ELSEVIER

Science of Computer Programming 26 (1996) 79–97

Science of Computer Programming

Specifying the Caltech asynchronous microprocessor

R.J.R. Back [a,1], A.J. Martin [b], K. Sere [c,1,*]

[a] *Åbo Akademi University, Department of Computer Science, FIN-20520 Turku, Finland*
[b] *California Institute of Technology, Department of Computer Science, Pasadena CA 91125, USA*
[c] *University of Kuopio, Department of Computer Science and Applied Mathematics, FIN-70211 Kuopio, Finland*

Abstract

The action systems framework for modelling parallel programs is used to formally *specify* a microprocessor. First the microprocessor is specified as a sequential program. The sequential specification is then *decomposed* and *refined* into a concurrent program using correctness-preserving program transformations. Previously this microprocessor has been specified at Caltech, where an asynchronous circuit for the microprocessor was derived from the specification. We propose a specification strategy that is based on the idea of *spatial decomposition* of the program variable space.

1. Introduction

An *action system* is a parallel or distributed program where parallel activity is described in terms of events, so-called actions. The actions are *atomic*: if an action is chosen for execution, it is executed to completion without any interference from the other actions in the system. Several actions can be executed in parallel, as long as the actions do not share any variables. Atomicity guarantees that a parallel execution of an action system gives the same results as a sequential and non-deterministic execution.

A recent extension of the action system framework, adding procedure declarations to action systems [6], gives us a very general mechanism for synchronized communication between action systems. When an action in one action system calls a procedure in another action system, the effect is that of a remote procedure call. The calling action and the procedure body involved in the call are each executed as a single atomic entity.

The use of action systems permits the design of the logical behaviour of a system to be separated from the issue of how the system is to be implemented. The decision whether the action system is to be executed in a sequential or parallel fashion can

* Corresponding author.

[1] Partially supported by the Academy of Finland.

SSDI 0167-6423(95)00023-2

be postponed to a later stage, when the logical behaviour of the action system has been designed. The construction of the program is thus done within a single unifying framework.

The action systems formalism was proposed by Back and Kurki-Suonio [3]. Later similar event-based formalisms have been put forward by several other researchers, see for example the work of Chandy and Misra [8], who describe their UNITY framework and Francez [10], who develops his IP-language.

The *refinement calculus* is a formalization of the stepwise refinement method of program construction. It was originally proposed by Back [1] and has been later studied and extended by several researchers, see [13, 14] among others.

Originally, the refinement calculus was designed as a framework for systematic derivation of sequential programs only. Back and Sere [5, 15] extended the refinement calculus to 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. In both cases parallel and concurrent activity is modelled within a purely sequential framework. In [6] Back and Sere show how action systems with remote procedure calls can be derived within the refinement calculus for reactive systems. We will here show how this extension of the refinement calculus/action system framework is applied to a non-trivial case study, the formal derivation of an asynchronous microprocessor.

The initial specification of the microprocessor will be given as a sequential program that has the syntactic form of an action system. Our goal is to isolate the different functional components of the microprocessor, like instruction memory, data memory, *ALU*, registers, etc., into action systems of their own. The component action systems are joined together in a parallel composition, where they interact with each other using shared variables and remote procedure calls. The parallel composition of action systems is derived from the sequential specification using correctness-preserving program transformations within the refinement calculus.

The derivation is based on the novel idea of *spatial decomposition* of the program variables. At each step we identify one functional component of the microprocessor and gather the program variables and their associated code relevant to this component into a separate module, i.e., an action system. The approach is well supported by the refinement calculus. Back and Sere [7] show how this idea is reflected in a specification language based on action systems and the refinement calculus.

Martin [12] has developed a methodology for designing asynchronous VLSI circuits that is based on methods familiar from parallel program design. Using this method he has specified the same microprocessor within the CSP-framework, but without a completely formal calculus. A delay-insensitive, asynchronous circuit for the microprocessor was derived from the concurrent program that is more or less equivalent to the parallel composition of action systems that we derive here.

Our purpose here is to demonstrate that, in addition with software design, action systems and the refinement calculus provide us with a uniform framework for *formal*

VLSI circuit design. In this paper we concentrate on the initial steps of circuit design focusing on a high level specification of the microprocessor as a collection of parallel processes. In an accompanying paper [11] we develop these ideas close to the architectureal level by e.g. taking into account the delay-insensitive features of the target circuit.

A somewhat related method and formalism is developed in [16], but the emphasis is put on the verification of and formal models for delay-insensitive circuits.

1.1. Overview of the paper

In Section 2, we describe the action systems formalism. In Section 3, we describe how action systems are composed into parallel systems. We also briefly describe the refinement calculus. In Section 4, we give an initial specification for the microprocessor 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 functional component of the target microprocessor. Finally in Section 6, we conclude with some remarks on the proposed method.

2. Action systems

An *action system* is a statement of the form

$$\mathscr{A} :: \textbf{var } v; \textbf{proc } w \bullet$$
$$|[\ \textbf{var } x := x0; \textbf{proc } p_1 = P_1; \ldots; p_n = P_n; \textbf{do } A_1 [] \ldots [] A_m \textbf{od}\ ]| : z$$

The identifiers x are the variables declared in $\mathscr{A}$ and initialized to $x0$, $p_1, \ldots, p_n$ 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 $\mathscr{A}$. Finally, z, v and w are pairwise disjoint lists of identifiers. The list z is the *import list*, indicating which variables and procedures are referenced, but not declared in $\mathscr{A}$. The lists v and w are the *export lists*, indicating which variables and procedures declared in $\mathscr{A}$ are accessible from other action systems. Procedure bodies and actions can be arbitrary statements, and may contain procedure calls.

