Formal Design of a Modulo-N Counter

Max Fuchs
Institut für Informatik
Technische Universität München
D-80290 München, Arcisstr. 21
e-mail: fuchs@informatik.tu-muenchen.de

Abstract

We illustrate the use of functional system specifications and their refinement in the formal development of hardware systems by a small electronic device, an asynchronous modulo N counter. The development includes modular specification, refinement and verification. We start with an intuitive abstract requirements specification and refine this into a non-trivial concrete bit-level implementation. The refinement steps comprise behavioral, structural and interface refinement. The emphasis of this study is laid on the modeling at different levels of abstraction and the verification conditions obtained by the refinement relations between this versions.

1 Introduction

The formal design of hardware systems is a subject of remarkable interest in the area of computer science [MT90]. The complexity of electronic systems in both area and functionality requires modular specification and refinement techniques. Moreover an appropriate design method for hardware systems should offer techniques for precise and clean interface descriptions. A specification method with these properties is Focus [BDD+92]. It is based on a functional setting and modularity allows Focus to scale up quite well with specifications of non-trivial complexity. For an overview of case-studies carried out so far in Focus see [BFG+94].

To illustrate the use of Focus we choose a small, but non-trivial example, namely an asynchronous modulo N counter. On many occasions in hardware systems counters capable of counting from state 0 through state $N-1$ and then cycle back to the state 0 are needed. We refer to such counters as modulo N counters [NCI75, EP92]. There are synchronous and asynchronous modulo N counters. The synchronous ones are controlled

by a common clock signal and in general they are slower than the asynchronous versions. This paper concentrates on the formal design of an asynchronous modulo N counter. In a number of straightforward refinement steps a non-trivial bit-level implementation is refined from an intuitive abstract requirement specification.

In the specification method Focus a system is modeled by a network of functional components working concurrently, and communicating asynchronously via unbounded FIFO channels [Kah74, Del87, BDD+92]. 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. The refinement steps comprise behavioral, structural and interface refinement [Bro92]. 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.

This paper is organized as follows. In Section 2 we introduce the underlying formalism. In Section 3 it is explained, what we mean by specification and refinement. The formal design of the modulo N counter is performed in Section 4. Here we start with an intuitive abstract specification and refine it into a non-trivial network of subcomponents at the bit-level. Section 5 summarizes and draws some conclusion.

2 Underlying Formalism

$\mathbf{N}$ denotes the set of natural numbers including 0 and $\mathbf{B}$ denotes the set of binary numbers $\{0,1\}$. 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 actions 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 $j \in \mathbf{N}$, then:

- ϵ denotes the empty stream;
- $\#r$ denotes the length of r, i.e. ∞ if r is infinite, and the number of actions in r otherwise;
- $ft.r$ denotes the first action of a stream r (undefined if r is empty);
- $rt.r$ denotes the rest of stream r (r without the first action);
- $d \& s$ denotes the result of appending d to s;

- $r \frown s$ denotes the concatenation of r and s, i.e. $r \frown s$; is equal to r if r is infinite, and is equal to s prefixed with r otherwise;
- $r \sqsubseteq s$ denotes that r is a prefix of s, i.e. $\exists p \in D^\omega : r \frown p = s$;
- r^j denotes a j-tuple of streams r only.

The stream operators defined above are overloaded to tuples of streams in a straightforward way. If $d \in D$, t is an n-tuple of actions, r, s are n-tuples of streams and $j \in \{1, \ldots, n\}$, then $\#r$ denotes the length of the shortest stream in r; $t \,\&\, s$ denotes the result of applying $\&$ pointwisely to the components of t and s; $d \,\&_j\, s$ denotes the result of appending d to the j'th stream in s only; $r \frown s$ 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 $j \in \mathbf{N}$, $c_j \sqsubseteq c_{j+1}$. $\sqcup c$ denotes c's least upper bound. Since streams may be infinite such least upper bounds always exist.

A function $f \in (D^\omega)^n \to (D^\omega)^m$ is called a (n, m)-ary stream processing function iff it is monotonic which means that

for stream tuples i and i' in $(D^\omega)^n : i \sqsubseteq i' \Rightarrow f(i) \sqsubseteq f(i')$,

and continuous which means that

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

That a function is 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. Continuity, on the other hand, implies that the function's behavior for infinite inputs is completely determined by its behavior for finite inputs.

3 Specification, Refinement and Networks

A specification of an agent with input channels and output channels is written in the form

$\mathsf{spec}\ S :: g : T_1 \to T_2 \equiv F.$

S is the specification's name and g is a variable ranging over the domain of stream processing functions characterized by $T_1 \to T_2$ where T_1 and T_2 are domains of stream tuples for input and output; and F is a formula with g as its only free variable. The variable g characterizes the interface of the component we want to design. A specification's denotation $[\![S]\!]$ is the set of all stream processing functions which satisfy F.

A specification S_2 refines another specification S_1, if $[\![S_2]\!] \subseteq [\![S_1]\!]$, i.e. if any stream processing function which satisfies S_2 also satisfies S_1. We choose here the most simple and most basic logical notion of refinement for specifications, namely logical implication. If F_1 and F_2 are the corresponding formulae to S_1 and S_2, respectively, then S_1 refines S_2 iff $F_1 \Rightarrow F_2$. This refinement concept is compositional and introduced in [Bro92].

We distinguish between three different styles of refinement, namely behavioral refinement, structural refinement and interface refinement. Behavioral refinement allows to add properties and consequently restrict the number of models. Structural refinement performs a splitting of a single specification into a network of specifications. Interface refinement changes the number of input and output channels of a component as well as the granularity of the actions coming along the channels – the latter is also called action refinement.

The definition of a network of specifications is the main structuring mechanism. A network can either be defined by equational definitions or by special composition operators. In this paper we choose equational definitions to describe networks of specifications. In an equational definition a network is defined by a set of mutually recursive stream equations. The semantics is the least function that fulfills the defining equations for all input values [Ded92]. To give an example, the semantics of the network shown in Figure 1 is given by the least function, which fulfills the following equation:

$$S_1.f_1 \wedge S_2.f_2 \wedge \forall i, r \; \exists x, y, o, s : f_1(i, x) = (o, y) \wedge f_2(y, r) = (x, s)$$

Note that for different functions f_1 and f_2, which fulfil S_1 and S_2, respectively, a different fixpoint is calculated – this phenomena reflects underspecification.

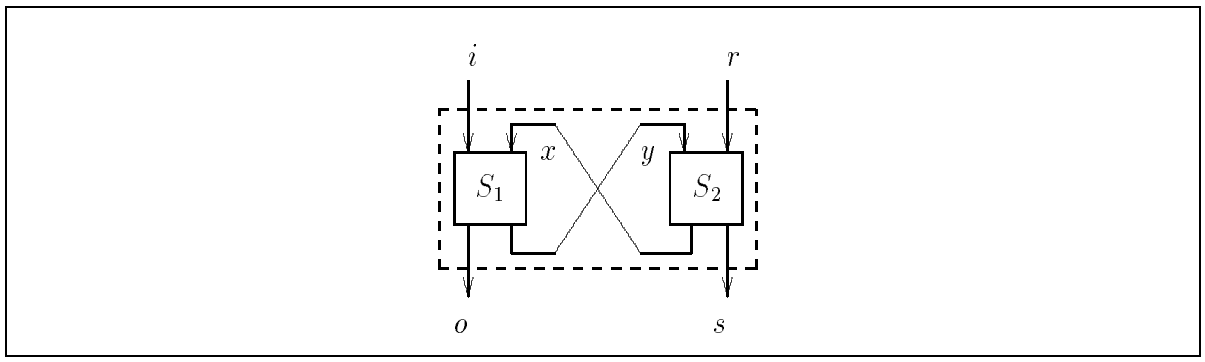

Figure 1: Mutual feedback composition of two specifications.

