# Synchronous Design Aspects Using Action Systems

Juha Plosila* Tiberiu Seceleanu†

October 5, 1998

**Abstract**

The action systems framework has recently been applied to the area of synchronous VLSI design. In this paper, we present a set of concepts necessary in the formal design of synchronous hardware, using synchronous action systems. These explicitly model synchronous two-phase behavior. They are developed using the refinement calculus in a usual manner. The fundamental idea of synchronous action systems is to make a formal description transparent in the sense that its hardware implementation can be obtained by direct mapping. The presented design methodology also allows a postponed decision on the final implementation aspects regarding the architecture (synchronous / asynchronous)

## 1 Introduction

The majority of modern digital hardware uses synchronous techniques, where sequencing of events is based on the usage of a common timing signal, called the clock. Although the asynchronous self-timed approach has become an interesting option in various applications, especially in low-power design, the synchronous approach will be the mainstream circuit design method also in years to come. The basic reason for this is the fact that the design experience base and CAD tool support for synchronous hardware are outstanding. Furthermore, a synchronous circuit seems to be equally fast or somewhat faster, and certainly smaller, than an asynchronous equivalent circuit [9].

Formal methods of concurrent programming can be applied to the area of circuit design to ensure mathematically logical correctness of the target circuit. However, these methods, our *action systems* framework included, are inherently

*University of Turku, Dep. of Applied Physics, Lab. of Electronics and Information Technology, FIN-20014 Turku, Finland, email: juplos@utu.fi

†Turku Center for Computer Science (TUCS), Lemminkäisenkatu 14, FIN-20520 Turku, Finland, tel: +358-2-265 4084, fax: +358-2-2410154, email: tsecelea@abo.fi

well-suited for describing asynchronous behavior, but they are usually not so suitable for modeling synchronous two-phase behavior, where read and write phases are explicitly separated. In one previous work [19], we have presented the concept of *two-phase actions* as the basis for synchronous two-phase modeling within the action systems formalism, as a functional, untimed approach. An action system which contains a *synchronous composition* of two-phase actions is called a *synchronous action system*. We have also shown the basic principles how to implement a synchronous action system as a network of synchronous circuit primitives called *basic cells*.

In this paper, we continue our work on synchronous action systems and their hardware implementation. Generally, a circuit design process includes the translation of abstract specifications of programs into efficient circuits which compute these programs [6]. As mentioned above, the action systems formalism is very well featured for derivation of asynchronous descriptions. We share the opinion that they are a natural and convenient way to specify behavior of complex digital hardware [14]. Therefore, the initial specification of a target circuit is usually given in an asynchronous form.

Further, in applying formal methods to hardware design one can distinguish two main approaches [12]:

1. *post-design verification*

2. *correctness-preserving transformations*

both with advantages and disadvantages, which we don't analyze here (however, it is known that verifying the correctness of a large circuit by simulation is impractical [23]).

Action systems formalism uses correctness-preserving transformations in order to derive system implementation from higher level specifications. We show how an initial asynchronous specification can be transformed within the *refinement calculus* into a synchronous one, which can be then implemented as a digital circuit, using custom methods or existing CAD tools. The synchronous interpretations of the most essential program constructs are given. As an extension to the study on circuit mapping of synchronous action systems [19], we also discuss circuit implementation of program constructs.

An action system description is systematically transformed into a format that supports automatization. When a concrete enough level is reached, one can feed the system description into existent tools for further processing towards the final target, the actual device. We have created a computer program that converts a synchronous action system description into the corresponding VHDL code [20]. Hence, our program can read the final formal description of a synchronous action system. After the automatic conversion, the resulting code can be then input into a VHDL synthesizer, which does the actual technology mapping.

**Overview of the paper** We proceed as follows. In Section 2 we briefly revise the key points of the action systems formalism. Then, in Section 3 we present in more detail the basics of the recently introduced synchronous action systems. The rules for deriving a synchronous description from an asynchronous one are stated in Section 4, followed, in Section 5, by hardware implementation issues. Section 6 presents an actual example for synchronous design in the action systems framework. We end with some concluding remarks in Section 7.