Both procedure bodies and actions will in general be *guarded commands*, i.e., statements of the form

$$A = g \rightarrow S,$$

where g is a boolean condition, the *guard*, and S is a program statement, the *body*. The guard of A will be denoted by gA and the body will be denoted by sA.

The *local* variables (procedures) of $\mathscr{A}$ are those variables x_i (procedures p_i) that are not listed in the export list. The *global* variables (procedures) of $\mathscr{A}$ are the variables (procedures) listed in the import and export lists. The local and global variables

(procedures) are assumed to be disjoint. Hence, $x \cap z = \emptyset$, where x denotes the list of variables declared in $\mathscr{A}$ (no redeclaration of variables is thus permitted). The *state variables* of $\mathscr{A}$ consist of the local variables and the global variables.

A statement or an action is said to be *local* to an action system if it only refers to local variables of the action system. The procedures and actions are allowed to refer to all the state variables of an action system. Furthermore, each procedure and action may have local variables of its own.

We consider two different parameter passing mechanisms for procedures, *call-by-value* and *call-by-result*. Call-by-value is denoted with $p(f)$, where f stands for the formal parameters, and call-by-result with $p(\mathbf{var}\, f)$. For simplicity, we will here assume that the procedures are not recursive.

3. Composing and refining action systems

Consider two action systems,

$$\mathscr{A} :: \mathbf{var}\ v; \mathbf{proc}\ r \bullet \\ \quad |[\ \mathbf{var}\ x := x0; \mathbf{proc}\ p = P; \mathbf{do}\ \ A_1\ [] \ldots [] \ A_m\ \mathbf{od}\ ]| : z$$

$$\mathscr{B} :: \mathbf{var}\ w; \mathbf{proc}\ s \bullet \\ \quad |[\ \mathbf{var}\ y := y0; \mathbf{proc}\ q = Q; \mathbf{do}\ \ B_1\ [] \ \ldots \ [] \ B_k\ \mathbf{od}\ ]| : u$$

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

The *parallel composition* $\mathscr{A} \parallel \mathscr{B}$ of $\mathscr{A}$ and $\mathscr{B}$ is the action system $\mathscr{C}$

$$\mathscr{C} :: \mathbf{var}\ b; \mathbf{proc}\ c \bullet \\ \quad |[\ \mathbf{var}\ x, y := x0, y0; \mathbf{proc}\ p = P; q = Q; \\ \quad\quad \mathbf{do}\ A_1\ [] \ldots [] \ A_m\ [] \ B_1\ [] \ldots [] \ B_k\ \mathbf{od} \\ \quad]| : a$$

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 $\mathscr{A}$ or $\mathscr{B}$. The exported identifiers are the variables and/or procedures declared global in $\mathscr{A}$ or $\mathscr{B}$. The procedure declarations and the actions in the parallel composition consists of the procedure declarations and actions in the original systems.

Parallel composition is a way of associating a meaning to procedures that are called in an action system but which are not declared there, i.e., they are part of the import list. The meaning can be given by a procedure declared in another action system,

provided the procedure has been declared global, i.e., it is included in the action systems export list.

The behaviour of a parallel composition of action systems is dependent on how the individual action systems, the *reactive components*, interact with each other via the shared global variables and remote procedure calls. We have for instance that a reactive component does not terminate by itself: termination is a global property of the composed action system. More on these topics can be found in [2].

3.1. Hiding and revealing

Let **var** v_1, v_2; **proc** $v_3, v_4 \bullet \mathscr{A} : z$ 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, $\mathscr{A}' =$ **var** v_1; **proc** $v_3 \bullet \mathscr{A} : z$. Hiding the variables v_2 and procedure names v_4 makes them inaccessible from other actions outside $\mathscr{A}'$ 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 use the following convention. Let **var** a_1; **proc** $a_2 \bullet \mathscr{A} : a_3$ and **var** b_1; **proc** $b_2 \bullet \mathscr{B} : b_3$ be two action systems. Then their parallel composition is the action system

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

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 $\mathscr{A}$ or $\mathscr{B}$. Sometimes there is no need to export all these identifiers, i.e., when they are exclusively accessed by the two component action systems $\mathscr{A}$ and $\mathscr{B}$. This effect is achieved with the following construct that turns out to be extremely useful later:

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

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

3.2. Decomposing action systems

Given an action system

$$\mathscr{C} :: \textbf{var } u; \textbf{proc } s \bullet |[\ \textbf{var } v := v0; \textbf{do } C_1 [] \ldots [] C_n \textbf{ od }\ ]| : z$$

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

$$\mathscr{C} = \textbf{var } u; \textbf{proc } s \bullet |[\ \textbf{var } w := w0; \textbf{proc } r = R; \mathscr{A} \parallel \mathscr{B}\ ]| : z$$

where

$$\mathscr{A} :: \textbf{var } a_2; \textbf{proc } a_3 \bullet$$
$$\quad |[\ \textbf{var } x := x0; \textbf{proc } p = P; \textbf{do } A_1 \,[]\, \ldots \,[]\, A_m \textbf{ od }\ ]| : a_1$$
$$\mathscr{B} :: \textbf{var } b_2; \textbf{proc } b_3 \bullet$$
$$\quad |[\ \textbf{var } y := y0; \textbf{proc } q = Q; \textbf{do } B_1 [] \ldots [] B_k \textbf{ od }\ ]| : b_1$$

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

In the process of decomposing the action system C into parallel reactive components, it may also be necessary to introduce some new procedures r, to handle situations where an action affects program variables in both x and y. As these variables are local in the decomposed action system, no procedure or action can access both. Hence, one needs to introduce auxiliary procedures that have access to the local variables, and in terms of which the original procedure/action can be expressed.

