leaf-block specification in SDL [BHS91] contains a set of processes that interact with each other and with the specification's environment via signal routes. BIN_FM, BIN_COPY, BIN_MAX and BIN_MIN are block specifications of this type, and the latter is sketched in Figure 11.

The block BIN_FM has one process $\text{IMP\_FM}_{\text{BW}}$ that is nondeterministic in the sense explained above. All other processes in our SDL specification are deterministic. Process diagrams representing respectively $\text{IMP\_FM}_{\text{BW}}$ and $\text{IMP\_CP}_{j(\textsf{min},1)}$ can be found in Figure 12 and 13, respectively.

7 Conclusions

A relational style for the specification and refinement of nondeterministic Kahn-networks has been introduced. Based on a simple case-study it has been shown how this formalism can be used to develop an SDL specification.

We have emphasized reasoning about communication — the type of reasoning that normally leads to complicated proofs in design/proof methods for distributed systems. In particular it has been shown that proofs about networks involving feedback can be carried out by formulating invariants in the style of a while-rule of Hoare-logic. Moreover, the close relationship between channel refinement and sequential data refinement [Hoa72] proofs is also striking. Finally, it has been explained how an executable Focus specification of a certain form can be mapped into SDL.

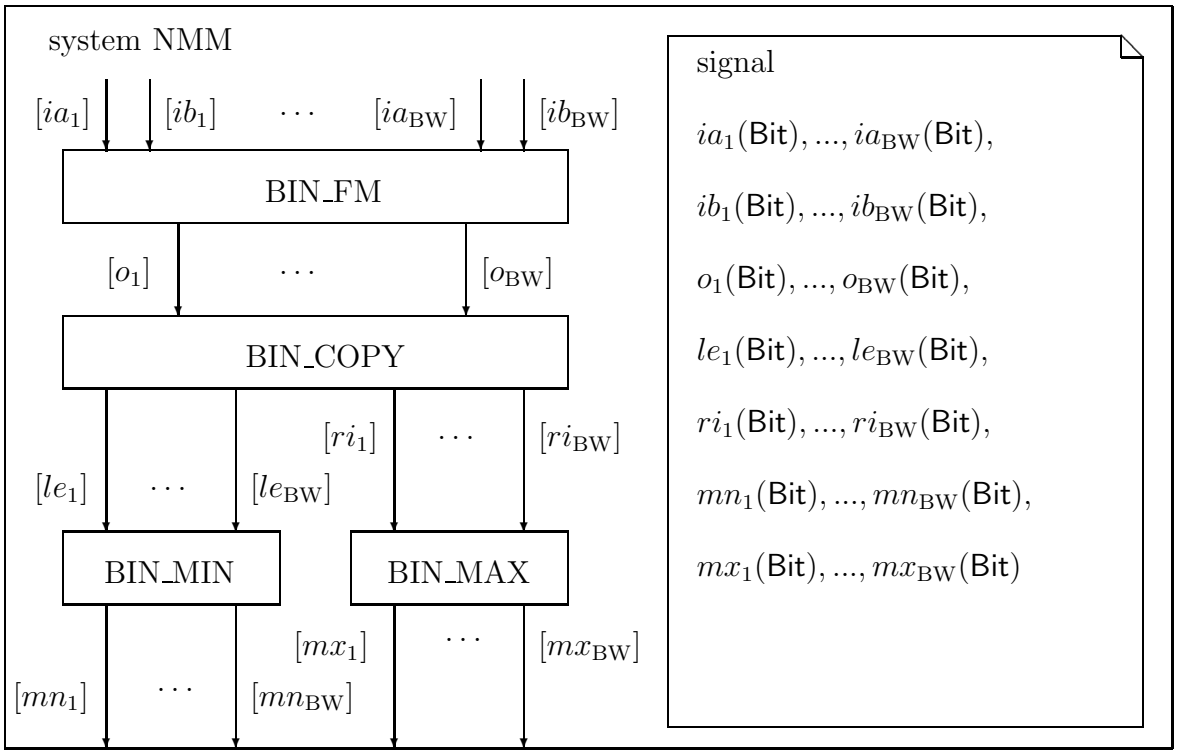

Figure 10: SDL system specification of NMM.

As already mentioned, an outline of a semantics for SDL based on timed stream processing functions can be found in [Bro91]. In [Bro92c] an SDL specification of the so-called INRES protocol, given in [Hog89], is translated into the Focus formalism. The style of reasoning employed in this paper has also been successfully used to develop a non-trivial SDL specification of a production cell [Phi93].