## 2 Action systems

The action systems formalism is based on an extended version of Dijkstra's language of *guarded commands* [7]. This language includes assignment, sequential composition, conditional choice, and iteration, and is defined using *weakest precondition* predicate transformers. Comprehensive study on this formalism can be found elsewhere [1, 4].

**Actions** An *action* $A$ is defined by

$$
\begin{array}{lll}
A & ::= & \mathit{abort} & \text{(abortion, nontermination)} \\
& | & \mathit{skip} & \text{(empty statement)} \\
& | & x := x'.R & \text{(nondeterministic assignment)} \\
& | & [P] & \text{(assumption)} \\
& | & A_1 \mathbin{[\!]} A_2 & \text{(nondeterministic choice)} \\
& | & A_1;\ A_2 & \text{(sequential composition)} \\
& | & |[\ \mathbf{var}\ x := x0;\ A\ ]| & \text{(block with local variables)} \\
& | & P \rightarrow A \mathrel{\widehat{=}} [P];\ A & \text{(guarded action)}
\end{array}
$$

where $P$ and $R$ are predicates (boolean conditions), $x$ a variable or a list of variables, $x0$ some value(s) of the variable(s) $x$, and $e$ an expression or a list of expressions.

The actions are defined using weakest precondition for predicate transformers [7]. For instance:

$$
\begin{array}{rcl}
\mathrm{wp}(\mathit{abort}, Q) & = & \mathit{false} \\
\mathrm{wp}(\mathit{skip}, Q) & = & Q \\
\mathrm{wp}(x := x'.R, Q) & = & (\forall x'.R \Rightarrow Q[x'/x]) \\
\mathrm{wp}(x := e, Q) & = & Q[e/x] \\
\mathrm{wp}((A_1 \mathbin{[\!]} A_2), Q) & = & \mathrm{wp}(A_1, Q) \wedge \mathrm{wp}(A_2, Q) \\
\mathrm{wp}((A_1;\ A_2), Q) & = & \mathrm{wp}(A_1, \mathrm{wp}(A_2, Q)) \\
\mathrm{wp}(P \rightarrow A, Q) & = & P \Rightarrow \mathrm{wp}(A, Q) \\
\mathrm{wp}(|[\ \mathbf{var}\ x := x0;\ A\ ]|, Q) & = & \forall x.\mathrm{wp}(A, Q)
\end{array}
$$

Total correctness of an action $A$ with respect to a *precondition* $P$ and a *post-condition* $Q$ is defined as

$$P \; \{A\} \; Q \quad \widehat{=} \quad P \Rightarrow \mathrm{wp}(A, Q).$$

Actions are considered *atomic*, which means that whenever an action is selected for execution, it will be completed without interference. The *scope of atomicity* is often denoted by the angle brackets $<>$. For example: $< b \rightarrow A_1; A_2 >$. We often use angle brackets for distinguish an action separately from a composition, too.

The *guard* $gA$ and the *body* $sA$ of an action $A$ are defined by

$$gA \mathrel{\widehat{=}} \neg \mathrm{wp}(A, \mathit{false}) \quad , \quad sA \mathrel{\widehat{=}} A \; [] \; \neg gA \rightarrow \mathit{abort}$$

The guard of a guarded action $b \rightarrow A$ is given as:

$$b \wedge gA.$$

An action is said to be *enabled* when its guard is *true*, *disabled* otherwise. If the guard is invariantly *true*, the action is *always enabled*.