3.3. *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 another 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 $S \leqslant S'$, if for any postcondition Q

$$wp(S, Q) \Rightarrow wp(S', Q).$$

Here *wp* is the *weakest precondition* predicate transformer of Dijkstra [9]

We will not go into details of this calculus here. The interested reader should consult [1, 5, 15, 2, 6].

4. Initial specification of the Caltech microprocessor

The microprocessor we want to specify has a conventional 16-bit-word instruction set of *load-store* type. The processor uses two separate memories for instructions and data. There are three types of instructions: *ALU*, memory and program-counter (*pc*). The *ALU* instructions operate on the 16 registers. The memory instructions involve a register and a data word. Some instructions use the following word as *offset*.

The initial action system is a sequential non-terminating loop. The variable i holds the instruction under execution. It is of record type containing several fields. Each instruction has an *op* field for the *opcode*, the other fields depend on the instruction. The two memories are represented by the globally visible arrays *imem* and *dmem*.

```
M0 :: var imem, dmem •
      |[ var i ∈ record, pc, offset, imem[ilow..ihigh], dmem[dlow..dhigh],
           reg[0..15], f
         pc := pc0;
         do  true →
             i, pc := imem[pc], pc + 1;
             if offset(i.op) → offset, pc := imem[pc], pc + 1
              [] ¬offset(i.op) → skip
              fi;
             if alu(i.op) →< reg[i.z], f >:= aluf(reg[i.x], reg[i.y], i.op, f)
              [] ld(i.op) → reg[i.z] := dmem[reg[i.x] + reg[i.y]]
              [] st(i.op) → dmem[reg[i.x] + reg[i.y]] := reg[i.z]
              [] ldx(i.op) → reg[i.z] := dmem[offset + reg[i.y]]
              [] stx(i.op) → dmem[offset + reg[i.y]] := reg[i.z]
              [] lda(i.op) → reg[i.z] := offset + reg[i.y]
              [] stpc(i.op) → reg[i.z] := pc + reg[i.y]
              [] jmp(i.op) → pc := reg[i.y]
              [] brch(i.op) →
                 if cond(f, i.cc) → pc := pc + offset [] ¬cond(f, i.cc) → skip fi
             fi
         od
      ]|:<>
```

Fig. 1. Initial specification of the microprocessor.

The index to *imem* is the program-counter variable *pc*. The registers are described as the array *reg*[0..15]. The action system $\mathcal{M}_0$ in Fig. 1 describes the processor. Here the statement $< reg[i.z], f >:= aluf(reg[i.x], reg[i.y], i.op, f)$ denotes a simultaneous assignment of values to a pair of variables *reg*[*i.z*] and *f*.

5. Decomposition into parallel action systems

Let us decompose the action system $\mathcal{M}_0$ into a parallel composition of action systems so that each system models one functional component of the microprocessor. At each step one component is identified by its program variables. These variables and the associated code is gathered into a module of its own. Furthermore, we make a decision on how the variables of the module should be accessed, exporting the variables and accessing them as shared variables, or making them local and accessing them via global procedures.

The components in the order of their introduction and their associated variables are as follows:

(1) Instruction memory: *imem*[*ilow..ihigh*]

(2) Program counter and offset: *pc*, *offset*

(3) Register array: $reg[0..15]$
(4) Arithmetic-logical unit: f
(5) Data memory: $dmem[dlow..dhigh]$
(6) Instruction execution: i

The main lines of the derivation will follow the presentation of Martin [12] rather closely. We describe the first three steps more carefully, the other steps follow a similar pattern.

5.1. Instruction memory

We start by making the instruction memory an action system of its own. We assign the variable *imem* to become local to this action system and hence, references to *imem* must be done via a procedure call. There are two such references in the specification, one that writes $imem[pc]$ into the variable i and the other that writes $imem[pc]$ to *offset*.

Let us create a procedure *IMEM* that reads the instruction denoted by pc, $imem[pc]$, into a variable k (k will be later instantiated to i and *offset* respective):

$$
\begin{aligned}
& k, pc := imem[pc], pc + 1 \\
= \; & k := imem[pc]; pc := pc + 1 \\
\leqslant \; & \{\text{introducing local variables}\} \\
& |[\ \textbf{var}\ j; j := imem[pc]; k := j\ ]|; pc := pc + 1 \\
= \; & \{\text{introducing a procedure}\} \\
& |[\ \textbf{proc}\ IMEM(\textbf{var}\ j) = (j := imem[pc]); IMEM(k); pc := pc + 1\ ]|
\end{aligned}
$$

Hence, the action system $\mathscr{M}_0$ is refined by the following action system:

$$
\begin{aligned}
& \mathscr{M}_1 :: \textbf{var}\ imem, dmem\ \bullet \\
& \quad |[\ \textbf{var}\ i \in record, pc, offset, imem[ilow..ihigh], dmem[dlow..dhigh], \\
& \qquad reg[0..15], f \\
& \quad\ \textbf{proc}\ IMEM(\textbf{var}\ j) = (j := imem[pc]); \\
& \quad\ pc := pc0; \\
& \quad\ \textbf{do}\ true \rightarrow \\
& \qquad IMEM(i); pc := pc + 1; \\
& \qquad \textbf{if}\ offset(i.op) \rightarrow IMEM(offset); pc := pc + 1 \\
& \qquad [\!]\neg offset(i.op) \rightarrow skip \\
& \qquad \textbf{fi}\ ; \\
& \qquad \textbf{if}\ \ \ldots \text{as before} \ldots\ \textbf{fi} \\
& \quad\ \textbf{od} \\
& \quad]|: < >
\end{aligned}
$$

