On the Formal Derivation of a FEAL Microprocessor

R. Rukšėnas*   K. Sere†   Y. Zhao*

Abstract

We present an outline of a method for *formal* derivation of asynchronous VLSI circuits. The proposed method focuses on transformational style of the design and it uses techniques familiar from the construction of parallel programs. *Refinement calculus* and *action systems* are used as a framework for the design process. As a case study we look at the derivation of an asynchronous encryption/decryption microprocessor.

1 Introduction

The paper describes ongoing work on exploring a methodology for *formal* derivation of asynchronous delay-insensitive VLSI circuits within the *refinement calculus* and the *action system* framework. It is aimed to be used in the design of *application-specific* circuits. The basic idea is to apply techniques familiar from the construction of parallel programs to VLSI design. This approach was originally taken by Martin [9] who has developed a methodology for designing asynchronous VLSI circuits as concurrent programs within the CSP-framework. Using his method he has specified and derived a number of nontrivial circuits. Delay-insensitive, asynchronous chips were then produced from the concurrent programs that can be seen as equivalent to the parallel composition of action systems that we derive here. However, the method of Martin is semiformal and the transformations carried out during the design process are not formally proved to be correct. We want to provide a completely formal basis for Martin's method.

An *action system* is a parallel or distributed program where parallel activity is described in terms of events, so called actions. Several actions can be executed in parallel, as long as the actions do not share any variables. A recent extension of the action system framework, adding procedure declarations to action systems [5], gives us a very general mechanism for communication between action systems. The action systems formalism was proposed by Back and Kurki-Suonio [3].

The *refinement calculus* is a formalization of the stepwise refinement method of program construction. It was originally proposed by Back [1] and it has been later studied and extended by several researchers, see [11, 12] among others.

*Åbo Akademi University, Department of Computer Science, FIN-20520 Turku, Finland, emails: {rruksena,zyi}@aton.abo.fi

†University of Kuopio, Department of Computer Science and Applied Mathematics, FIN-70211 Kuopio, Finland, email: Kaisa.Sere@uku.fi, fax: +358-71-162595

Originally, the refinement calculus was designed as a framework for systematic derivation of sequential programs only. Back and Sere [4, 13] extended the refinement calculus for the design of action systems and hence, it was possible to handle *parallel algorithms* within the calculus. Back [2] made yet another extension to the calculus showing how *reactive programs* could be derived in a stepwise manner within it relying heavily on the work done in data refinement.

Action systems and the refinement calculus approach has already proved to be suitable for VLSI circuit specification and design on logical behaviour level [6]. Now we want to explore its applicability in the design on rather low (signal) level.

The refinement calculus has been also formalised within a mechanical theorem prover by Back and von Wright [7]. This gives us the tool to mechanise the design process and prove correctness of the derived circuit. A somewhat related method and formalism is developed in [14], but the emphasis is put on the verification of and formal models for delay-insensitive circuits.

As a case study we look at the design of an asynchronous FEAL (Fast Encipherment Algorithm) processor [10]. We start from the sequential algorithm which describes logical behaviour of the encryption processor. Our goal is to identify and isolate the basic functional components of the circuit into action systems of their own. The component action systems are joint together in a parallel composition, where they interact with each other using remote procedure calls. Synchronous communication between different componnets of the circuit is achieved through the procedures mechanism [6].

Overview of the paper In section 2, we briefly describe the action systems formalism. In section 3, we describe how action systems are composed into parallel systems. We also briefly mention the refinement calculus. In section 4, we describe the the target processor and give an initial specification for it as a sequential program. In section 5, this specification is stepwise turned into a parallel composition of action systems, where each action system represents one basic functional component of the circuit. In section 6, we introduce real parallelism into our system by creating 3-level pipeline. Finally in section 7, we conclude with some remarks on the proposed method and further steps in the derivation process.

2 Action systems

An *action system* (with procedures) is a statement of the form

$$\begin{array}{ll} \mathcal{A} :: & \textbf{var } v; \textbf{proc } w \bullet \\ & |[\ \textbf{var } x_1, \ldots, x_h := x_10, \ldots, x_h0; \textbf{proc } p_1 = P_1;\ \ldots; p_n = P_n; \\ & \quad \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m \textbf{ od} \\ &]|: z \end{array}$$

The identifiers $x_1, \ldots, x_h$ are the variables declared in $\mathcal{A}$ and initialized to $x_10, \ldots, x_h0$, $p_1, \ldots, p_n$ are the *procedure headers*, and P_i is the *procedure body* of p_i, $i = 1, \ldots, n$. Within the loop, $A_1, \ldots, A_m$ are the *actions* of $\mathcal{A}$. Finally,

z, v and w are pairwise distinct lists of identifiers. The list z is the *import list*, indicating which variables and procedures are referenced, but not declared in $\mathcal{A}$. The lists v and w are the *export lists*, indicating which variables and procedures declared in $\mathcal{A}$ are accessible from other action systems. Procedure bodies and actions can be arbitrary statements, and may contain procedure calls.

The *guard* of a program statement S is the condition gS, defined by

$$gS \;=\; \neg\mathrm{wp}(S, \mathit{false})$$

where wp is the *weakest precondition* predicate transformer of Dijkstra [8]. Both procedure bodies and actions will in general be *guarded commands*, i.e., statements of the form

$$C = g \rightarrow S,$$

where g is a boolean condition and S is a program statement. In this case, the guard of C is $g \wedge \neg\mathrm{wp}(S, \mathit{false})$.