4 Design of a Modulo N counter

In this section we refine an abstract specification of a modulo N counter into a non-trivial bit-level implementation. Before going through the different design steps in detail we

give an overview of the entire development process. As shown in Figure 2 we start with a black box specification MNC that characterizes the external behavior of the counter.

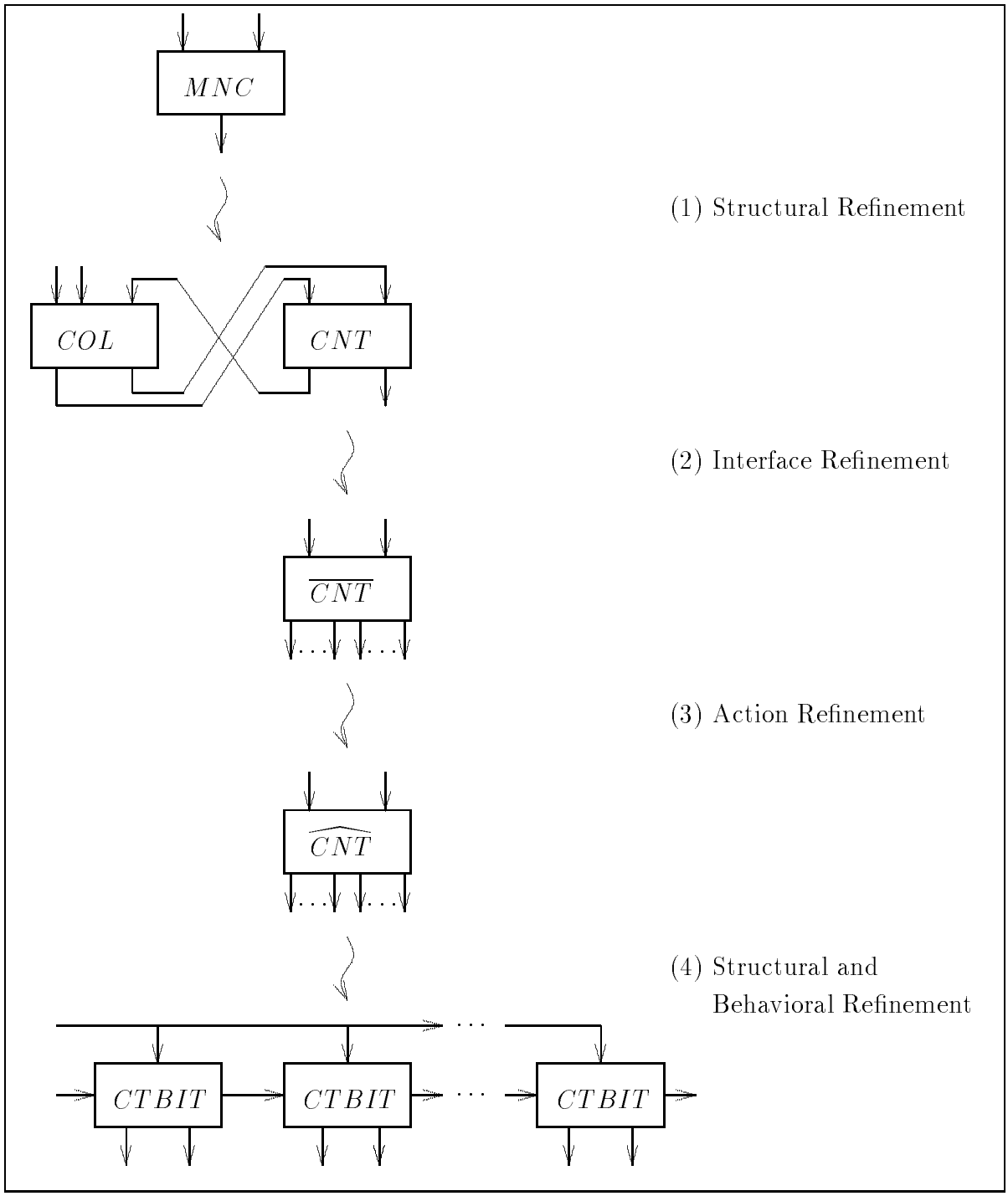

Figure 2: An overview about the design of a modulo N counter.

The input lines carry streams of bits and the output line carries natural numbers between 0 and N − 1. In the first design step (1), a so called structural refinement, the black box specification is refined into a specification of a controller COL and a counter CNT. The controller is responsible for resetting the counter whenever the counter's most recent output was N−1 and a new count signal is received. The counter itself increases or resets the output value on demand. In the following we restrict ourself to the development of the counter only. The second design step (2), a so called interface refinement, replaces each output line which carries natural numbers by an appropriate number of output lines carrying bits. The necessary number of lines is of course a function of N. The third design step (3) is a so called action refinement. To allow hardware implementations based on master/slave flipflops, where only impulses and not signals (sequence of 1's) are counted, we have to refine the bits on the input lines in an adequate way. The interesting action refinement is the refinement of a 1, which is represented by a 1 followed by a 0 – consequently a sequence of 1's is replaced by a sequence of impulses. The fourth design step (4) is a combination of a structural and a behavioral refinement step. The specification of the counter achieved during the third design step is split into a network of identical component specifications. Each specification describes a bit-slice of the counter and could be implemented by a master/slave flipflop. Note that the development of the modulo N counter would of course also include the corresponding refinement steps for the controller to ensure that both components, the controller and the counter, work properly together.

4.1 Requirement Specification

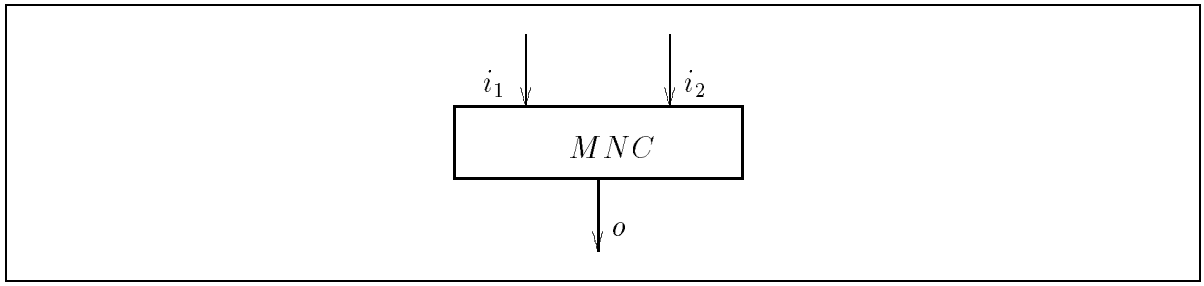

Figure 3: A modulo N counter.

We start with an abstract specification of a modulo N counter. The modulo N counter has two external input channels, which corresponds to the count (i_1) and to the clear (i_2) input, and one external output channel, which carries natural numbers $\in \{0,\ldots,\text{N}-1\}$, as indicated by Figure 3. The counter counts from state 0 through state N−1 and than cycles back to state 0. Regardless of the count input, a 1 at the clear input resets the counter to 0. On the other hand a 1 at the count input increments the counter with respect to modulo N. The formal specification is given in Figure 4.

$$
\begin{array}{l}
\textsf{spec } MNC :: f : \mathbf{B}^\omega \times \mathbf{B}^\omega \to \mathbf{N}^\omega \equiv \\
\quad \forall i_1, i_2 \in \mathbf{B}^\omega : f(i_1, i_2) = g(i_1, i_2, 0) \\
\quad \textsf{where } \forall co, cl \in \mathbf{B}^\omega, z \in \mathbf{N} : \\
\qquad g(co, 1\&cl, z) = 0\&g(rt.co, cl, 0) \\
\qquad g(0\&co, 0\&cl, z) = z\&g(co, cl, z) \\
\qquad g(1\&co, 0\&cl, z) = \textsf{ if } z = \mathrm{N} - 1 \\
\qquad\qquad\qquad\qquad \textsf{then } 0\&g(co, cl, 0) \\
\qquad\qquad\qquad\qquad \textsf{else } z + 1\&g(co, cl, z + 1)
\end{array}
$$

Figure 4: Specification of a modulo N counter.