5.1.1. Separate FETCH and IMEM

The next step is to separate the instruction memory into an action system of its own. We therefore decompose the initial specification of the microprocessor as follows:

$$\mathscr{M}_1 = \textbf{var } imem, dmem \bullet |[\ \mathscr{M}_2 \parallel \mathscr{I}\ ]| : <>$$

where

$$\begin{array}{l}
\mathscr{M}_2 :: \textbf{var } pc, dmem \bullet \\
\quad |[\ \textbf{var } i \in record, pc, offset, dmem[dlow..dhigh], reg[0..15], f \\
\quad\ pc := pc0; \\
\quad\ \textbf{do } true \rightarrow \\
\quad\quad IMEM(i);\ pc := pc + 1; \\
\quad\quad \textbf{if } offset(i.op) \rightarrow IMEM(offset);\ pc := pc + 1 \\
\quad\quad [\!]\neg offset(i.op) \rightarrow skip \\
\quad\quad \textbf{fi}\ ; \\
\quad\quad \textbf{if}\ \ldots \text{as before} \ldots\ \textbf{fi} \\
\quad\ \textbf{od} \\
\quad]| : IMEM
\end{array}$$

$$\begin{array}{l}
\mathscr{I} :: \textbf{var } imem; \textbf{proc } IMEM \bullet \\
\quad |[\ \textbf{var } imem[ilow..ihigh]; \textbf{proc } IMEM(\textbf{var } j) = (j := imem[pc])\ ]| : pc
\end{array}$$

The instruction memory *imem* is now located in the module $\mathscr{I}$. The program counter, *pc*, is a shared variable between the two component modules. It is located in the module $\mathscr{M}_2$. Also the procedure *IMEM* has become global as it is accessed from $\mathscr{M}_2$. The instruction memory *imem* is not directly accessed in $\mathscr{M}_2$. However, both *pc* and *IMEM* are local to the parallel composition. Therefore this exports only the two memories, *imem* and *dmem*.

5.2. Program counter

Our next step is to isolate the program counter and offset administration from the rest of the processor. The program counter *pc* is referenced at instruction and offset fetch and during the execution of the *stpc*, *jmp*, and *brch* instructions.

Let us start by refining the *pc* updates at instruction fetch time as follows:

$$\begin{array}{ll}
 & IMEM(k);\ pc := pc + 1 \\
\leqslant & \{\text{introducing a local variable}\} \\
 & |[\ \textbf{var } y; IMEM(k);\ y := pc + 1;\ pc := y\ ]| \\
\leqslant & \{\text{commuting statements}\} \\
 & |[\ \textbf{var } y;\ y := pc + 1; IMEM(k);\ pc := y\ ]|
\end{array}$$

$= \{\text{introducing procedures}\}$
$\quad |[\ \textbf{var}\ y; \textbf{proc}\ PCI1 = (y := pc + 1); \textbf{proc}\ PCI2 = (pc := y)\ ]|;$
$\quad PCI1; IMEM(k); PCI2$

where y is a fresh variable. We have refined the pc access into a separate read-access and a write-access. This will allow us a parallel pc update and instruction fetch as will become clear below. The pc update at the *brch* instruction can be treated similarly:

$\quad pc := pc + offset$
$\leqslant \{\text{introducing a local variable}\}$
$\quad |[\ \textbf{var}\ z; z := pc + offset;\ pc := z\ ]|$
$= \{\text{introducing procedures}\}$
$\quad |[\ \textbf{var}\ z; \textbf{proc}\ PCA1 = (z := pc + offset); \textbf{proc}\ PCA2 = (pc := z)\ ]|$
$\quad PCA1; PCA2$

The other two pc accesses are read-accesses and hence, correspond to procedure calls.

Now it is a straightforward task to make the pc accesses via procedures:

$\mathscr{P} :: \textbf{var}\ pc; \textbf{proc}\ PCI1, PCI2, PCA1, PCA2, PCST, PCJMP \bullet$
$\quad |[\ \textbf{var}\ pc;$
$\quad\quad |[\ \textbf{var}\ y, z$
$\quad\quad\quad \textbf{proc}\ PCI1 = (y := pc + 1);$
$\quad\quad\quad \textbf{proc}\ PCI2 = (pc := y);$
$\quad\quad\quad \textbf{proc}\ PCA1 = (z := pc + offset);$
$\quad\quad\quad \textbf{proc}\ PCA2 = (pc := z);$
$\quad\quad\quad \textbf{proc}\ PCST(\textbf{var}\ o) = (o := pc);$
$\quad\quad\quad \textbf{proc}\ PCJMP(o) = (pc := o);$
$\quad\quad\ ]|;$
$\quad\quad pc := pc0;$
$\quad]| : offset$

Every access to pc is now done via these procedures. For instance, the instruction fetch and the subsequent pc-update $IMEM(i);\ pc := pc + 1$ is transformed to

$$PCI1; IMEM(i); PCI2.$$

Furthermore, the pc update $pc := pc + offset$ in the branch instruction *brch* is transformed to a pair of procedure calls

$$PCA1; PCA2.$$

The offset is read during the execution of the *ldx*, *stx*, and *lda* instructions. Furthermore, it is referenced at instruction fetch and during the program counter update at *brch* execution.