Enabledness of an action Let $p = (b \rightarrow T)$ be a procedure and let $a \rightarrow S; p$ be an action that calls on p. Then the enabledness of this action is determined by the value of the action guard

$$g(a \rightarrow S; p) \;=\; a \wedge g(S; p)$$

If a procedure or action contains a call to a procedure that is not declared in the action system, then the behavior of the action system will depend on the way in which the procedures are declared in some other action system, which constitutes the environment of the action system as will be described below.

3 Composing and refining action systems

Consider two action systems,

$$\begin{array}{ll} \mathcal{A} :: & \textbf{var } v; \textbf{ proc } r \bullet \\ & |[\ \textbf{var } x := x0; \textbf{ proc } p_1 = P_1;\ \ldots;\ p_n = P_n; \\ & \quad \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m \textbf{ od} \\ &]|\!: z \end{array}$$

and

$$\begin{array}{ll} \mathcal{B} :: & \textbf{var } w; \textbf{ proc } s \bullet \\ & |[\ \textbf{var } y := y0; \textbf{ proc } q_1 = Q_1;\ \ldots;\ q_l = Q_l; \\ & \quad \textbf{do } B_1 \ [\!]\ \ldots\ [\!]\ B_k \textbf{ od} \\ &]|\!: u \end{array}$$

where $x \cap y = \emptyset$, $v \cap w = \emptyset$, and $r \cap s = \emptyset$. Furthermore, the local procedures declared in the two action systems are required to be distinct.

We define the *parallel composition* $\mathcal{A} \parallel \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$ to be the action system $\mathcal{C}$

$$
\begin{array}{ll}
\mathcal{C} :: & \textbf{var } b; \textbf{ proc } c \bullet \\
& |[\ \ \textbf{var } x, y := x0, y0; \\
& \quad \textbf{proc } p_1 = P_1;\ \ldots; p_n = P_n;\ q_1 = Q_1;\ \ldots; q_l = Q_l; \\
& \quad \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m \ [\!]\ B_1 \ [\!]\ \ldots\ [\!]\ B_k \textbf{ od} \\
&]|: a
\end{array}
$$

where $a = z \cup u - (v \cup r \cup w \cup s), b = v \cup w, c = r \cup s$.

Thus, parallel composition will combine the state spaces of the two constituent action systems, merging the global variables and global procedures and keeping the local variables distinct. The imported identifiers denote those global variables and/or procedures that are not declared in either $\mathcal{A}$ or $\mathcal{B}$. The exported identifiers are the variables and/or procedures declared global in $\mathcal{A}$ or $\mathcal{B}$. The procedure declarations and the actions in the parallel composition consists of the procedure declarations and and actions in the original systems.

Enabledness We permit procedure bodies to have guards that are not identically true. Hence, it is possible that an action which is enabled, calls a procedure in another action system, which then turns out not to be enabled in the state in which it is called. This situation is then the same as if the calling action had not been enabled at all, and had therefore never initiated the call. In other words, the enabledness of an action is determined by the enabledness of the whole statement that is invoked when the action is executed, including enabledness of all procedures that might be invoked.

Decomposing action systems Given an action system

$$
\begin{array}{ll}
\mathcal{C} :: & \textbf{var } u; \textbf{ proc } s \bullet \\
& |[\ \ \textbf{var } v := v0; \\
& \quad \textbf{do } C_1 \ [\!]\ \ldots\ [\!]\ C_n \textbf{ od} \\
&]|: z
\end{array}
$$

we can *decompose* it into smaller action systems by parallel composition. This means that we split the variables, actions and procedures of $\mathcal{C}$ into disjoint sets so that

$$
\begin{array}{ll}
\mathcal{C} = & \textbf{var } u; \textbf{ proc } s \bullet \\
& |[\ \ \textbf{var } w := w0; \textbf{ proc } r = R; \\
& \quad \mathcal{A} \parallel \mathcal{B} \\
&]|: z
\end{array}
$$

where

$$
\begin{array}{ll}
\mathcal{A} :: & \textbf{var } a_2; \textbf{ proc } a_3 \bullet \\
& |[\ \ \textbf{var } x := x0; \textbf{ proc } p = P; \\
& \quad \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m \textbf{ od} \\
&]|: a_1
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{B} :: & \textbf{var } b_2; \textbf{ proc } b_3 \bullet \\
& |[\ \ \textbf{var } y := y0; \textbf{ proc } q = Q; \\
& \quad \textbf{do } B_1 \ [\!]\ \ldots\ [\!]\ B_k \textbf{ od} \\
&]|: b_1
\end{array}
$$

The reactive components $\mathcal{A}$ and $\mathcal{B}$ interact with each other via the global variables and procedures included in the lists a_2, a_3, b_2, b_3.

Hiding and revealing Let

$$\textbf{var } v_1, v_2; \textbf{ proc } v_3, v_4 \bullet \mathcal{A} : z$$

be an action system of the form above, where z denotes the import list and v_1, v_2, v_3, v_4 denote the export lists. We can *hide* some of the exported global variables (v_2) and procedure names (v_4) by removing them from the export list,

$$\mathcal{A}' \;=\; \textbf{var } v_1; \textbf{ proc } v_3 \bullet \mathcal{A} : z.$$

Hiding the variables v_2 and procedure names v_4 makes them inaccessible from other actions outside $\mathcal{A}'$ in a parallel composition. Hiding thus has an effect only on the variables and procedures in the export list. The opposite operation, *revealing*, is also useful.