In addition to the sequential composition of actions and the nondeterministic choice, we use the following constructs:

- *Simultaneous composition of actions:* Assume that $A_1$ and $A_2$ are actions of the form $< b_i \rightarrow v_i := v_i'.Q_i >$, $i = 1, 2$, then the simultaneous composition of $A_1$ and $A_2$, denoted $A_1 \star A_2$, is defined as

$$A_1 \star A_2 \quad \widehat{=} \quad < \bigwedge_{i=1}^{2} b_i \rightarrow v_1, v_2 := v_1', v_2'. \bigwedge_{i=1}^{2} Q_i > .$$

- *Prioritizing composition of actions*, denoted $A_1 \; // \; A_2$, is defined as

$$A_1 \; // \; A_2 \quad \widehat{=} \quad A_1 \; [] \; < \neg gA_1 \rightarrow A_2 >$$

  The extension of the prioritized composition to action systems is defined and presented in details in [21].

The scope of a constructor is indicated with parenthesis, for example

$$A; ((B \; [] \; C) \; // \; E).$$

**Action systems** An *action system* has the form:

$$\begin{array}{l} \mathbf{sys}\mathcal{A}\,(g) :: \\ |[ \; \mathbf{var} \; l; \; \mathbf{init} g, l := g_0, l_0; \\ \quad \mathbf{do} \; A_1 \; [] \; \ldots \; [] \; A_m \mathbf{od} \\ \; ]| \end{array}$$

where $g$, and $l$ are lists of identifiers initialized to $g_0$ and $l_0$, respectively. The identifiers $l$ are the *local* variables visible only within $\mathcal{A}$. The identifiers $g$, in turn, are the *global* variables of $\mathcal{A}$ visible to other action systems as well. The local and global variables are assumed to be distinct. The actions $A_i$ of $\mathcal{A}$ are allowed to refer to all of the *state variables* consisting of the local and global variables. If the system $\mathcal{A}$ does not have any global variables, it is called a *closed* action system, *open* otherwise.

The action system $\mathcal{A}$ operates as follows. After the initialization of the state variables, an enabled action $A_i$ is non-deterministically selected and executed. This is then repeated as long as there are enabled actions present. When every action of the system is disabled, the execution terminates. Because the actions are considered atomic, two actions $A_i$ and $A_j$ that are enabled at the same time and do not have any read-write conflicts can be executed in any order. Hence, we can model parallel behavior with action systems taking the view of interleaving action executions.

**Parallel composition** Consider two action systems $\mathcal{A}$ and $\mathcal{B}$

$$\begin{array}{ll}
\mathbf{sys}\mathcal{A}\,(g_A) :: & \mathbf{sys}\mathcal{B}\,(g_B) :: \\
|[\ \mathbf{var}\ l_A;\ \mathbf{init} g_A, l_A := g_{A0}, l_{A0}; & |[\ \mathbf{var}\ l_B;\ \mathbf{init} g_B, l_B := g_{B0}, l_{B0}; \\
\quad \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_m \mathbf{od} & \quad \mathbf{do}\ B_1\ [\!]\ \ldots\ [\!]\ B_n \mathbf{od} \\
\ ]| & \ ]|
\end{array}$$

where $l_A \cap l_B = \emptyset$, and the initializations of the global variables $g_A \cap g_B \neq \emptyset$ in the systems $\mathcal{A}$ and $\mathcal{B}$ are consistent with each other (the shared global variables are initialized with the same values, in both systems). The *parallel composition* of $\mathcal{A}$ and $\mathcal{B}$, denoted $\mathcal{A} \parallel \mathcal{B}$, is the action system

$$\begin{array}{l}
\mathbf{sys}\mathcal{C}\,(g_A \cup g_B) :: \\
|[\ \mathbf{var}\ l_A \cup l_B;\ \mathbf{init} g_A \cup g_B, l_A \cup l_B := g_{A0} \cup g_{B0}, l_{A0} \cup l_{B0}; \\
\quad \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_m\ [\!]\ B_1\ [\!]\ \ldots\ [\!]\ B_n \mathbf{od} \\
\ ]|
\end{array}$$

Thus, parallel composition combines the state spaces of the constituent action systems keeping the local variables $l_A$ and $l_B$ distinct. The *reactive components* $\mathcal{A}$ and $\mathcal{B}$ interact with each other via the global variables that are referenced in both components. Termination of computation is a global property of the composed action system $\mathcal{C}$.