The *offset* value in the load and store instructions is received via a call to a procedure *XOFF* as follows:

$$
\begin{aligned}
& reg[i.z] := dmem[offset + reg[i.y]] \\
\leqslant\ & \{\text{introducing a local variable}\} \\
& |[\ \textbf{var}\ off; off := offset; reg[i.z] := dmem[off + reg[i.y]]\ ]| \\
=\ & \{\text{introducing a procedure}\} \\
& |[\ \textbf{proc}\ XOFF(\textbf{var}\ o) = (o := offset); \\
& |[\ \textbf{var}\ off; XOFF(off); reg[i.z] := dmem[off + reg[i.y]]\ ]|\ ]|
\end{aligned}
$$

The module for offset administration is defined next:

$$
\mathscr{X} :: \textbf{var}\ offset; \textbf{proc}\ XOFF\ \bullet\ |[\ \textbf{var}\ offset; \textbf{proc}\ XOFF(\textbf{var}\ o) = (o := offset)\ ]| : <>
$$

We now have that

$$
\mathscr{M}_2 = \textbf{var}\ dmem, pc\ \bullet\ |[\ \mathscr{M}_3 \parallel \mathscr{P} \parallel \mathscr{X}\ ]| : IMEM
$$

where $\mathscr{M}_3$ is given in Fig. 2.

```
M3 :: var dmem •
      |[ var i ∈ record, dmem[dlow..dhigh], reg[0..15], f
         do  true →
               PCI1; IMEM(i); PCI2;
               if offset(i.op) → PCI1; IMEM(offset); PCI2
               [] ¬offset(i.op) → skip
               fi;
               if alu(i.op) →< reg[i.z], f >:= aluf(reg[i.x], reg[i.y], i.op, f)
               [] ld(i.op) → reg[i.z] := dmem[reg[i.x] + reg[i.y]]
               [] st(i.op) → dmem[reg[i.x] + reg[i.y]] := reg[i.z]
               [] ldx(i.op) →
                    |[ var off; XOFF(off); reg[i.z] := dmem[off + reg[i.y]] ]|
               [] stx(i.op) →
                    |[ var off; XOFF(off); dmem[off + reg[i.y]] := reg[i.z] ]|
               [] lda(i.op) →
                    |[ var off; XOFF(off); reg[i.z] := off + reg[i.y] ]|
               [] stpc(i.op) → |[ var r; PCST(r); reg[i.z] := r + reg[i.y] ]|
               [] jmp(i.op) → |[ var y; y := reg[i.y]; PCJMP(y) ]|
               [] brch(i.op) →
                    if cond(f, i.cc) → PCA1; PCA2 [] ¬cond(f, i.cc) → skip fi
               fi
         od
      ]|: IMEM, PCI1, PCI2, PCA1, PCA2, PCST, PCJMP, XOFF, offset
```

Fig. 2. The action system $\mathscr{M}_3$.

Observe that *offset* is shared between $\mathcal{M}_3$, $\mathcal{P}$ and $\mathcal{X}$, whereas *pc* is shared between $\mathcal{M}_3$, $\mathcal{P}$ and $\mathcal{I}$. Therefore *offset* is a hidden variable. The variable *pc* is revealed from the parallel composition, because it is needed in the module $\mathcal{I}$.

5.3. Other components

Let us now briefly consider the registers, arithmetic-logical unit, data memory and the instruction execution. These modules are specified using similar argumentation as above.

5.3.1. Registers

First we isolate the register array from the rest of the program. The 16 registers are accessed through four buses in [12]. The buses are used by the *ALU* and the memory unit to concurrently access the registers. With this in mind we decompose our system further.

We define three procedures *REGRX*, *REGRY*, and *REGRZ* to read the value stored in a register, corresponding to the x, y and z fields of an instruction i. Furthermore, the *ALU* and the memory unit will use different procedures (buses), *REGWA* and *REGWM* respectively, to write on the registers. The instruction under consideration is kept in a shared variable j which is imported to the register modules from $\mathcal{M}_3$.

Let $\mathcal{R}$ be the action system

$$
\begin{aligned}
&\mathcal{R} :: \textbf{proc}\ REGRX, REGRY, REGRZ, REGWA, REGWM\ \bullet \\
&\quad |[\ \textbf{var}\ reg[0..15] \\
&\qquad \textbf{proc}\ REGRX(\textbf{var}\ v) = (v := reg[j.x]) \\
&\qquad \textbf{proc}\ REGRY(\textbf{var}\ v) = (v := reg[j.y]) \\
&\qquad \textbf{proc}\ REGRZ(\textbf{var}\ v) = (v := reg[j.z]) \\
&\qquad \textbf{proc}\ REGWA(v) = (reg[j.z] := v) \\
&\qquad \textbf{proc}\ REGWM(v) = (reg[j.z] := v) \\
&\quad]| : j
\end{aligned}
$$

This module represents the register array and also the four buses as will become clear later when we derive modules for the memory unit, the *ALU* and the instruction execution.

We now have that

$$
\begin{aligned}
&\mathcal{M}_3 = \textbf{var}\ dmem\ \bullet |[\ \mathcal{M}_4 \parallel \mathcal{R}\ ]| : \\
&\qquad IMEM, PCI1, PCI2, PCA1, PCA2, PCST, PCJMP, XOFF, offset
\end{aligned}
$$

where $\mathcal{M}_4$ is derived from $\mathcal{M}_3$ by the following changes. Each read-access to $reg[i.x]$ in $\mathcal{M}_3$ is replaced with a call to *REGRX* in $\mathcal{M}_4$, every read-access to $reg[i.y]$ is replaced with *REGRY* and every read-access to $reg[i.z]$ is replaced with *REGRZ*. A write-access to $reg[i.z]$ is replaced with a call to *REGWA* when the *ALU* writes this

register and with a call to *REGWM* when the access is made from the memory unit as will be seen below. The register array *reg*[0..15] is missing from $\mathcal{M}_4$.