In connection with the parallel composition below we will follow the following convention. Let **var** a_1; **proc** $a_2 \bullet \mathcal{A} : a_3$ and **var** b_1; **proc** $b_2 \bullet \mathcal{B} : b_3$ be two action systems. Then their parallel composition is the action system

$$\textbf{var } a_1 \cup b_1; \textbf{ proc } a_2 \cup b_2 \bullet \mathcal{A} \parallel \mathcal{B} : c$$

where $c = a_3 \cup b_3 - (a_1 \cup a_2 \cup b_1 \cup b_2)$ according to the definition above. Hence, the parallel composition exports all the variables and procedures exported by either $\mathcal{A}$ or $\mathcal{B}$. Sometimes there is no need to export all these identifiers, i.e., when they are exclusively accessed by the two component action systems $\mathcal{A}$ and $\mathcal{B}$. This effect is achieved with the following construct that turns out to be extremely useful later:

$$\textbf{var } v; \textbf{ proc } p \bullet |[\; \mathcal{A} \parallel \mathcal{B} \;]| : c$$

Here the identifiers v and p are as follows: $v \subseteq a_1 \cup b_1$ and $p \subseteq a_2 \cup b_2$.

Refining action systems Most of the steps we will carry out within the microprocessor derivation are purely syntactic decomposition steps. There are, however, a couple of steps where a higher level action system is refined into an other action system. These steps are formally carried out within the refinement calculus, where we consider action systems as ordinary statements, i.e., as initialized iteration statements.

The refinement calculus is based on the following definition. Let S and S' be two statements. Then S is correctly *refined* by S', denoted $S \leq S'$, if for any postcondition Q

$$\mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S', Q).$$

4 FEAL algorithm and initial specification of the processor

Algorithm As a case study we look at the design of an asynchronous encryption/decryption (FEAL) processor that implements a special case of the so called FEAL-N algorithm [10]. The algorithm takes plaintext (64-bit word) as an input and produces encrypted text of the same length. While encrypting it uses 16-bit keys that are computed from the original one given by the user. We here assume that there are available (when needed) $N+8$ keys in the so called key memory.

Encryption of input word T is given by the following set of equations:

$$\begin{aligned} L_0.R_{-1} &= K_{N..N+3} \oplus T && (1)\\ R_0 &= R_{-1} \oplus L_0 && (2) \end{aligned}$$

for $i \in 1..N$:

$$\begin{aligned} R_i &= L_{i-1} \oplus F(R_{i-1}, K_{i-1}) && (3)\\ L_i &= R_{i-1} && (4) \end{aligned}$$

$$\begin{aligned} L_{N+1} &= R_N \oplus L_N && (5)\\ C &= K_{N+4..N+7} \oplus R_N.L_{N+1} && (6) \end{aligned}$$

Here C stands for the output (encrypted text). A concatenation of two words W_1 and W_2 is writen as $W_1.W_2$, while exclusive-or (for W_1, W_2 of the equal length) is denoted as $W_1 \oplus W_2$. The word under encryption is then represented as a concatenation $L.R$ of the left and right part (both 32-bit long). Furthermore, let $K_{m..n}$ be short for $K_m.K_{m+1}...K_n$ with K_j being j-th key. We assume that function F is available and do not consider its implementation. Decryption algorithm has the same structure only the keys are used in different order.

The parameter N is the number of rounds in the main encryption loop (Equations 3, 4). We consider the case with $N = 8$. Some extra processing of the word is done before the loop (Equations 1, 2) and after (Equations 5, 6) it. So the algorithm consists of three logical parts.

Structure of the microprocessor We make the design decision to implement each logical part of the algorithm by a separate functional component in the target microprocessor. When processing one word the components are activated one after the other without any possibility for simultaneous activity. However, the processor is supposed to work with a sequence of words. In this case concurrent activity of all its parts is possible. The solution is to have a 3-level pipeline system. To keep input of the second and third components unchanged until the processing of data has been completed and result passed to the next component we introduce two registers (see Figure 1). All components of the circuit communicates through the indicated channels.