The advantage of the integration of SDL in the Focus framework is that since SDL specifications are not only required to characterize what is to be achieved, but also how it is to be achieved, they are often complicated and hard to understand. Thus, from an SDL user's point of view, Focus can provide an elegant formalism for the formulation of more abstract specifications, and moreover, a powerful calculus for the development of SDL specifications, which then can be further refined using already available SDL tools and methods.

A central question at this point is of course: what happens when we try to apply the same strategy to specifications of a non-trivial complexity? We believe that our technique scales up quite well for the simple reason that we conduct our reasoning at a very abstract level. For example, shared-state concurrency is hard to handle formally because of the very complicated way the different processes are allowed to interfere with each other. In our approach, we still have interference, because the different processes may communicate, but the interference is much more controlled. For an overview of other case-studies carried out in Focus, see [BDD+92b].

In this paper specifications characterize the relation between input and output streams. Because of the well-known Brock/Ackermann anomaly [BA81], to ensure relative completeness without giving up compositionality, specifications must in some situations be allowed to have an additional parameter — a so-called prophecy [Kel78] modeling the nondeterministic choices made during an execution. Thus in that case specifications can be understood as indexed sets of relations. However, the Brock/Ackermann anomaly is mainly a theoretical problem. In practical program development, it does not often occur and is therefore ignored in this paper. See [SDW93], [BS93] for a more detailed

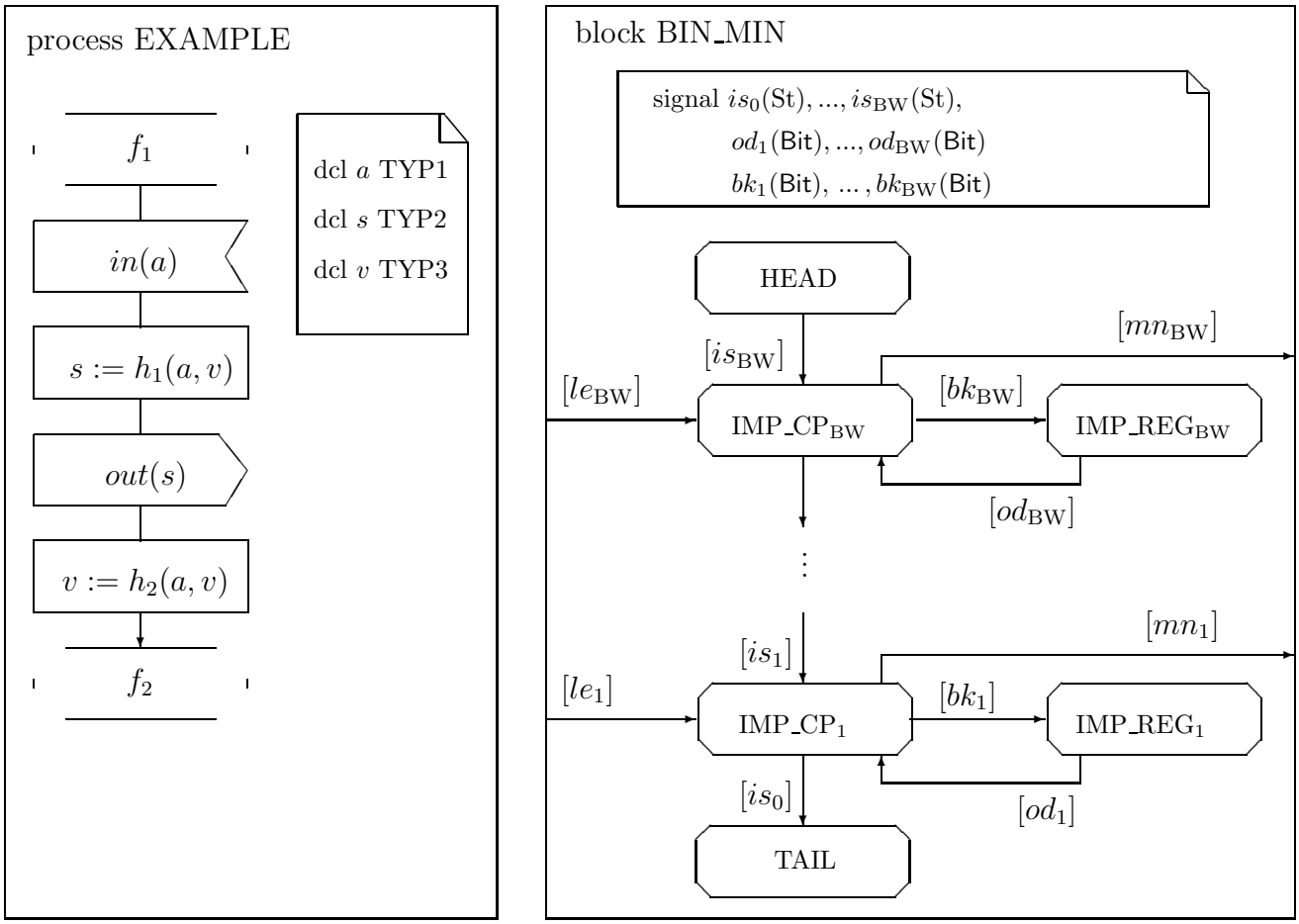

Figure 11: Equation in SDL and Block Specification of BIN_MIN.

discussion.