The register module $\mathcal{R}$ and the action system $\mathcal{M}_4$ communicate via a shared variable j which is exported from $\mathcal{M}_4$. The variable j is assigned the value i immediately prior the execution of the instruction kept in i. This effect is achieved in $\mathcal{M}_4$ by refining $\mathcal{M}_3$ as follows:

if *offset*(*i.op*) → ... as before ... **fi** ;
if *alu*(*i.op*) → ... as before ... **fi**
⩽ {introducing a local variable}
if *offset*(*i.op*) → ... as before ... **fi** ;
$j := i$;
if *alu*(*i.op*) → ... as before ... **fi**

The variable j is needed further on when generating a module for the instruction execution. In the full paper [4] we refine the registers further in order to allow more parallelism. In the final configuration each of the 16 registers constitutes a module of its own.

5.3.2. Arithmetic-logical unit

Our following task is to isolate the arithmetic-logical unit. This unit is accessed in the *alu* instruction execution. As mentioned above, it has its own bus to access the register array, modelled by the procedure *REGWA*. Hence, this piece of code is refined as follows:

$< reg[i.z], f >:= aluf(reg[i.x], reg[i.y], i.op, f)$
⩽ {introducing register procedures, from above}
|[**var** x, y; $REGRX(x), REGRY(y)$;
|[**var** v; $< v, f >:= aluf(x, y, i.op, f)$; $REGWA(v)$]|]|

from where it is a straightforward task to generate the module for the arithmetic-logical unit:

$\mathcal{A}$ :: **proc** $ALU, ALUF$ •
|[**var** f
proc $ALU(u, w, op) =$
(|[**var** v; $< v, f >:= aluf(u, w, op, f)$; $REGWA(v)$]|);
proc $ALUF$(**var** e) $= (e := f)$
]| : $REGWA$

The *ALUF* procedure is used during the *brch* execution to read the value of the flag f.

We now have that

$$\mathscr{M}_4 = \textbf{var } dmem \bullet |[\ \mathscr{M}_5 \parallel \mathscr{A}\ ]| : IMEM, PCI1, PCI2, PCA1, PCA2, PCST, PCJMP, XOFF, offset$$

where for instance the above *ALU* reference in $\mathscr{M}_4$ is transformed to a call to the *ALU* unit as follows:

$$\begin{aligned} & < reg[i.z], f > := aluf(reg[i.x], reg[i.y], i.op, f) \\ \leqslant\ & \{\text{introducing procedures from above}\} \\ & |[\ \textbf{var } x, y; REGRX(x); REGRY(y); ALU(x, y, i.op)\ ]| \ . \end{aligned}$$

5.3.3. Data memory

Let us now consider the data memory, *dmem*. It also has its own bus to access the registers, modelled by the procedures *REGRZ* and *REGWM*. The data memory is read during the execution of the *ld* and *ldx* instructions and it is written during the store instructions *st* and *stx*. In the final implementation, the execution of the *lda* instruction is also carried out via the data memory unit. Let us look at the *ld* and *st* instructions more carefully.

We have for the load instruction that

$$\begin{aligned} & reg[i.z] := dmem[reg[i.x] + reg[i.y]] \\ \leqslant\ & \{\text{introducing register procedures, from above}\} \\ & |[\ \textbf{var } x, y; REGRX(x); REGRY(y); \\ & |[\ \textbf{var } v; v := dmem[x + y]; REGWM(v)\ ]|\ ]| \end{aligned}$$

and for the store instruction that

$$\begin{aligned} & dmem[reg[i.x] + reg[i.y]] := reg[i.z] \\ \leqslant\ & \{\text{introducing register procedures, from above}\} \\ & |[\ \textbf{var } x, y; REGRX(x); REGRY(y); \\ & |[\ \textbf{var } v; REGRZ(v); dmem[x + y] := v\ ]|\ ]| \end{aligned}$$

We now define

$$\begin{aligned} \mathscr{D} :: & \textbf{var } dmem; \textbf{proc } MADD, MSTO, MLDA \bullet \\ & |[\ \textbf{var } dmem[dlow..dhigh] \\ & \quad \textbf{proc } MADD(u, w) = (|[\ \textbf{var } v; v := dmem[u + w]; REGWM(v)\ ]|); \\ & \quad \textbf{proc } MSTO(u, w) = \\ & \qquad (\textbf{return}; |[\ \textbf{var } v; REGRZ(v); dmem[u + w] := v\ ]|); \\ & \quad \textbf{proc } MLDA(u, w) = (|[\ \textbf{var } ma; ma := u + w; REGWM(ma)\ ]|); \\ &]| : REGWM, REGRZ \end{aligned}$$

The memory is now represented by the following module:

$$\textbf{var } dmem; \textbf{proc } MADD, MSTO, MLDA \bullet \mathscr{D} : REGWM, REGRZ$$

which exports the three memory access procedures *MADD*, *MSTO*, and *MLDA* and imports the bus, i.e., the procedures *REGWM* and *REGRZ*.

The memory unit $\mathscr{D}$ is now removed from the rest of the code in module $\mathscr{M}_5$. Therefore, we have that

$$\begin{aligned}\mathscr{M}_5 = \textbf{var } dmem \bullet |[\ \mathscr{M}_6 \parallel \mathscr{D}\ ]| :\\ IMEM, PCI1, PCI2, PCA1, PCA2, PCST, PCJMP, XOFF, offset,\\ REGWM, REGRZ\end{aligned}$$

In $\mathscr{M}_6$ we have replaced the direct memory accesses with the appropriate procedure calls, for instance the above refined load and store instructions are transformed to

$$\begin{aligned}&reg[i.z] := dmem[reg[i.x] + reg[i.y]]\\ \leqslant\ &\{\text{introducing procedures from above}\}\\ &|[\ \textbf{var } x, y; REGRX(x); REGRY(y); MADD\ (x, y)\ ]|\end{aligned}$$

and

$$\begin{aligned}&dmem[reg[i.x] + reg[i.y]] := reg[i.z]\\ \leqslant\ &\{\text{introducing procedures from above}\}\\ &|[\ \textbf{var } x, y; REGRX(x); REGRY(y); MSTO(x, y)\ ]|\end{aligned}$$

in $\mathscr{M}_6$ respectively.