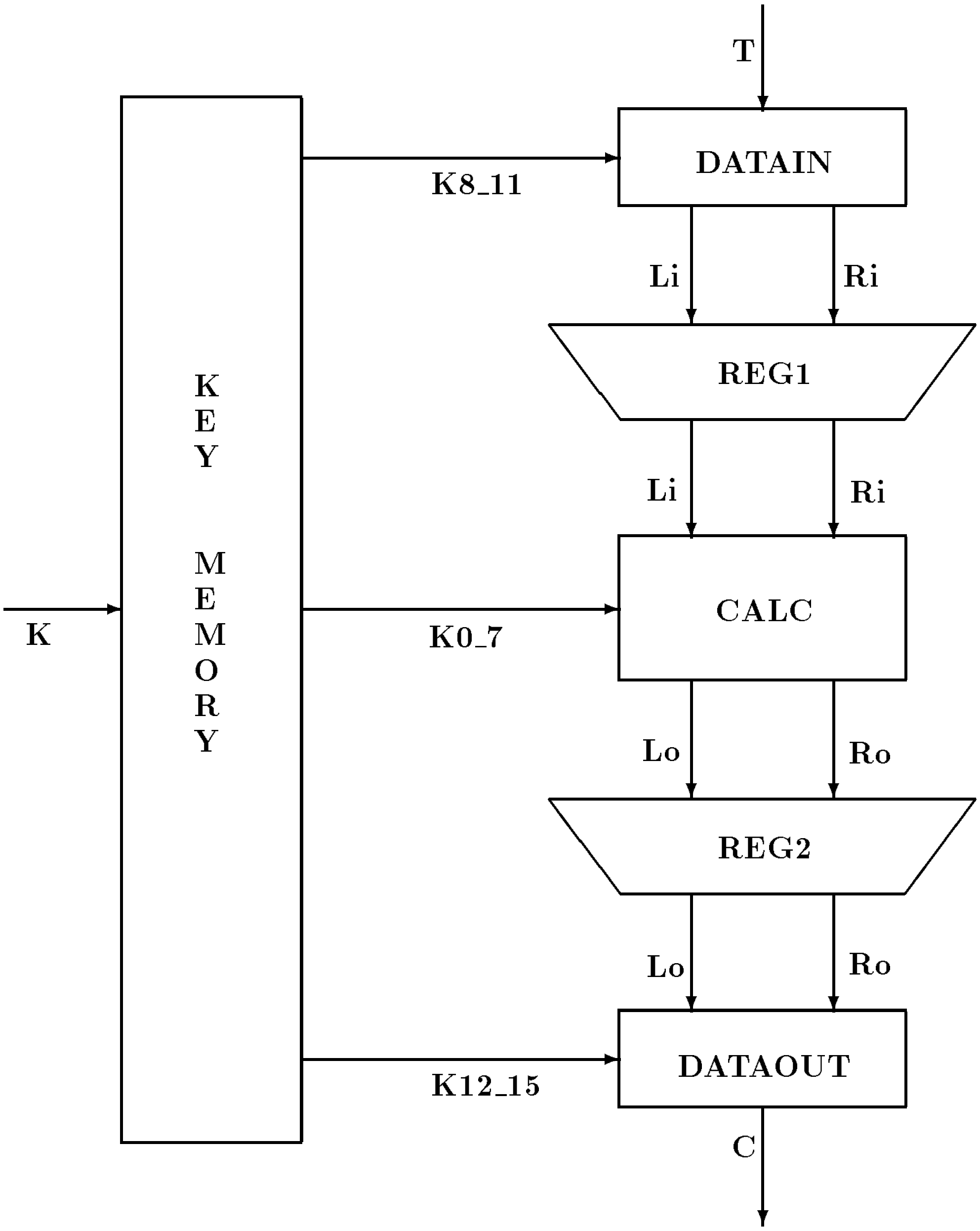

Figure 1: Structure of the microprocessor

We start from the sequential action system that implements above algorithm in the procedure $FEAL$:

$$
\begin{array}{l}
\mathcal{F}0{::}\ \ \textbf{proc}\ FEAL\ \bullet \\
\quad [\![\ \ \textbf{var}\ L_i,R_i\ ,\ R_o,L_o\ ,\ T\ ; \\
\qquad \textbf{proc}\ FEAL(\mathrm{t})= \\
\qquad\qquad (T:=t; \\
\qquad\qquad\ \ [\![\textbf{var}\ R_{\underline{i}},L_{\underline{o}},L[j],R[j]\ \textbf{for}\ j=0,\ldots,8\ ; \\
\qquad\qquad\quad L_i.R_{\underline{i}}:=\ K8..11\oplus T\ ; \\
\qquad\qquad\quad R_i:=\ L_i\oplus R_{\underline{i}}\ ; \\
\qquad\qquad\quad L[0],R[0]:=\ L_i,R_i\ ; \\
\qquad\qquad\quad [\ L[j],R[j]:=\ R[j\text{-}1]\ ,\ L[j-1]\oplus F(R[j\text{-}1],K0..7[16j\text{-}15..16j])\ ] \\
\qquad\qquad\qquad\qquad \textbf{for}\ j=0,\ldots,8\ ; \\
\qquad\qquad\quad R_o,L_o:=R[8]\ ,\ L[8]\ ; \\
\qquad\qquad\quad L_{\underline{o}}:=\ R_o\oplus L_o\ ; \\
\qquad\qquad\quad C:=\ K12..15\oplus R_o.L_{\underline{o}} \\
\qquad\qquad\ \ ]\!]\ ) \\
\quad\ \ ]\!]{:}C\ ,\ K
\end{array}
$$

In our target program each component of the processor is represented as an action system of its own. Parallel composition of them models the entire system where all elements can be active at the same time. Communication on the channels (synchronization of the action systems) is expressed as a remote procedure calls. We do not consider a derivation of any particular component of the processor here, instead we look more carefully at the task of isolating the basic functional parts and modelling communication through the channels on the low (signal) level.

5 Decomposition into parallel action systems

As described above, the algorithm consists of three parts. We therefore start by making three procedures to represent them:

$$
\begin{array}{l}
\mathcal{F}1{::}\ \ \textbf{proc}\ FEAL\ \bullet \\
\quad [\![\ \ \textbf{var}\ L_i,R_i\ ,\ R_o,L_o\ ,\ T\ ; \\
\qquad \textbf{proc}\ INXOR(\textbf{var}\ l_i,r_i)=([\![\ \textbf{var}\ R_{\underline{i}}\ ;\ \ l_i.R_{\underline{i}}:=\ K8..11\oplus T\ ;\ \ r_i:=\ l_i\oplus R_{\underline{i}}\ ]\!])\ ; \\
\qquad \textbf{proc}\ CIPHER(l_i,r_i,\textbf{var}\ r_o,l_o)= \\
\qquad\qquad ([\![\textbf{var}\ L[j],R[j]\ \textbf{for}\ j=0,\ldots,8\ ; \\
\qquad\qquad\quad L[0],R[0]:=\ l_i,r_i\ ; \\
\qquad\qquad\quad [\ L[j],R[j]:=\ R[j\text{-}1]\ ,\ L[j-1]\oplus F(R[j\text{-}1],K0..7[16j\text{-}15..16j])\ ] \\
\qquad\qquad\qquad\qquad \textbf{for}\ j=0,\ldots,8\ ; \\
\qquad\qquad\quad r_o\ ,\ l_o:=R[8]\ ,\ L[8] \\
\qquad\qquad\ \ ]\!]\ )\ ; \\
\qquad \textbf{proc}\ OUTXOR(r_o\ ,\ l_o)=([\![\ \textbf{var}\ L_{\underline{o}}\ ;\ \ L_{\underline{o}}:=\ r_o\oplus l_o\ ;\ \ C:=\ K12..15\oplus r_o.L_{\underline{o}}\ ]\!])\ ; \\
\qquad \textbf{proc}\ FEAL(t)= \\
\qquad\qquad (T:=\ t\ ;\ \ INXOR(L_i,R_i);\ CIPHER(L_i,R_i,R_o,L_o);\ OUTXOR(R_o,L_o)) \\
\quad]\!]{:}C\ ,\ K
\end{array}
$$

We have that $\mathcal{F}0 \leq \mathcal{F}1$.