The counter is specified in terms of a function g which has an additional state parameter to store the last output value. Whenever the signal received on the second input line, which corresponds to the clear input, is a 1, a reset is performed and 0 is output. If a 0 is received on both input lines the counter's state does not change and it's current state value, which represents the last output value, is output. When a 1 is received on the first input line, which represents the count input, and a 0 is received on the second there are two cases to consider. If the counter's state is equal to $\mathrm{N} - 1$ the counter is reset and 0 is output, otherwise the state is incremented and the incremented state value is output. Note that N is a constant which can be instantiated as needed.

4.2 First Structural Refinement

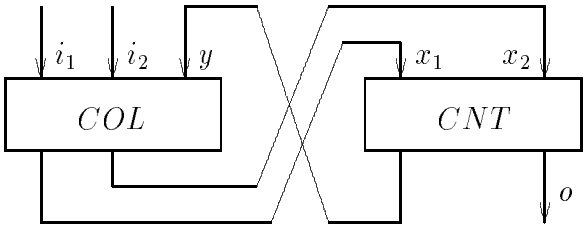

Figure 5: First structural refinement of a modulo N counter.

The first refinement step is a structural decomposition. As indicated in Figure 5, the modulo N counter MNC is decomposed into two component specifications – a controller COL and a counter CNT. The counter, which is specified in Figure 6, differs from the modulo N counter in the sense that it does not reset itself whenever a count request is received and the upper limit $\mathrm{N} - 1$ has already been reached. This task has been

transferred to the controller, which is informed about the counter's current state via y. The specification of the latter is given in Figure 7.

$$
\begin{array}{l}
\textbf{spec}\ CNT :: f : \mathbf{B}^\omega \times \mathbf{B}^\omega \to \mathbf{N}^\omega \times \mathbf{N}^\omega \equiv \\
\quad \forall x_1, x_2 \in \mathbf{B}^\omega : f(x_1, x_2) = 0 \&_1 g(x_1, x_2, 0)^2 \\
\quad \textsf{where}\ \forall co, cl \in \mathbf{B}^\omega; z \in \mathbf{N} : \\
\qquad g(co, 1\&cl, z) = 0\&g(rt.co, cl, 0) \\
\qquad g(0\&co, 0\&cl, z) = z\&g(co, cl, z) \\
\qquad g(1\&co, 0\&cl, z) = z+1\&g(co, cl, z+1)
\end{array}
$$

Figure 6: Specification of the counter CNT.

Note that the expression $g(x_1, x_2, 0)^2$ used in the specification of CNT in Figure 6 represents the tuple $(g(x_1, x_2, 0), g(x_1, x_2, 0))$ and that an initial value 0 is output on y via the $\&_1$-operator.

$$
\begin{array}{l}
\textbf{spec}\ COL :: f : \mathbf{B}^\omega \times \mathbf{B}^\omega \times \mathbf{N}^\omega \to \mathbf{B}^\omega \times \mathbf{B}^\omega \equiv \\
\quad \forall i_1, i_2 \in \mathbf{B}^\omega; y \in \mathbf{N}^\omega : \\
\quad f(i_1, 1\&i_2, y) = (0,1)\&f(rt.i_1, i_2, rt.y) \\
\quad f(0\&i_1, 0\&i_2, y) = (0,0)\&f(rt.i_1, i_2, rt.y) \\
\quad f(1\&i_1, 0\&i_2, y) = \textsf{if}\ ft.y = \mathrm{N}-1 \\
\qquad\qquad\qquad\qquad \textsf{then}\ (0,1)\&f(i_1, i_2, rt.y) \\
\qquad\qquad\qquad\qquad \textsf{else}\ (1,0)\&f(i_1, i_2, rt.y)
\end{array}
$$

Figure 7: Specification of the controller COL.

The controller only resets the counter whenever there is an external reset signal or there is a count signal and the current state of the counter is already N-1 (seen via y).

The correctness of this decomposition, i.e. that

$$
\begin{array}{l}
\forall f_1, f_2 : COL.f_1 \wedge CNT.f_2 \Rightarrow \exists f : MNC.f \wedge \forall i_1, i_2, x_1, x_2, y, o : \\
\qquad f_1(i_1, i_2, y) = (x_1, x_2) \wedge f_2(x_1, x_2) = (y, o) \Rightarrow f(i_1, i_2) = o,
\end{array}
$$

follows by fixpoint induction and straightforward predicate calculus.

Note that the fixpoint concept reflects the characteristics of feedback in communication in an appropriate manner. If further output requires more input (via feedback) than available this way no more output is generated. This is properly modeled by the assumption that the fixpoint is the least one. The initial value 0 in the specification of CNT indicates, that the counter's internal state is 0 at the very beginning and allows the calculation of an non-trivial fixpoint ($\neq \epsilon$).

In the following we restrict ourself to the development of the counter CNT only. Note that of course all interface refinements of CNT, which would affect the controller's interface, would also require refinements of COL.

4.3 Interface Refinement

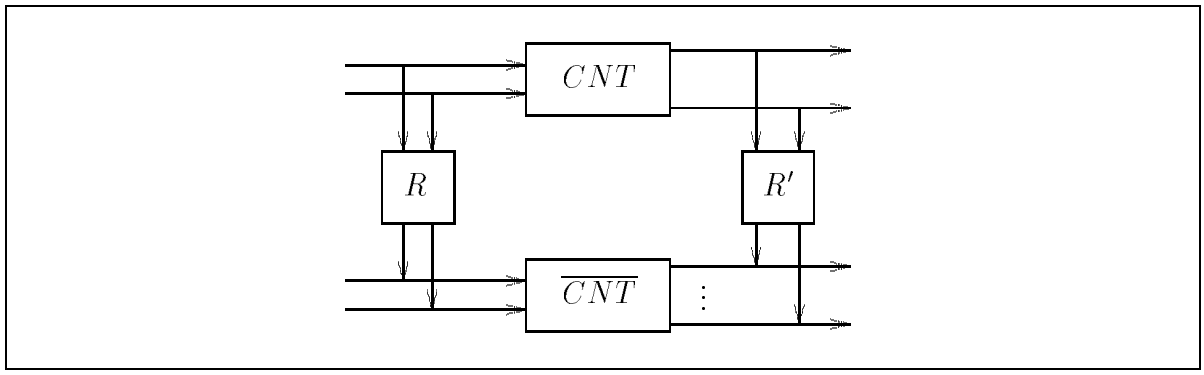

Figure 8: Interface refinement of the counter CNT.

The CNT component we have so far consists of two input channels of type **B** and two output channels of type **N**. To go a step further towards hardware implementation, we replace the channels of type **N** by an adequate number of channels of type **B**. As indicated by Figure 8, we conduct an interface refinement in the sense that the interface of the counter is refined modulo to representation specifications R and R'. More explicitly, each channel of type **N** is refined into $\tilde{\mathrm{N}}$ channels of type **B**, where $\tilde{\mathrm{N}}$ is the bandwidth required for a modulo N counter.

The auxiliary function ntb (nat-to-bin)

$$ntb :: \mathbf{N} \to \mathbf{B}^{\tilde{\mathrm{N}}}$$

$$i < \mathrm{N} \Rightarrow ntb(i) = b,$$

where iff $i = \sum_{j=1}^{\tilde{\mathrm{N}}} b_j * 2^{(j-1)}$ and b_j represents the j'th bit of the binary word b, yields the binary representation of any natural number i less than N.