A slightly more optimized version of the memory unit is derived in the full paper [4].

5.3.4. Instruction execution

We next isolate the instruction execution into a separate module. The code for this module, $\mathscr{E}$, is given in Fig. 3.

The instruction under execution is in this module represented by the variable j, which is shared with the register array. The module uses two buses, modelled by the procedures *REGRX* and *REGRY* respective, for additional communication with the registers. We have replaced all the register, *pc*, *offset*, memory, and *ALU* references with appropriate procedure calls. Observe that the *pc* update during *stpc* execution is carred out via a call to *ALU*.

This gives us the system

$$\begin{aligned}\mathscr{M}_6 = \textbf{var } j \bullet |[\ \mathscr{M}_7 \parallel \mathscr{E}\ ]| :\\ IMEM, PCI1, PCI2, PCA1, PCA2, PCST, PCJMP, XOFF,\\ REGRX, REGRY, ALU, ALUF, MADD, MSTO, MLDA\end{aligned}$$

$$
\begin{array}{l}
\mathcal{E} :: \ \textbf{var } j; \textbf{ proc } EXEC \bullet \\
\quad [\![\ \textbf{var } j \in record \\
\quad\quad \textbf{proc } EXEC(k) = \\
\quad\quad\quad (j := k; \\
\quad\quad\quad\ \textbf{if } alu(k.op) \rightarrow \textbf{return}; \\
\quad\quad\quad\quad\ [\![\ \textbf{var } x, y;\ REGRX(x);\ REGRY(y);\ ALU(x, y, k.op) \ ]\!] \\
\quad\quad\quad\ [\!] \ ld(k.op) \rightarrow \textbf{return}; \\
\quad\quad\quad\quad\ [\![\ \textbf{var } x, y;\ REGRX(x);\ REGRY(y);\ MADD(x, y) \ ]\!] \\
\quad\quad\quad\ [\!] \ st(k.op) \rightarrow \textbf{return}; \\
\quad\quad\quad\quad\ [\![\ \textbf{var } x, y;\ REGRX(x);\ REGRY(y);\ MSTO(x, y) \ ]\!] \\
\quad\quad\quad\ [\!] \ ldx(k.op) \rightarrow \\
\quad\quad\quad\quad\ [\![\ \textbf{var } off, y;\ XOFF(off);\ REGRY(y);\ MADD(off, y) \ ]\!] \\
\quad\quad\quad\ [\!] \ stx(k.op) \rightarrow \\
\quad\quad\quad\quad\ [\![\ \textbf{var } off, y;\ XOFF(off);\ REGRY(y);\ MSTO(off, y) \ ]\!] \\
\quad\quad\quad\ [\!] \ lda(k.op) \rightarrow \\
\quad\quad\quad\quad\ [\![\ \textbf{var } off, y;\ XOFF(off);\ REGRY(y);\ MLDA(off, y) \ ]\!] \\
\quad\quad\quad\ [\!] \ stpc(k.op) \rightarrow \\
\quad\quad\quad\quad\ [\![\ \textbf{var } r, y;\ PCST(r);\ REGRY(y);\ ALU(r, y, add) \ ]\!] \\
\quad\quad\quad\ [\!] \ jmp(k.op) \rightarrow [\![\ \textbf{var } y;\ REGRY(y);\ PCJMP(y) \ ]\!] \\
\quad\quad\quad\ [\!] \ brch(k.op) \rightarrow [\![\ \textbf{var } ff;\ ALUF(ff); \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{if } cond(ff, k.cc) \rightarrow PCA1;\ PCA2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\ [\!] \ \neg cond(ff, k.cc) \rightarrow skip \\
\quad\quad\quad\quad\quad\quad\quad\quad\ \textbf{fi} \ ]\!] \\
\quad\quad\quad \textbf{fi}) \\
\quad]\!]: PCA1, PCA2, PCST, PCJMP, XOFF, REGRX, REGRY, \\
\quad\ ALU, ALUF, MADD, MSTO, MLDA
\end{array}
$$

Fig. 3. Instruction execution.

where the execution of an instruction in the variable i in $\mathcal{M}_7$ is now initiated via a procedure call

$$EXEC(i).$$

5.4. *Refine the fetch-and-execute cycle*

Finally, we refine the fetch and execute cycle to make parallel instruction fetch and execution possible. In our framework, as mentioned earlier, we have to create independent actions in order to make parallel activity possible. This calls for atomicity refinement.

5.4.1. *Create FETCH*

Let us first collect all the transformations above, and see what is left of the action system $\mathcal{M}_3$, i.e., the system $\mathcal{M}_7$ above. The procedures *EXEC* and its related *ALU* together with the instruction and data memories, register array, and p and *offset*

administration were all isolated into action systems of their own, separate from $\mathcal{M}_3$, leaving only the appropriate procedure calls behind. The result is the system $\mathcal{M}_7$ that will be from here on called $\mathcal{F}_0$ where

$$
\begin{aligned}
\mathcal{F}_0 :: \; & |[\ \textbf{var}\ i \in record \\
& \quad \textbf{do}\ true \rightarrow \\
& \qquad PCI1; IMEM(i); PCI2; \\
& \qquad \textbf{if}\ offset(i.op) \rightarrow PCI1; IMEM(offset); PCI2 \\
& \qquad [\!]\ \neg offset(i.op) \rightarrow skip \\
& \qquad \textbf{fi}\ ; \\
& \qquad EXEC(i) \\
& \quad \textbf{od} \\
&]| : offset, IMEM, PCI1, PCI2, EXEC
\end{aligned}
$$