Now the encryption itself is done in procedure *CIPHER*, while *INXOR* performs preparatory processing of the word and *OUTXOR* ends whole process.

Parallelize F There is no parallel activity possible in the above action system. Hence, we proceed by creating a parallel composition of action systems which contains actions that can be executed simultaneously.

First we split the procedure *FEAL* into three parts by creating new procedures *DIN*, *CALC* and *DOUT*. *DIN* isolates procedure *INXOR* from the rest while *CALC* and *DOUT* separate procedures *CIPHER* and *OUTXOR*. More formally this can be represented as the following sequence of refinements:

$(T := t\ ;\ INXOR(L_i\ ,R_i);\ CIPHER(L_i,R_i,R_o,L_o);\ OUTXOR(R_o,L_o))$
$\leq$
proc $DIN(l_i,r_i)=(INXOR(l_i,r_i)\ ;\ \ CIPHER(L_i,R_i,R_o,L_o);\ OUTXOR(R_o,L_o))$;
$(T := t\ ;\ DIN(L_i,R_i))$
$\leq$
proc $DIN(l_i,r_i)=(INXOR\ (l_i,r_i)\ ;\ \ CALC(l_i,r_i))$;
proc $CALC(l_i,r_i)=(CIPHER(l_i,r_i,R_o,L_o)\ ;\ \ OUTXOR(R_o,L_o))$;
$(T := t\ ;\ DIN(L_i,R_i))$
$\leq$
proc $DIN(l_i,r_i)=(INXOR(l_i,r_i)\ ;\ \ CALC(l_i,r_i))$;
proc $CALC(l_i,r_i)=(CIPHER(l_i,r_i,R_o,L_o)\ ;\ \ DOUT(R_o,L_o))$;
proc $DOUT(r_o\ ,\ l_o)=(OUTXOR(r_o\ ,\ l_o))$;
$(T := t\ ;\ DIN(L_i,R_i))$

Now a new action system is as below:

$\mathcal{F}2$:: **proc** *FEAL* $\bullet$
$[\![$ **var** $L_i,R_i\ ,\ R_o,L_o\ ,\ T$;
proc $INXOR($**var** $l_i,r_i)=([\![$ **var** $R_{\underline{i}}$; $l_i.R_{\underline{i}} := K8..11 \oplus T$; $r_i := l_i \oplus R_{\underline{i}}\ ]\!])$;
proc $CIPHER(l_i,r_i,$**var** $r_o,l_o)=$
$([\![$**var** $L[j],R[j]$ **for** $j=0,\ldots,8$;
$L[0],R[0] := l_i,r_i$;
$[\ L[j],R[j] := R[j\text{-}1]\ ,\ L[j-1] \oplus F(R[j\text{-}1],K0..7[16j\text{-}15..16j])\ ]$
for $j=0,\ldots,8$;
$r_o\ ,\ l_o := R[8]\ ,\ L[8]$
$]\!]\ )$;
proc $OUTXOR(r_o\ ,\ l_o)=([\![$ **var** $L_{\underline{o}}$; $L_{\underline{o}} := r_o \oplus l_o$; $C := K12..15 \oplus r_o.L_{\underline{o}}\ ]\!])$;
proc $DIN(l_i,r_i)=(INXOR(l_i,r_i)\ ;\ CALC(l_i,r_i))$;
proc $CALC(l_i,r_i)=(CIPHER(l_i,r_i,R_o,L_o)\ ;\ DOUT(R_o,L_o))$;
proc $DOUT(r_o\ ,\ l_o)=(OUTXOR(r_o\ ,\ l_o))$;
proc $FEAL(t)=(T := t\ ;\ DIN(L_i,R_i))$
$]\!]:C\ ,\ K$

We have also proved the refinement relation $\mathcal{F}1 \leq \mathcal{F}2$.

Next we can relax the atomicity constraints as follows. First we add and initialize two new variables *in* and *ca*:

$\mathcal{F}_3$:: **proc** *FEAL* $\bullet$

$[\![$ **var** L_i, R_i , R_o, L_o , T ; ic , $co \in boolean$;

proc $INXOR($**var** $l_i, r_i)=([\![$ **var** $R_{\underline{i}}$; $l_i.R_{\underline{i}} := K8..11 \oplus T$; $r_i := l_i \oplus R_{\underline{i}}$ $]\!])$;

proc $CIPHER(l_i, r_i,$**var** $r_o, l_o)=$

$([\![$**var** $L[j], R[j]$ **for** $j=0,\ldots,8$;

$L[0], R[0] := l_i, r_i$;