The new counter specification $\overline{CNT}$, given in Figure 9, is a straightforward adaptation of CNT, and the corresponding representation specifications are trivial. The latter are given in Figure 10.

$$\textbf{spec } \overline{CNT} :: f : \mathbf{B}^\omega \times \mathbf{B}^\omega \to (\mathbf{B}^\omega)^{\tilde{\mathbf{N}}} \times (\mathbf{B}^\omega)^{\tilde{\mathbf{N}}} \equiv$$
$$\forall x_1, x_2 \in \mathbf{B}^\omega : f(x_1, x_2) = ntb(0)\&_1 g(x_1, x_2, 0)^2$$
$$\textsf{where } \forall co, cl \in \mathbf{B}^\omega; z \in \mathbf{N}$$
$$g(co, 1\&cl, z) = ntb(0)\&g(rt.co, cl, 0)$$
$$g(0\&co, 0\&cl, z) = ntb(z)\&g(co, cl, z)$$
$$g(1\&co, 0\&cl, z) = ntb(z+1)\&g(co, cl, z+1)$$

Figure 9: Specification of the counter $\overline{CNT}$.

Note that the specification $\overline{CNT}$ just differs from CNT in the sense that it simply converts the output into a bit representation by applying the auxiliary function ntb. The representation specification R' also converts natural numbers coming along the input lines into their corresponding bit representation and R simply specifies the identity function.

$$\textbf{spec } R :: f : \mathbf{B}^\omega \times \mathbf{B}^\omega \to \mathbf{B}^\omega \times \mathbf{B}^\omega \equiv$$
$$\forall i_1, i_2 \in \mathbf{B}^\omega : f(i_1, i_2) = (i_1, i_2)$$

$$\textbf{spec } R' :: f : \mathbf{N}^\omega \times \mathbf{N}^\omega \to (\mathbf{B}^\omega)^{\tilde{\mathbf{N}}} \times (\mathbf{B}^\omega)^{\tilde{\mathbf{N}}} \equiv$$
$$\forall y, o \in \mathbf{N}^\omega : f(y, o) = (ntb(ft.y), ntb(ft.o))\&f(rt.y, rt.o)$$

Figure 10: Representation specifications R and R'.

The correctness of this interface refinement, i.e. that

$\forall g_1, f_1 : R.g_1 \wedge \overline{CNT}.f_1 \Rightarrow \exists f, g : R'.g \wedge CNT.f \wedge \forall i_1, i_2 : f_1(g_1(i_1, i_2)) = g(f(i_1, i_2))$,

again follows by fixpoint induction and straightforward predicate calculus.

4.4 First Action Refinement

So far $\overline{CNT}$ already counts on every single count message, which is represented by a single "1". We now refine the counter to only count on impulses. This is closer to a hardware implementation based on master/slave flipflops. Each occurrence of a 0 and a 1 in the input streams is represented by 0&0 and 1&0, respectively. The new counter specification differs from the old one in the sense that it only assures to give the correct count value for every second output.

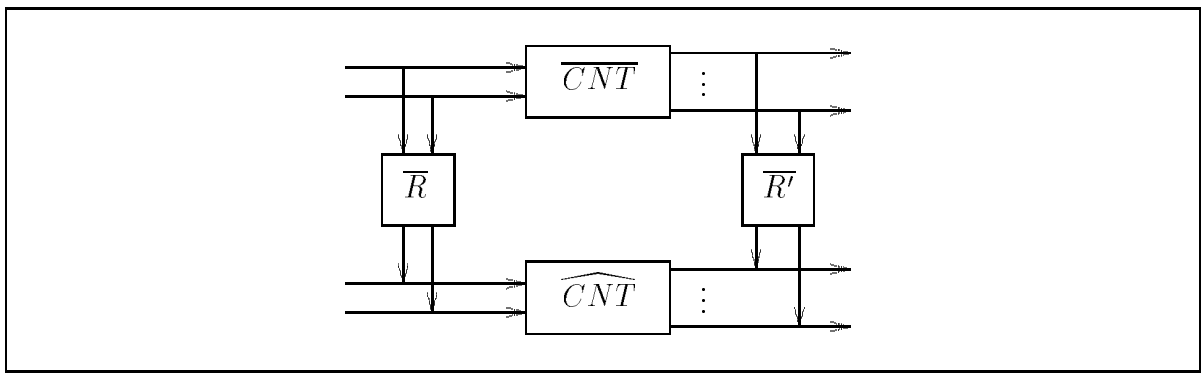

Figure 11: First action refinement.

Given the two auxiliary functions

$$\delta :: \mathbf{B}^\omega \to \mathbf{B}^\omega$$

$$\delta(0\&s) = 0\&0\&\delta(s)$$
$$\delta(1\&s) = 1\&0\&\delta(s)$$

$$\sigma :: (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}} \to (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}}$$

$$\sigma(a\&b\&c) = b\&\sigma(c),$$

we now conduct an action refinement in accordance with Figure 11. Note that δ does exactly the expansion of the input data required. The function σ allows us to take care only of every second element in a stream – this property is needed to specify $\widehat{CNT}$. The specification of $\widehat{CNT}$ is given in Figure 12.

$$\begin{array}{l}\textbf{spec } \widehat{CNT} :: f : \mathbf{B}^\omega \times \mathbf{B}^\omega \to (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}} \times (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}} \equiv \\ \quad \forall x_1, x_2 \in \mathbf{B}^\omega : \sigma.f(x_1, x_2) = ntb(0)\&_1\sigma.g(x_1, x_2, 0)^2 \\ \quad \textsf{where } \forall co, cl \in \mathbf{B}^\omega; z \in \mathbf{N} \\ \qquad g(co, 1\&cl, z) = ntb(0)\&g(rt.co, cl, 0) \\ \qquad g(0\&co, 0\&cl, z) = ntb(z)\&g(co, cl, z) \\ \qquad g(1\&co, 0\&cl, z) = ntb(z+1)\&g(co, cl, z+1)\end{array}$$

Figure 12: Specification of the counter $\widehat{CNT}$.

Note that in principle the specification $\widehat{CNT}$ is defined for arbitrary input – however if it receives input expanded by δ, it performs on every second output the same counting

CNT performs on non-expanded input. Taking only every second output value into consideration is necessary for the correctness of the next structural refinement step of section 4.5. Here $\widehat{CNT}$ still counts on every 1 whereas on 0's the output is kept stable. In the next structural refinement step however, we split up $\widehat{CNT}$ into a network of components, each working just on one bit-slice like in a master/slave flipflop implementation. In this network the counting takes only place on a 0 following immediately after a 1, which is in fact counting based on impulses. Consequently on every second output value the more abstract and the more concrete specifications correspond. On the other values however, there can be a mismatch, which is handled by underspecification via the σ-operator in the specification of $\widehat{CNT}$ – every value is allowed for the not specified output values.

Again the representation specifications are straightforward. They are given in Figure 13. $\overline{R}$ is responsible for the transformations of 1 and 0 into 1&0 and 0&0, respectively. The functions specified by $\overline{R'}$ must have the property that filtering out every second value in the output streams returns the input streams.

$$\textbf{spec } \overline{R} :: f : \mathbf{B}^\omega \times \mathbf{B}^\omega \to \mathbf{B}^\omega \times \mathbf{B}^\omega \equiv$$
$$\forall i_1, i_2 \in \mathbf{B}^\omega : f(i_1, i_2) = (\delta.i_1, \delta.i_2)$$

$$\textbf{spec } \overline{R'} :: f : (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}} \times (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}} \to (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}} \times (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}} \equiv$$
$$\forall t_1, t_2 \in (\mathbf{B}^\omega)^{\tilde{\mathrm{N}}} : \sigma.f(t_1, t_2) = (t_1, t_2)$$

Figure 13: Representation specifications $\overline{R}$ and $\overline{R'}$.