8 Acknowledgements

We would first of all like to thank Manfred Broy who has influenced this work in a number of ways. Detailed comments which led to many improvements have been received from Pierre Collette. Jürgen Kazmeyer and Bernard Schätz have read earlier drafts and provided valuable feedback. We have also benefited from discussions with Joachim Fischer, Eckhard Holz and Andreas Prinz.

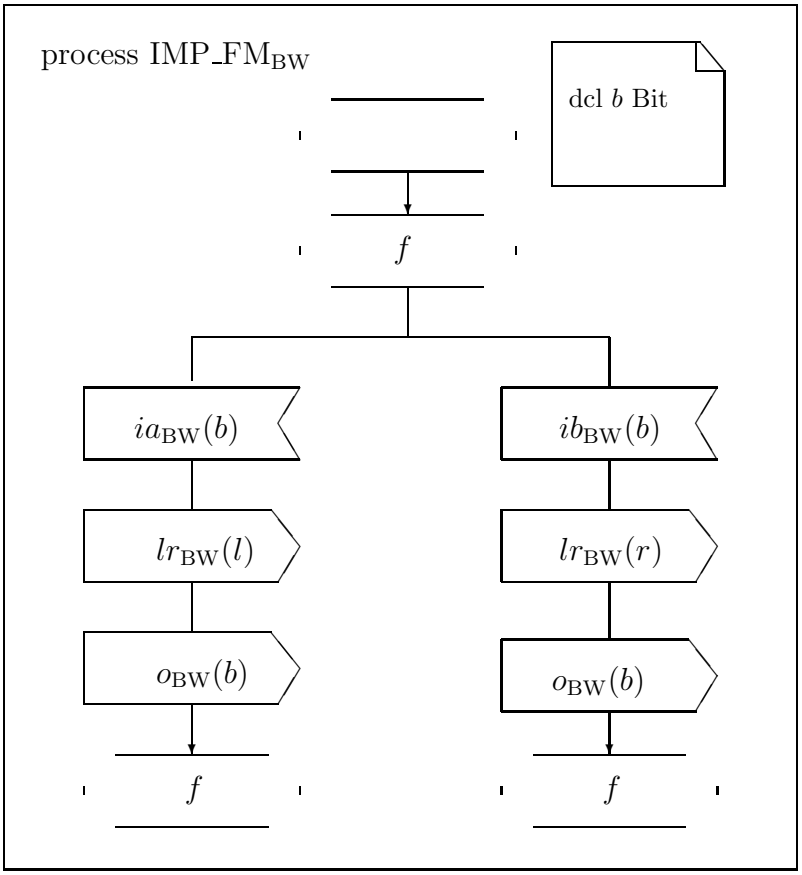

Figure 12: SDL specification of IMP_FM$_{\mathrm{BW}}$.