$[$ $L[j], R[j] := R[j\text{-}1]$, $L[j-1] \oplus F(R[j\text{-}1], K0..7[16j\text{-}15..16j])$ $]$

for $j=0,\ldots,8$;

r_o , $l_o := R[8]$, $L[8]$

$]\!]$) ;

proc $OUTXOR(r_o$, $l_o)=([\![$ **var** $L_{\underline{o}}$; $L_{\underline{o}} := r_o \oplus l_o$; $C := K12..15 \oplus r_o.L_{\underline{o}}$ $]\!])$;

proc $DIN(l_i, r_i)=($ $\neg in \rightarrow INXOR(l_i, r_i)$; $in := true$) ;

proc $CALC(l_i, r_i)=($ $\neg ca \rightarrow CIPHER(l_i, r_i, R_o, L_o)$; $ca := true$) ;

proc $DOUT(r_o$, $l_o)=(OUTXOR(r_o$, $l_o))$;

proc $FEAL(t)=(T := t$; $DIN(L_i, R_i))$;

in , $ca := false$, $false$;

do

$in \rightarrow CALC(l_i, r_i)$; $in := false$

$[\!]$ $ca \rightarrow DOUT(R_o, L_o)$; $ca := false$

od

$]\!]$:K , C

We have that $\mathcal{F}_2 \leq \mathcal{F}_3$.

Both *DIN* and *CALC* can be viewed as consisting of two parts. The first one execute procedures *INXOR* and *CIPHER* correspondingly. Then the control is returned to the caller. Later on the procedures *CALC* and *DOUT* are called from separate actions that can be enabled simultaneously. The effect of these refinements is as if we would have introduced an explicit **return** statement into our language:

$\mathcal{F}_4$:: **proc** *FEAL* $\bullet$

$[\![$ **var** L_i, R_i , R_o, L_o , T ;

proc $INXOR($**var** $l_i, r_i)=([\![$ **var** $R_{\underline{i}}$; $l_i.R_{\underline{i}} := K8..11 \oplus T$; $r_i := l_i \oplus R_{\underline{i}}$ $]\!])$;

proc $CIPHER(l_i, r_i,$**var** $r_o, l_o)=$

$([\![$**var** $L[j], R[j]$ **for** $j=0,\ldots,8$;

$L[0], R[0] := l_i, r_i$;

$[$ $L[j], R[j] := R[j\text{-}1]$, $L[j-1] \oplus F(R[j\text{-}1], K0..7[16j\text{-}15..16j])$ $]$

for $j=0,\ldots,8$;

r_o , $l_o := R[8]$, $L[8]$

$]\!]$) ;

proc $OUTXOR(r_o$, $l_o)=([\![$ **var** $L_{\underline{o}}$; $L_{\underline{o}} := r_o \oplus l_o$; $C := K12..15 \oplus r_o.L_{\underline{o}}$ $]\!])$;

proc $DIN(l_i, r_i)=(INXOR(l_i, r_i)$; **return** ; $CALC(l_i, r_i))$;

proc $CALC(l_i, r_i)=(CIPHER(l_i, r_i, R_o, L_o)$; **return** ; $DOUT(R_o, L_o))$;

proc $DOUT(r_o$, $l_o)=(OUTXOR(r_o, l_o))$;

proc $FEAL(t)=(T := t$; $DIN(L_i, R_i))$

$]\!]$:C , K

Hence we have that $\mathcal{F}_3 = \mathcal{F}_4$.

The effect of carrying out $\mathcal{F}3$ is the same as that of $\mathcal{F}4$. The **return** statement is merely syntactic sugaring.

Separation of DATAIN, CALC and DATAOUT Let us now decompose the action system $\mathcal{F}4$ into a parallel composition of three action systems $\mathcal{D}atain$, $\mathcal{C}alc$ and $\mathcal{D}ataout$:

$$\mathcal{F}4 \leq \mathcal{D}atain \parallel \mathcal{C}alc \parallel \mathcal{D}ataout$$

where new action systems are defined as below:

$\mathcal{D}atain$:: **proc** *FEAL* •
  |[**var** L_i,R_i , T ;
    proc *INXOR*=(|[**var** $R_{\underline{i}}$; $L_i.R_{\underline{i}}$:= $K8..11 \oplus T$; R_i:= $L_i \oplus R_{\underline{i}}$]|) ;
    proc $DIN(l_i,r_i)$=(L_i,R_i:= l_i,r_i ; *INXOR* ; **return** ; $CALC(L_i,R_i)$);
    proc $FEAL(t)$=(T:= t ; $DIN(L_i,R_i)$)
  ]|:K , *CALC*

$\mathcal{C}alc$:: **proc** *CALC* •
  |[**var** L_i,R_i,R_o,L_o ;
    proc *CIPHER*=
        (|[**var** $L[j]$,$R[j]$ **for** j=0,...,8 ;
          $L[0]$,$R[0]$:= L_i,R_i ;
          [$L[j]$,$R[j]$:= $R[j\text{-}1]$, $L[j-1] \oplus F(R[j\text{-}1], K0..7[16j\text{-}15..16j])$]
              for j=0,...,8 ;
          R_o,L_o :=$R[8]$, $L[8]$
        ]|) ;
    proc $CALC(l_i,r_i)$=(L_i,R_i:= l_i,r_i ; *CIPHER* ; **return** ; $DOUT(R_o , L_o)$)
  ]|:K , *DOUT*

$\mathcal{D}ataout$:: **proc** *DOUT* •
  |[**var** R_o , L_o ;
    proc *OUTXOR*=(|[**var** $L_{\underline{o}}$; $L_{\underline{o}}$:= $R_o \oplus L_o$; C:= $K12..15 \oplus R_o.L_{\underline{o}}$]|) ;
    proc $DOUT(r_o , l_o)$=(R_o , L_o:= r_o , l_o ; *OUTXOR*)
  ]|:C , K

Now the systems communicate through the global procedures *CALC* and *DOUT*. We have also removed the parameters (l_i,r_i) and (r_o,l_o) from procedures *INXOR*, *CIPHER* and *OUTXOR* and added some extra assignments into procedures *DIN*, *CALC* and *DOUT* to play the role of the now unnecessary parameters.