The correctness is once more straightforward. It must be shown that

$$\forall g_1, f_1 : \overline{R}.g_1 \wedge \widehat{CNT}.f_1 \Rightarrow \exists f, g : \overline{R'}.g \wedge \overline{CNT}.f \wedge \forall i_1, i_2 : \sigma.f_1(g_1(i_1, i_2)) = \sigma.g(f(i_1, i_2)).$$

4.5 Second Structural Refinement

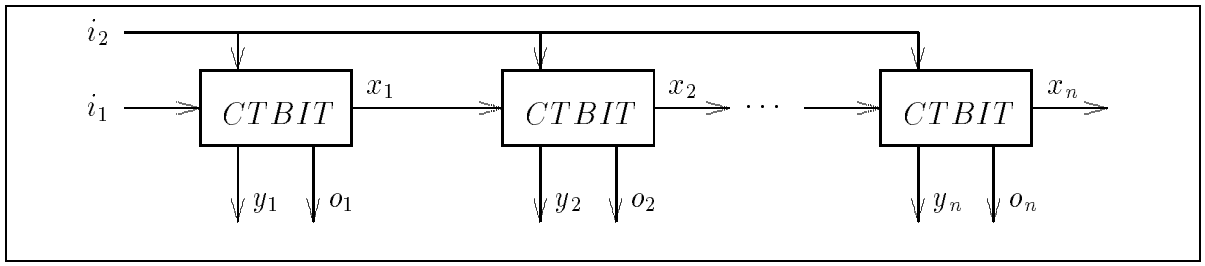

Figure 14: Second structural refinement of the counter.

The counter is now decomposed into $\tilde{\mathrm{N}}$ identical components whose specification is given in Figure 15. The specification describes a bit-slice of the counter which could be finally implemented by a master/slave flipflop.

$$
\begin{array}{l}
\textbf{spec } CTBIT :: f : \mathbf{B}^\omega \times \mathbf{B}^\omega \to \mathbf{B}^\omega \times \mathbf{B}^\omega \times \mathbf{B}^\omega \equiv \\
\quad \forall i_1, i_2 \in \mathbf{B}^\omega : f\ (i_1, i_2) = 0 \&_1 0 \&_1 g(i_1, i_2, 0, 0)^3 \\
\quad \textsf{where } \forall co, cl \in \mathbf{B}^\omega; z, o \in \mathbf{B} : \\
\qquad g(co, 1\&cl, z, o) = 0 \& g(rt.co, cl, 0, 0) \\
\qquad g(1\&co, 0\&cl, z, o) = o \& g(co, cl, 1, o) \\
\qquad g(0\&co, 0\&cl, z, o) = \textsf{ if } z = 1 \\
\qquad\qquad\qquad\qquad\qquad \textsf{then } \neg o \& g(co, cl, 0, \neg o) \\
\qquad\qquad\qquad\qquad\qquad \textsf{else } o \& g(co, cl, z, o)
\end{array}
$$

Figure 15: Specification of $CTBIT$.

The two state variables in function g represent the last read value at the count input (z) and the last output value (o), respectively. Note that a 1 at the count input does not change the current output value of $CTBIT$, but a 0 right after a 1 changes the output and finally increments the counter. To increment a bit slice means simply to invert the previous output.

The correctness of this decomposition, i.e. that

$$
\begin{array}{l}
\forall g, f_1, \ldots, f_n : \bar{R}.g \wedge (\bigwedge_{j=1}^{n} CTBIT.f_j) \Rightarrow \exists f : \widehat{CNT}(n).f \wedge \forall i_1, i_2, c, d, o_1, \ldots, o_n : \\
g(i_1, i_2) = (c, d) \Rightarrow ((\sigma.f_1(c, d) = (o_1)^3 \wedge \sigma.f_2(o_1, d) = (o_2)^3 \wedge \ldots \wedge \sigma.f_n(o_{n-1}, d) = (o_n)^3) \\
\qquad\qquad \Rightarrow \sigma.f(c, d) = (o_1, \ldots, o_n)^2),
\end{array}
$$

can be shown by induction and straightforward predicate calculus where n is a shorthand for $\tilde{\mathrm{N}}$. For the detailed proof we refer the reader to the Appendix. The refinement applied here also includes behavioral refinement. It fixes the entire output values and not only every second value as it is done in $\widehat{CNT}$.

5 Conclusion

A functional style for the formal development of an asynchronous modulo N counter has been applied. Based on this case study it has been shown that the modular design method Focus can be used to specify and stepwisely refine an intuitive abstract requirement specification of a hardware component into a non-trivial bit-level implementation. Because our method is modular and we conduct our reasoning at a very abstract level

we believe that our technique scales up quite well with specifications of a non-trivial complexity.

6 Acknowledgments

I would like to thank Ketil Stølen for fruitful discussions and valuable comments which led to many improvements. I would like to thank also Manfred Broy for a careful reading of a previous version. Financial support has been received from the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen".

References

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

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems — An Introduction to FOCUS. SFB-Bericht 342/2/92 A, Technische Universität München, January 1992.

[BFG+94] M. Broy, M. Fuchs, T. F. Gritzner, B. Schätz, K. Spies, and K. Stølen. Summary of case studies in Focus - a design method for distributed systems. Technical Report SFB 342/13/94 A, Technische Universität München, 1994.

[Bro92] M. Broy. Compositional Refinement of Interactive Systems. Technical Report 89, Digital Systems Research Center, Palo Alto, California 94301, July 1992.

[Ded92] F. Dederichs. Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen. SFB-Bericht 342/17/92 A, Technische Universität München, August 1992. Doktorarbeit.

[Del87] C. Delgado Kloos. *Semantics of Digital Circuits.* Springer, 1987. LNCS 285.

[EP92] C. Ebergen and M. G. Peeters. Modulo-N Counters: Design and Analysis of Delay-Insensitive Circuits. In J. Staunstrup and R. Sharp, editors, *Second Workshop on Designing Correct Cicuits*, January 1992.

[Fuc95] M. Fuchs. Formal Design of a Modulo-N Counter. SFB-Bericht 342/06/95 A, Technische Universität München, April 1995.

[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 *Proc. Information Processing 74*, pages 471–475. North-Holland, 1974.

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

[MT90] K. McEvoy and J. Tucker. Theoretical Foundations of Hardware Design. In K. McEvoy and J. Tucker, editors, *Theoretical Foundations of VLSI Design*, volume 10 of *Cambridge Tracts in Theoretical Computer Science*, chapter 1, pages 1 – 62. Cambridge University Press, 1990.

[NCI75] H. Nagle, B. Carroll, and J. Irwin. *An Introduction to Computer Logic.* Prentice-Hall, 1975.