In this system there is only one atomic action and no parallelism is possible. Hence, we split the action into two distinct parts so that $\mathcal{F}_0 \leqslant \mathcal{F}_1$ where

$$
\begin{aligned}
\mathcal{F}_1 :: \; & |[\ \textbf{var}\ i \in record \\
& \quad \textbf{do} \\
& \quad < PCI1; IMEM(i); PCI2; \\
& \quad \textbf{if}\ offset(i.op) \rightarrow PCI1; IMEM(offset); PCI2 \\
& \quad [\!]\neg offset(i.op) \rightarrow skip \\
& \quad \textbf{fi}\ >; \\
& \quad < EXEC(i) > \\
& \quad \textbf{od} \\
&]| : offset, IMEM, PCI1, PCI2, EXEC.
\end{aligned}
$$

We have used sequential notation for $\mathcal{F}_1$ by denoting the atomicity of actions explicitly with brackets.

We have that $\mathcal{F}_1$ and $\mathcal{E}$ share no variables. They communicate through the global procedure *EXEC* only. When looking into the specification of the procedure *EXEC* we notice, that when we are executing an *ALU*, load or store instruction, the control returns to $\mathcal{F}_1$ immediately after the call of *EXEC* due to the **return** statements. At this point the next instruction is fetched from the instruction memory. Hence, the execution of these three instructions in $\mathcal{E}$ can proceed in parallel with the fetch of the next instruction in $\mathcal{F}_1$.

6. Concluding remarks

We have created the action system $\mathcal{M}_8$

$$\mathcal{M}_8 :: \textbf{var}\ imem, dmem \bullet |[\ \mathcal{I} \parallel \mathcal{F}_1 \parallel \mathcal{P} \parallel \mathcal{X} \parallel \mathcal{E} \parallel \mathcal{A} \parallel \mathcal{D} \parallel \mathcal{R}\ ]| : < >$$

that is by construction a correct refinement of the initial high level microprocessor specification $\mathcal{M}_0$, i.e.,

$$\mathcal{M}_0 \leqslant \mathcal{M}_8.$$

At Caltech, a delay-insensitive circuit is derived from a concurrent program that is essentialy the same as our resulting action system [12]. The advantage of our method is that it is based on a formal calculus for reasoning about programs, the refinement calculus.

The main method used throughout our derivation was the spatial decomposition of an action system into a parallel composition of action systems that mainly communicate via (remote) procedure calls. Hence, most of the steps we carried out are correct by construction. Only a couple of steps required more tedious proofs, i.e., those where the atomicity of the system was refined.

When we compare our system to that in [12] there are a couple of notions that are implicit in an action system. The bullet operator used in [12] corresponds to an action in the sense that when an action is chosen for execution, it is jointly executed to completion by the involved modules without interference from other actions. The point of termination for an action need not coincide for every module involved in it as long as atomicity is guaranteed. The probes in [12] are here modelled by the interplay between the caller and the callee while making procedure calls.

References

[1] R.J.R. Back, On the correctness of refinement steps in program development, Ph.D. 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, ed., *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proc. 1989*, Lecture Notes in Computer Science, Vol. 430 (Springer, Berlin, 1990).

[3] R.J.R. Back and R. Kurki-Suonio, Decentralization of process nets with centralized control, in: *Proc. ACM SIGACT–SIGOPS Symp on Principles of Distributed Computing* (1983) 131–142.

[4] R.J.R. Back, A.J. Martin and K. Sere, Specification of a microprocessor, Tech. Report, Åbo Akademi University, Department of Computer Science. Ser. A, No 148, Turku, Finland, 1992.

[5] R.J.R. Back and K. Sere, Stepwise refinement of parallel algorithms, *Sci. Comput. Programming* **13** (1989) 133–180.

[6] R.J.R. Back and K. Sere, Action systems with synchronous communication, in:E.-R. Olderog, ed., Proc. *PROCOMET'94*, San Miniato, Italy, June 1994. *Programming Concepts, Methods and Calculi*, IFIP Trans. A-56 (North-Holland, Amsterdam, 1994) 107–126.

[7] R.J.R. Back and K. Sere, From modular systems to action systems, in: Proc. *Formal Methods Europe'94*, Spain, October 1994, Lecture Notes in Computer Science (Springer, Berlin, 1994).

[8] K. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison–Wesley, Reading, MA, 1988).

[9] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[10] N. Francez, Cooperating proofs for distributed programs with multiparty interactions, *Inform. Processing Lett.* **32** (1989) 235–242.

[11] T. Kuusela, J. Plosila, R. Ruksenas, K. Sere and Zhao Yi, Designing delay-insensitive circuits within the action systems framework, Manuscript, 1995.

[12] A.J. Martin, Synthesis of asynchronous VLSI circuits, CalTech, Tech. Report, 1993.

[13] C.C. Morgan, The specification statement, *ACM Trans. Programming Languages and Systems* **10** (1988) 403–419.
[14] J.M. Morris, A theoretical basis for stepwise refinement and the programming calculus, *Sci. Comput. Programming* **9** (1987) 287–306.
[15] K. Sere, Stepwise refinement of parallel algorithms, Ph.D. Thesis, Department of Computer Science, Åbo Akademi University, Turku, Finland, 1990.
[16] J. Staunstrup and M.R. Greenstreet, Synchronized transitions, in: IFIP WG 10.5, Summer School on Formal Methods for VLSI Design, Lecture Notes (1990).