6 Registers

From three action systems above only $\mathcal{D}atain$ and $\mathcal{D}ataout$ can be executed in parallel. To make all components executable simultaneously, we add two registers between the action systems. New procedures *REG1* and *REG2* are created to keep intermediate results of the computation unchanged while they are used by the next component.

First we refine $\mathcal{D}atain$ to $\mathcal{D}atain'$:

$(L_i,R_i\!:= l_i,r_i\ ;\ \mathit{INXOR}\ ;\ \mathbf{return}\ ;\ \mathit{CALC}(L_i,R_i))$
$\leq$
$\mathbf{proc}\ \mathit{REG1}(l_i,r_i)=(\|[\ \mathbf{var}\ L_{\underline{i}},R_{\underline{i}}\ ;\ L_{\underline{i}},R_{\underline{i}}\!:= l_i,r_i\ ;\ \mathbf{return}\ ;\ \mathit{CALC}(L_{\underline{i}},R_{\underline{i}})\ ]\|)\ ;$
$(L_i,R_i\!:= l_i,r_i\ ;\ \mathit{INXOR}\ ;\ \mathbf{return}\ ;\ \mathit{REG1}(L_i,R_i))$

$\mathcal{D}atain'$:: $\mathbf{proc}\ \mathit{FEAL}\ \bullet$
$\|[\ \mathbf{var}\ L_i,R_i\ ,\ T\ ;$
$\mathbf{proc}\ \mathit{INXOR}{=}(\|[\ \mathbf{var}\ \ R_{\underline{i}};\ L_i.R_{\underline{i}}\!:=\ K8..11{\oplus}T;\ R_i\!:=\ L_i{\oplus}R_{\underline{i}}\ ]\|)\ ;$
$\mathbf{proc}\ \mathit{FEAL}(t)=(T\!:=\ t\ ;\ \mathit{DIN}(L_i,R_i))\ ;$
$\mathbf{proc}\ \mathit{DIN}(l_i,r_i)=(L_i,R_i\!:=\ l_i,r_i\ ;\ \mathit{INXOR}\ ;\ \mathbf{return}\ ;\ \mathit{REG1}(L_i,R_i))\ ;$
$\mathbf{proc}\ \mathit{REG1}(l_i,r_i)=$
$(\|[\ \ \mathbf{var}\ L_{\underline{i}},R_{\underline{i}}\ ;\ L_{\underline{i}},R_{\underline{i}}\!:=\ l_i,r_i\ ;\ \mathbf{return}\ ;\ \mathit{CALC}(L_{\underline{i}},R_{\underline{i}})\ \ ]\|);$
$\mathbf{proc}\ \mathit{FEAL}(t)=(T\!:=\ t\ ;\ \mathit{DIN}(L_i,R_i))$
$]\|{:}K\ ,\ \mathit{CALC}$

Then $\mathcal{C}alc$ is refined by $\mathcal{C}alc'$:

$(L_i,R_i\!:= l_i,r_i\ ;\ \mathit{CIPHER}\ ;\ \mathbf{return}\ ;\ \mathit{DOUT}(R_o\ ,\ L_o))$
$\leq$
$\mathbf{proc}\ \mathit{REG2}(r_o,l_o)=(\|[\ \mathbf{var}\ R_{\underline{o}},L_{\underline{o}}\ ;\ R_{\underline{o}},L_{\underline{o}}\!:=\ R_o,L_o\ ;\ \mathbf{return}\ ;\ \mathit{DOUT}(R_{\underline{o}},L_{\underline{o}})\ ]\|)\ ;$
$(L_i,R_i\!:= l_i,r_i\ ;\ \mathit{CIPHER}\ ;\ \mathbf{return}\ ;\ \mathit{REG2}(R_o\ ,\ L_o))$

$\mathcal{C}alc'$:: $\mathbf{proc}\ \mathit{CALC}\ \bullet$
$\|[\ \mathbf{var}\ L_i,R_i\ ,\ R_o,L_o\ ;$
$\mathbf{proc}\ \mathit{CIPHER}=$
$(\|[\mathbf{var}\ L[j],R[j]\ \mathbf{for}\ j{=}0,\ldots,8\ ;$
$L[0],R[0]\!:=\ L_i,R_i\ ;$
$[\ L[j],R[j]\!:=\ R[j\text{-}1]\ ,\ L[j-1]{\oplus}F(R[j\text{-}1],K0..7[16j\text{-}15..16j])\ ]$
$\mathbf{for}\ j{=}0,\ldots,8\ ;$
$R_o,L_o\ :=R[8]\ ,\ L[8]$
$]\|\ )\ ;$
$\mathbf{proc}\ \mathit{CALC}(l_i,r_i)=(L_i,R_i\!:=\ l_i,r_i\ ;\ \mathit{CIPHER}\ ;\ \mathbf{return}\ ;\ \mathit{REG2}(R_o\ ,\ L_o))\ ;$
$\mathbf{proc}\ \mathit{REG2}(r_o,l_o)=$
$(\|[\ \ \mathbf{var}\ R_{\underline{o}},L_{\underline{o}}\ ;\ R_{\underline{o}},L_{\underline{o}}\!:=\ R_o,L_o\ ;\ \mathbf{return}\ ;\ \mathit{DOUT}(R_{\underline{o}},L_{\underline{o}})\ \ ]\|)$
$]\|{:}K\ ,\ \mathit{DOUT}$

Finally, we split $\mathcal{D}atain'$ into two components $\mathcal{D}atain''$ and $\mathcal{R}eg1$ as follows:

$\mathcal{D}atain''$:: $\mathbf{proc}\ \mathit{FEAL}\ \bullet$
$\|[\ \mathbf{var}\ L_i,R_i\ ,\ T\ ;$
$\mathbf{proc}\ \mathit{INXOR}{=}(\|[\ \mathbf{var}\ \ R_{\underline{i}};\ L_i.R_{\underline{i}}\!:=\ K8..11{\oplus}T;\ R_i\!:=\ L_i{\oplus}R_{\underline{i}}\ ]\|)\ ;$
$\mathbf{proc}\ \mathit{DIN}(l_i,r_i)=(L_i,R_i\!:=\ l_i,r_i\ ;\ \mathit{INXOR}\ ;\ \mathbf{return}\ ;\ \mathit{REG1}(L_i,R_i));$
$\mathbf{proc}\ \mathit{FEAL}(t)=(T\!:=\ t\ ;\ \mathit{DIN}(L_i,R_i))$
$]\|{:}K\ ,\ \mathit{REG1}$

$\mathcal{R}eg1$:: **proc** *REG1* $\bullet$
$\quad$ $[\![$ **proc** $REG1(l_i,r_i)=$
$\qquad\qquad$ $([\![$ **var** $L_{\underline{i}},R_{\underline{i}}$; $L_{\underline{i}},R_{\underline{i}}:= l_i,r_i$; **return** ; $CALC(L_{\underline{i}},R_{\underline{i}})$ $]\!])$
$\quad$ $]\!]:K$, *CALC*

Similarly, $\mathcal{C}alc'$ is divided into $\mathcal{C}alc''$ and $\mathcal{R}eg2$:

$\mathcal{C}alc''$:: **proc** *CALC* $\bullet$
$\quad$ $[\![$ **var** L_i,R_i , R_o,L_o ;
$\qquad$ **proc** *CIPHER*=
$\qquad\qquad$ $([\![$**var** $L[j],R[j]$ **for** $j=0,\ldots,8$;
$\qquad\qquad\quad$ $L[0],R[0]:= L_i,R_i$;
$\qquad\qquad\quad$ $[\ L[j],R[j]:= R[j\text{-}1]\ ,\ L[j-1]\oplus F(R[j\text{-}1],K0..7[16j\text{-}15..16j])\ ]$
$\qquad\qquad\qquad\qquad$ **for** $j=0,\ldots,8$;
$\qquad\qquad\quad$ $R_o,L_o := R[8]\ ,\ L[8]$
$\qquad\qquad$ $]\!]$) ;
$\qquad$ **proc** $CALC(l_i,r_i)=(L_i,R_i:= l_i,r_i$; *CIPHER* ; **return** ; $REG2(R_o\ ,\ L_o))$
$\quad$ $]\!]:K$, *REG2*

$\mathcal{R}eg2$:: **proc** *REG2* $\bullet$
$\quad$ $[\![$ **proc** $REG2(r_o,l_o)=$
$\qquad\qquad$ $([\![$ **var** $R_{\underline{o}},L_{\underline{o}}$; $R_{\underline{o}},L_{\underline{o}}:= R_o,L_o$; **return** ; $DOUT(R_{\underline{o}},L_{\underline{o}})$ $]\!])$
$\quad$ $]\!]:K$, *DOUT*

As a result, we have derived the action system that models the processor in Figure 1:

$\mathcal{F}5$:: **proc** *FEAL* $\bullet$ $[\![\mathcal{D}atain'' \parallel \mathcal{R}eg1 \parallel \mathcal{C}alc'' \parallel \mathcal{R}eg2 \parallel \mathcal{D}ataout]\!]:\ C\ ,\ K$

We have also established the refinement relation between initial and final programs:

$$\mathcal{F}0 \leq \mathcal{F}5$$

7 Concluding remarks

The final parallel composition of action systems is a refinement of the initial sequential program. We isolated so far the basic functional components of the processor and introduced two registers thus creating 3-level pipeline system. All manipulations with the program were completely formal and based on the refinement calculus. To achieve this result we mostly used the method of parallel decomposition of an action system. Communication through the channels was modelled by remote procedure calls. Some steps involved a refinement of the atomicity of the system so they were true refinements and required proofs.

Next step in the derivation would be a refinement of the communication between action systems (components of the processor). We intend to implement it by so called handshaking protocol. Then an isolation of control and data parts of the circuit could follow.

References

[1] R. J. R. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, Helsinki, Finland, 1978. Report A–1978–4.

[2] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.–P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989*, volume 430 of *Lecture Notes in Computer Science*. Springer–Verlag, 1990.

[3] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proc. of the 2nd ACM SIGACT–SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[4] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming* 13, pages 133–180, 1989.

[5] R. J. R. Back and K. Sere. Action systems with synchronous communication. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi. Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference*, pages 107–126, 1994.

[6] R. J. R. Back and K. Sere. Specification of a Microprocessor. Reports on computer science and mathematics 148, Åbo Akademi, 1994.

[7] R. J. R. Back and J. von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing* 2, pages 247–272, 1990.

[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[9] A. J. Martin. Synthesis of Asynchronous VLSI Circuits. *CalTech*, Technical Report, 1993.

[10] S. Miyaguchi and A. Shimizu. Fast Data Encipherment Algorithm. *EUROCRYPT '87*, pages 267–278, 1987.

[11] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[12] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[13] K. Sere *Stepwise Refinement of Parallel Algorithms.* PhD thesis, Department of Computer Science, Åbo Akademi University, Turku, Finland, 1990.

[14] J. Staunstrup and M. R. Greenstreet. *Synchronized Transitions.* IFIP WG 10.5, Summer school on Formal Methods for VLSI Design, Lecture Notes 1990.