7 Appendix

In this section the proof of the refinement step applied in section 4.5 is carried out in detail. Due to the fact that the two output streams of $\widehat{CNT}$ and the three output streams of any $CTBIT$ are identical for each component, we simplify these components by taking just one output stream into consideration. The corresponding proof obligation is as follows:

$$\forall g, f_1, \ldots, f_n : \bar{R}.g \wedge (\bigwedge_{j=1}^{n} CTBIT.f_j) \Rightarrow \exists f : \widehat{CNT}(n).f \wedge \forall i_1, i_2, c, d, o_1, \ldots, o_n :$$
$$g(i_1, i_2) = (c, d) \Rightarrow ((\sigma.f_1(c, d) = o_1 \wedge \sigma.f_2(o_1, d) = o_2 \wedge \ldots \wedge \sigma.f_n(o_{n-1}, d) = o_n)$$
$$\Rightarrow \sigma.f(c, d) = (o_1, \ldots, o_n))$$

The proof is carried out by induction on n:

Base case: $n = 1$

Must be shown that:

$$\forall g, f_1 : \bar{R}.g \wedge CTBIT.f_1 \Rightarrow \exists f : \widehat{CNT}(1).f \wedge \forall i_1, i_2, c, d :$$
$$g(i_1, i_2) = (c, d) \Rightarrow \sigma.f_1(c, d) = \sigma.f(c, d)$$

To show this we refer to the internal functions $\tilde{f}_1$ and $\tilde{f}$ used to define f_1 and f. The proof for the initial values, which are only covered by f_1 and f, is trivial. The functions $\tilde{f}_1$ and $\tilde{f}$ are based on states. To relate the concrete states in $\tilde{f}_1$ to the abstract state in $\tilde{f}$ we introduce a function

(1) $\quad m \in \mathbf{B}^n \times \mathbf{B}^n \rightarrow \mathbf{N}$

where

(2) $\quad \forall z, \overline{z} \in \mathbf{B}^n : m(z, \overline{z}) = btn(\overline{z})$

We then get the following proof-obligation:

(3)
$$\forall g, f_1 : \bar{R}.g \wedge CTBIT.f_1 \Rightarrow \exists f : \widehat{CNT}(1).f \wedge \forall i_1, i_2, c, d, \overline{z} :$$
$$g(i_1, i_2) = (c, d) \Rightarrow \sigma.\tilde{f}_1(c, d, 0, \overline{z}) = \sigma.\tilde{f}(c, d, m(0, \overline{z}))$$

The proof of (3) is by induction on the sum of the lengths of i_1 and i_2. The base case $\#i_1 + \#i_2 = 0$ follows trivially because of monotonicity. We assume that (3) holds for $\#i_1 + \#i_2 = n$. We want to show that (3) holds for $\#i_1 + \#i_2 = n + 1$. Assume

(4) $\quad g(i_1, i_2) = (c, d)$

There are five cases to consider:

Case 1: "count"

(5) $\quad ft.i_1 = 1 \wedge ft.i_2 = 0$

(4) and (5) imply:

(6) $\quad g(i_1, i_2) = (1\&0, 0\&0) \frown g(rt.i_1, rt.i_2)$

(4) and (6) imply:

(7) $\quad \sigma.\tilde{f}_1(c, d, 0, \overline{z}) = \sigma.\tilde{f}_1((1\&0, 0\&0) \frown g(rt.i_1, rt.i_2), 0, \overline{z})$

(7) and the definition of $\tilde{f}_1$ imply:

(8) $\quad \sigma.\tilde{f}_1(c, d, 0, \overline{z}) = \sigma.\tilde{f}_1(1\&0, 0\&0, 0, \overline{z}) \frown \sigma.\tilde{f}_1(g(rt.i_1, rt.i_2), 0, \neg\overline{z})$

(8) and induction hypothesis imply:

(9) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \sigma.\tilde{f}_1(1\&0,0\&0,0,\overline{z}) \frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(0,\neg\overline{z}))$

(9) and the definition of σ and $\tilde{f}_1$ imply:

(10) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \neg\overline{z} \frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(0,\neg\overline{z}))$

(10) and (2) imply:

(11) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = ntb(m(0,\neg\overline{z})) \frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(0,\neg\overline{z}))$

(11) and definition of $\tilde{f}$ imply:

(12) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \sigma.\tilde{f}(1\&0,0\&0,0,\overline{z}) \frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(0,\neg\overline{z}))$

(4) and (12) imply:

(13) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \sigma.\tilde{f}(c,d,m(0,\overline{z}))$

which ends the proof

□

Case 2: "no change"

(14) $ft.i_1 = 0 \wedge ft.i_2 = 0$

(4) and (14) imply:

(15) $g(i_1,i_2) = (0\&0,0\&0) \frown g(rt.i_1,rt.i_2)$

(4) and (15) imply:

(16) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \sigma.\tilde{f}_1((0\&0,0\&0) \frown g(rt.i_1,rt.i_2),0,\overline{z})$

(16), the definition of $\tilde{f}_1$ and the fact that g applied to an element yields $0\&0$ or $1\&0$ imply:

(17) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \sigma.\tilde{f}_1(0\&0,0\&0,0,\overline{z}) \frown \sigma.\tilde{f}_1(g(rt.i_1,rt.i_2),0,\overline{z})$

(17) and induction hypothesis imply:

(18) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \sigma.\tilde{f}_1(0\&0,0\&0,0,\overline{z}) \frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(0,\overline{z}))$

(18) and the definition of σ and $\tilde{f}_1$ imply:

(19) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \overline{z} \frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(0,\overline{z}))$

(19) and (2) imply:

(20) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = ntb(m(0,\overline{z})) \frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(0,\overline{z}))$

(20) and definition of $\tilde{f}$ imply:

(21) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \sigma.\tilde{f}(0\&0,0\&0,0,\overline{z}) \frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(0,\overline{z}))$

(4) and (21) imply:

(22) $\sigma.\tilde{f}_1(c,d,0,\overline{z}) = \sigma.\tilde{f}(c,d,m(0,\overline{z}))$

which ends the proof

□

Case 3: "clear"

(23) $ft.i_1 \neq \perp \wedge ft.i_2 = 1$ $(ft.i_1 = 0$ *is taken here* $-$ $ft.i_1 = 1$ *is analagous*$)$

(4) and (23) imply:

(24) $g(i_1, i_2) = (0\&0, 1\&0) \frown g(rt.i_1, rt.i_2)$

(4) and (24) imply:

(25) $\sigma.\tilde{f}_1(c, d, 0, \overline{z}) = \sigma.\tilde{f}_1((0\&0, 1\&0) \frown g(rt.i_1, rt.i_2), 0, \overline{z})$

(25) and the definition of $\tilde{f}_1$ imply:

(26) $\sigma.\tilde{f}_1(c, d, 0, \overline{z}) = \sigma.\tilde{f}_1(0\&0, 1\&0, 0, \overline{z}) \frown \sigma.\tilde{f}_1(g(rt.i_1, rt.i_2), 0, 0)$

(26) and induction hypothesis imply:

(27) $\sigma.\tilde{f}_1(c, d, 0, \overline{z}) = \sigma.\tilde{f}_1(0\&0, 1\&0, 0, \overline{z}) \frown \sigma.\tilde{f}(g(rt.i_1, rt.i_2), m(0, 0))$

(27) and the definition of σ and $\tilde{f}_1$ imply:

(28) $\sigma.\tilde{f}_1(c, d, 0, \overline{z}) = 0 \frown \sigma.\tilde{f}(g(rt.i_1, rt.i_2), m(0, 0))$

(28) and (2) imply:

(29) $\sigma.\tilde{f}_1(c, d, 0, \overline{z}) = ntb(m(0, 0)) \frown \sigma.\tilde{f}(g(rt.i_1, rt.i_2), m(0, 0))$

(29) and definition of $\tilde{f}$ imply:

(30) $\sigma.\tilde{f}_1(c, d, 0, \overline{z}) = \sigma.\tilde{f}(0\&0, 1\&0, 0, \overline{z}) \frown \sigma.\tilde{f}(g(rt.i_1, rt.i_2), m(0, 0))$

(4) and (30) imply:

(31) $\sigma.\tilde{f}_1(c, d, 0, \overline{z}) = \sigma.\tilde{f}(c, d, m(0, \overline{z}))$

which ends the proof

□

Case 4: "still waiting for i_1"

(32) $ft.i_1 = \perp \wedge ft.i_2 \neq \perp$

which is trivial because of monotonicity

□

Case 5: "still waiting for i_2"