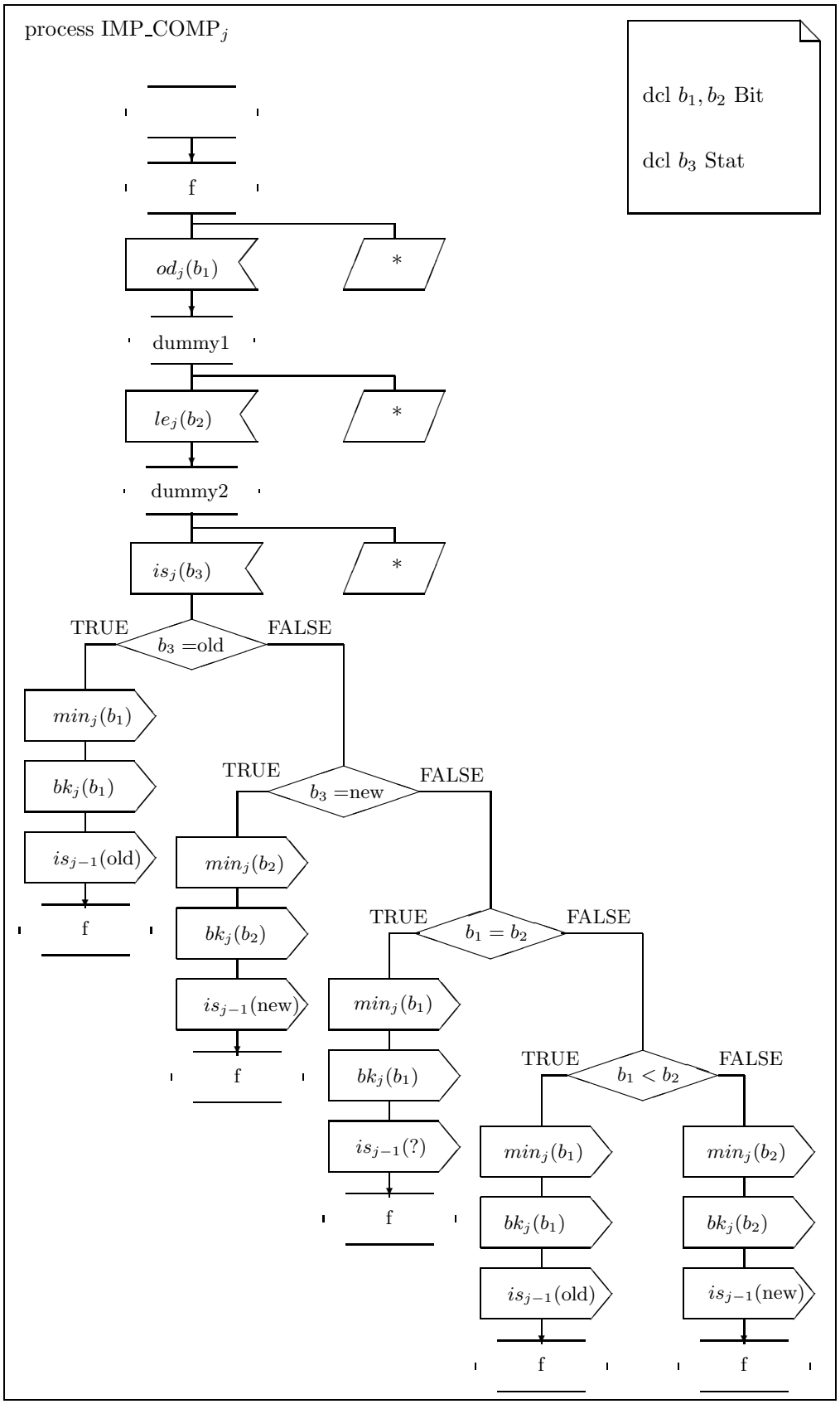

Figure 13: SDL specification of IMP_CP$_{j(\mathsf{min},1)}$

References

[BA81] J. D. Brock and W. B. Ackermann. Scenarios: A model of non-determinate computation. In J. Diaz and I. Ramos, editors, *Proc. Formalization of Programming Concepts, Lecture Notes in Computer Science 107*, pages 252–259, 1981.

[Bac88] R. J. R. Back. A calculus of refinments for program derivations. *Acta Informatica*, 25:593–624, 1988.

[BDD+92a] 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.

[BDD+92b] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. Summary of case studies in Focus - a design method for distributed systems. Technical Report SFB 342/3/92 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.

[Bro92a] M. Broy. Compositional refinement of interactive systems. Working Material, International Summer School on Program Design Calculi, August 1992.

[Bro92b] M. Broy. Functional specification of time sensitive communicating systems. In M. Broy, editor, *Proc. Programming and Mathematical Method*, pages 325–367. Springer, 1992.

[Bro92c] M. Broy. Functional system specification based on the specification and description language SDL. Manuscript, December 1992.

[Bro92d] M. Broy. (Inter-) action refinement: the easy way. Working Material, International Summer School on Program Design Calculi, August 1992.

[BS93] M. Broy and K. Stølen. Specification and design of finite dataflow networks — a relational approach. Manuscript, 1993.

[CCI89] CCITT, editor. *Functional Specification and Description Language (SDL)*. International Telecommunication Union, 1989.

[FM89] O. Faergemand and M. M. Marques, editors. *The Language at Work*. North-Holland, 1989.

[FR91] O. Faergemand and R. Reed, editors. *Evolving Methods*. North-Holland, 1991.

[FS93] O. Faergemand and A. Sarma, editors. *Using Objects*. North-Holland, 1993.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–282, 1972.

[Hog89] D. Hogrefe. *Estelle, LOTOS and SDL, Standard-Spezifikationssprachen für verteilte Systeme*. Springer, 1989.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM, Second Edition*. Prentice-Hall, 1990.

[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.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[Mor90] C. Morgan. *Programming from Specification*. Prentice-Hall, 1990.

[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.

[ST87] P. Saracco and P. Tilanus, editors. *State of the Art and Future Trands*. North-Holland, 1987.