(33) $ft.i_1 \neq \perp \wedge ft.i_2 = \perp$

which is trivial because of monotonicity

□

Induction hypothesis (Induction on n)

$$\forall g, f_1, \ldots, f_n : \bar{R}.g \wedge (\bigwedge_{j=1}^{n} CTBIT.f_j) \Rightarrow \exists f : \widehat{CNT}(n).f \wedge \forall i_1, i_2, c, d, o_1, \ldots, o_n :$$
$$g(i_1, i_2) = (c, d) \Rightarrow ((\sigma.f_1(c, d) = o_1 \wedge \sigma.f_2(o_1, d) = o_2 \wedge \ldots \wedge \sigma.f_n(o_{n-1}, d) = o_n)$$
$$\Rightarrow \sigma.f(c, d) = (o_1, \ldots, o_n))$$

Inductive case (Induction on n)

Must be shown that:

$$\forall g, f_1, \ldots, f_{n+1} : \bar{R}.g \wedge (\bigwedge_{j=1}^{n+1} CTBIT.f_j) \Rightarrow \exists f : \widehat{CNT}(n+1).f \wedge \forall i_1, i_2, c, d, o_1, \ldots, o_{n+1} :$$
$$g(i_1, i_2) = (c, d) \Rightarrow \; ((\sigma.f_1(c, d) = o_1 \; \wedge \sigma.f_2(o_1, d) = o_2 \wedge \ldots \wedge \sigma.f_{n+1}(o_n, d) = o_{n+1})$$
$$\Rightarrow \sigma.f(c, d) = (o_1, \ldots, o_{n+1}))$$

To show this we refer to internal functions $\tilde{f}_1, \ldots, \tilde{f}_{n+1}$ and $\tilde{f}$ used to define $f_1, \ldots, f_{n+1}$ and f. The proof for the initial values, which are only covered by $f_1, \ldots, f_{n+1}$ and f, is trivial. The functions $\tilde{f}_1, \ldots, \tilde{f}_{n+1}$ and $\tilde{f}$ are based on states. To relate the concrete states in $\tilde{f}_1, \ldots, \tilde{f}_{n+1}$ to the abstract state in $\tilde{f}$ we again use the function declared in (1) and (2). Assume:

(34) $\quad \overline{0} = (0, \ldots, 0)$

We then get the following proof-obligation:

$$\begin{array}{ll} & \forall g, f_1, \ldots, f_{n+1} : \bar{R}.g \wedge (\bigwedge_{j=1}^{n+1} CTBIT.f_j) \\ & \Rightarrow \exists f : \widehat{CNT}(n+1).f \wedge \forall i_1, i_2, c, d, o_1, \ldots, o_{n+1}, \overline{z}_1, \ldots, \overline{z}_{n+1} : \\ (35) & \quad g(i_1, i_2) = (c, d) \Rightarrow \\ & \quad ((\sigma.\tilde{f}_1(c, d, 0, \overline{z}_1) = o_1 \wedge \ldots \wedge \sigma.\tilde{f}_{n+1}(o_n, d, 0, \overline{z}_{n+1}) = o_{n+1}) \\ & \quad \Rightarrow \sigma.\tilde{f}(c, d, m(\overline{0}, (\overline{z}_1, \ldots, \overline{z}_{n+1}))) = (o_1, \ldots, o_{n+1})) \end{array}$$

To prove (35) we first of all show that the lemma (36) holds:

$$\begin{array}{ll} & \forall g, f_{n+1}, \tilde{f^n} : \bar{R}.g \wedge CTBIT.f_{n+1} \wedge \widehat{CNT}(n).\tilde{f^n} \\ & \Rightarrow \exists \tilde{f} : \widehat{CNT}(n+1).\tilde{f} \wedge \forall i_1, i_2, c, d, o_1, \ldots, o_{n+1}, \overline{z}_1, \ldots, \overline{z}_{n+1} : \\ (36) & \quad g(i_1, i_2) = (c, d) \Rightarrow \\ & \quad \sigma.\tilde{f}(c, d, m(\overline{0}, (\overline{z}_1, \ldots, \overline{z}_{n+1}))) = \\ & \quad \sigma.\tilde{f^n}(c, d, m((0, \ldots, 0), (\overline{z}_1, \ldots, \overline{z}_n))) \bigoplus \sigma.\tilde{f}_{n+1}(o_n, d, 0, \overline{z}_{n+1}) \end{array}$$

The proof of (36) is by induction on the sum of the lengths of i_1 and i_2. The base case $\#i_1 + \#i_2 = 0$ follows trivially because of monotonicity. We assume that (35) holds for $\#i_1 + \#i_2 = n$. We want to show that (36) holds for $\#i_1 + \#i_2 = n + 1$. Assume

(37) $\quad g(i_1, i_2) = (c, d)$

(38) $\quad \overline{Z} = (\overline{z}_1, \ldots, \overline{z}_{n+1})$

There are six cases to consider:

Case 1: "count but not transmitted beyond o_n" – $\overline{Z}$ must not be $(1, \ldots, 1, \overline{z}_{n+1})$

(39) $\quad ft.i_1 = 1 \wedge ft.i_2 = 0$

(37) and (39) imply:

(40) $\quad g(i_1, i_2) = (1\&0, 0\&0) \frown g(rt.i_1, rt.i_2)$

(37) and (40) imply:

(41) $\quad \sigma.\tilde{f}(c, d, m(\overline{0}, \overline{Z})) = \sigma.\tilde{f}((1\&0, 0\&0) \frown g(rt.i_1, rt.i_2), m(\overline{0}, \overline{Z}))$

(41) and definition of $\tilde{f}$ imply:

(42) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(1\&0,0\&0,m(\overline{0},\overline{Z}))^\frown$
$\sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(\overline{0},ntb(btn(\overline{Z})+1)))$

We assume:

(43) $ntb(btn(\overline{Z})+1) = (\overline{y}_1,\ldots,\overline{y}_{n+1})$

(43) and (42) and Induction hypothesis imply:

(44) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(1\&0,0\&0,m(\overline{0},\overline{Z}))^\frown$
$\sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(\overline{y}_1,\ldots,\overline{y}_n))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\overline{z}_{n+1})$

(44) and definition of $\tilde{f}$ and of $\bigoplus$ imply:

(45) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = (\overline{y}_1,\ldots,\overline{y}_n)\bigoplus(\overline{z}_{n+1})^\frown$
$\sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(\overline{y}_1,\ldots,\overline{y}_n))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\overline{z}_{n+1})$

(45) and definition of m and ntb imply:

(46) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = ntb(m((0,\ldots,0),(\overline{y}_1,\ldots,\overline{y}_n)))\bigoplus(\overline{z}_{n+1})^\frown$
$\sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(\overline{y}_1,\ldots,\overline{y}_n))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\overline{z}_{n+1})$

(46), (39) and definition of $\tilde{f^n}$ imply:

(47) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f^n}(1\&0,0\&0,m((0,\ldots,0),(\overline{z}_1,\ldots,\overline{z}_n))\bigoplus(\overline{z}_{n+1})^\frown$
$\sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(\overline{y}_1,\ldots,\overline{y}_n))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\overline{z}_{n+1})$

(47), (40) and (37) and the definition of $\bigoplus$ imply:

(48) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z}))$

which ends the proof

□

Case 2: "count but transmitted beyond o_n" – $\overline{Z}$ has to be $(1,\ldots,1,\overline{z}_{n+1})$

(49) $ft.i_1 = 1 \wedge ft.i_2 = 0$

(37) and (49) imply:

(50) $g(i_1,i_2) = (1\&0,0\&0)^\frown g(rt.i_1,rt.i_2)$

(37) and (50) imply:

(51) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}((1\&0,0\&0)^\frown g(rt.i_1,rt.i_2),m(\overline{0},\overline{Z}))$

(51) and definition of $\tilde{f}$ imply:

(52) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(1\&0,0\&0,m(\overline{0},\overline{Z}))^\frown$
$\sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(\overline{0},(0,\ldots,0,\neg\overline{z}_{n+1})))$

Induction hypothesis implies:

(53) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(1\&0,0\&0,m(\overline{0},\overline{Z}))^\frown$
$\sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(0,\ldots,0))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\neg\overline{z}_{n+1})$

(53) and definition of $\tilde{f}$ and of $\bigoplus$ imply:

(54) $$\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = (0,\ldots,0)\bigoplus(\neg\overline{z}_{n+1})^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(0,\ldots,0)))\bigoplus\sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\neg\overline{z}_{n+1})\end{array}$$

(54) and definition of m and ntb imply:

(55) $$\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = ntb(m((0,\ldots,0),(0,\ldots,0)))\bigoplus(\neg\overline{z}_{n+1})^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(0,\ldots,0)))\bigoplus\sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\neg\overline{z}_{n+1})\end{array}$$

(55), (49) and definition of $\tilde{f^n}$ imply:

(56) $$\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f^n}(1\&0,0\&0,m((0,\ldots,0),(1,\ldots,1)))\bigoplus(\neg\overline{z}_{n+1})^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(0,\ldots,0)))\bigoplus\sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\neg\overline{z}_{n+1})\end{array}$$

(56), (50) and (37) and the definition of $\bigoplus$ imply:

(57) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z}))$

which ends the proof □

Case 3: "no change"

(58) $ft.i_1 = 0 \wedge ft.i_2 = 0$

(37) and (58) imply:

(59) $g(i_1,i_2) = (0\&0,0\&0)^\frown g(rt.i_1,rt.i_2)$

(37) and (59) imply:

(60) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}((0\&0,0\&0)^\frown g(rt.i_1,rt.i_2),m(\overline{0},\overline{Z}))$

(60) and definition of $\tilde{f}$ imply:

(61) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(0\&0,0\&0,m(\overline{0},\overline{Z}))^\frown\sigma.\tilde{f}(g(rt.i_1,rt.i_2),m(\overline{0},\overline{Z}))$

Induction hypothesis implies:

(62) $$\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(0\&0,0\&0,m(\overline{0},\overline{Z}))^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(\overline{z}_1,\ldots,\overline{z}_n)))\bigoplus\sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\overline{z}_{n+1})\end{array}$$

(62) and definition of $\tilde{f}$ and of $\bigoplus$ imply:

(63) $$\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = (\overline{z}_1,\ldots,\overline{z}_n)\bigoplus(\overline{z}_{n+1})^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(\overline{z}_1,\ldots,\overline{z}_n)))\bigoplus\sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\overline{z}_{n+1})\end{array}$$

(63) and definition of m and ntb imply:

(64) $$\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = ntb(m((0,\ldots,0),(\overline{z}_1,\ldots,\overline{z}_n)))\bigoplus(\overline{z}_{n+1})^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(\overline{z}_1,\ldots,\overline{z}_n)))\bigoplus\sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\overline{z}_{n+1})\end{array}$$

(64), (58) and definition of $\tilde{f^n}$ imply:

(65) $$\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f^n}(0\&0,0\&0,m((0,\ldots,0),(\overline{z}_1,\ldots,\overline{z}_n)))\bigoplus(\overline{z}_{n+1})^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2),m((0,\ldots,0),(\overline{z}_1,\ldots,\overline{z}_n)))\bigoplus\sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,\overline{z}_{n+1})\end{array}$$

(65), (59) and (37) and the definition of $\bigoplus$ imply:

(66) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z}))$

which ends the proof

□

Case 4: "clear"

(67) $ft.i_1 \neq \perp \wedge ft.i_2 = 1$ $(ft.i_1 = 0$ *is taken here* $-$ $ft.i_1 = 1$ *is analagous*$)$

(37) and (67) imply:

(68) $g(i_1,i_2) = (0\&0, 1\&0)^\frown g(rt.i_1, rt.i_2)$

(37) and (68) imply:

(69) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}((0\&0,1\&0)^\frown g(rt.i_1,rt.i_2), m(\overline{0},\overline{Z}))$

(69) and definition of $\tilde{f}$ imply:

(70) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(0\&0,1\&0,m(\overline{0},\overline{Z}))^\frown \sigma.\tilde{f}(g(rt.i_1,rt.i_2), m(\overline{0},\overline{0}))$

Induction hypothesis implies:

(71) $\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(0\&0,1\&0,m(\overline{0},\overline{Z}))^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2), m((0,\ldots,0),(0,\ldots,0))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,0)\end{array}$

(71) and definition of $\tilde{f}$ and of $\bigoplus$ imply:

(72) $\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = (0,\ldots,0)\bigoplus(0)^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2), m((0,\ldots,0),(0,\ldots,0))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,0)\end{array}$

(72) and definition of m and ntb imply:

(73) $\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = ntb(m((0,\ldots,0),(0,\ldots,0)))\bigoplus(0)^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2), m((0,\ldots,0),(0,\ldots,0))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,0)\end{array}$

(73), (67) and definition of $\tilde{f^n}$ imply:

(74) $\begin{array}{l}\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f^n}(0\&0,1\&0,m((0,\ldots,0),(\overline{z}_1,\ldots,\overline{z}_n)))\bigoplus(0)^\frown \\ \sigma.\tilde{f^n}(g(rt.i_1,rt.i_2), m((0,\ldots,0),(0,\ldots,0))) \bigoplus \sigma.\tilde{f}_{n+1}(g(rt.o_n,rt.i_2),0,0)\end{array}$

(74), (68) and (37) and the definition of $\bigoplus$ imply:

(75) $\sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z})) = \sigma.\tilde{f}(c,d,m(\overline{0},\overline{Z}))$

which ends the proof

□

Case 5: "still waiting for i_1"

(76) $ft.i_1 = \perp \wedge ft.i_2 \neq \perp$

which is trivial because of monotonicity

□

Case 6: "still waiting for i_2"

(77) $ft.i_1 \neq \perp \wedge ft.i_2 = \perp$

which is trivial because of monotonicity

□

Now we want to show (35) based on the lemma (36). We assume

(78) $g(i_1, i_2) = (c, d)$

(79) $\sigma.\tilde{f}_1(c, d, 0, \overline{z}_1) = o_1 \wedge \ldots \wedge \sigma.\tilde{f}_{n+1}(o_n, d, 0, \overline{z}_{n+1}) = o_{n+1}$

Now we have to proof

(80) $\sigma.\tilde{f}(c, d, m(\overline{0}, \overline{Z})) = (o_1, \ldots, o_{n+1})$

(78), (79) and (80) imply:

(81) $\sigma.\tilde{f}(c, d, m(\overline{0}, \overline{Z})) = (\sigma.\tilde{f}_1(c, d, 0, \overline{z}_1), \ldots, \sigma.\tilde{f}_{n+1}(o_n, d, 0, \overline{z}_{n+1}))$

definition of $\bigoplus$ implies:

(82) $\sigma.\tilde{f}(c, d, m(\overline{0}, \overline{Z})) = (\sigma.\tilde{f}_1(c, d, 0, \overline{z}_1), \ldots, \sigma.\tilde{f}_n(o_{n-1}, d, 0, \overline{z}_n)) \bigoplus$
$\sigma.\tilde{f}_{n+1}(o_n, d, 0, \overline{z}_{n+1})$

Induction Hypothesis implies:

(83) $\sigma.\tilde{f}(c, d, m(\overline{0}, \overline{Z})) = \sigma.\tilde{f^n}(c, d, m((0, \ldots, 0), (\overline{z}_1, \ldots, \overline{z}_n))) \bigoplus$
$\sigma.\tilde{f}_{n+1}(o_n, d, 0, \overline{z}_{n+1})$

which holds because of (36)

□