The definition of the parallel composition is used inversely to *decompose* an action system into a parallel composition of smaller systems.

## 2.1 Refinement of action systems

Action systems are intended to be developed in a stepwise manner within the *refinement calculus*. The action $A$ is *( correctly ) refined* by action $A$, denoted

$A \leq A'$, if

$$\forall Q.(wp(A,Q) \Rightarrow wp(A',Q)),$$

This is equivalent to the condition:

$$\forall P,Q : P\{A\}Q \Rightarrow P\{A'\}Q$$

In practice refinement is obtained using dedicated *transformation rules* rather than the above definition itself. The details of the refinement calculus can be found elsewhere [2].

**Data refinement of actions** Let $A$ be an action on program variables $a$ and $u$, and let $C$ be an action on variables $c$ and $u$. Assume that $R(a,c)$ is a relation between the variables $a$ and $c$. Then the abstract action $A$ is *data-refined* by the concrete action $C$ using the *abstraction relation* $R(a,c)$, denoted $A \leq_R C$, if

$$\forall Q.(R \wedge \mathrm{wp}(A,Q) \Rightarrow \mathrm{wp}(C, \exists a.R \wedge Q))$$

where $(\exists a.R \wedge Q)$ is a predicate on the program variables $c$ and $u$. The above refinement condition can be further decomposed into:

1. Guard: $R \wedge gC \Rightarrow gA$,
2. Body: $\forall Q.(R \wedge gC \wedge \mathrm{wp}(sA,Q) \Rightarrow \mathrm{wp}(sC, \exists a.R \wedge Q))$.

**Refinement of action systems** Consider action systems

| | |
|---|---|
| $\mathbf{sys}\mathcal{A}\,(u)$ :: | $\mathbf{sys}\mathcal{C}\,(u)$ :: |
| $\vert[$ **var** $a$; **init**$u, a := u_0, a_0$; | $\vert[$ **var** $c$; **init**$c, u := c_0, u_0$; |
| **do** $A_1 \,[\!]\, \ldots \,[\!]\, A_m$**od** | **do** $C_1 \,[\!]\, \ldots \,[\!]\, C_n \,[\!]\, X_1 \,[\!]\, \ldots \,[\!]\, X_m$**od** |
| $]\vert$ | $]\vert$ |

The system $\mathcal{C}$ is obtained from $\mathcal{A}$ by replacing the local variables $a$ with new local variables $c$, the actions $A_i$ with $C_i$, and adding a set of auxiliary actions $X_j$ into the system. The global variables $u$ are not changed. The following lemma is then used to prove (trace) refinement of the action system $\mathcal{A}$.

*LEMMA 1. The action system $\mathcal{A}$ is (trace) refined by the system $\mathcal{C}$, denoted $A \sqsubseteq C$, if there exists an abstraction relation $R(a,c)$ such that:*

1. Initialization (the abstraction relation is established by the initializations):

$$R(a_0, c_0)$$

2. Main actions (each action $A_i$ is data-refined by the corresponding action $C_i$ using $R(a,c)$):

$$A_i \leq_R C_i, i = 1, \ldots, n$$

3. Auxiliary actions (the auxiliary actions $X_j$ behave like *skip* with respect to the global variables $u$ while preserving $R(a, c)$):

$$skip \leq_R X_j, j = 1, \ldots, m$$

4. Continuation condition (an action in $\mathcal{C}$ is enabled whenever an action in $\mathcal{A}$ is enabled and $R(a, c)$ holds):

$$R \wedge g\mathcal{A} \Rightarrow g\mathcal{C}$$

5. Internal convergence (the execution of the auxiliary actions taken separately cannot continue forever whenever $R(a, c)$ holds):

$$R \Rightarrow \mathrm{wp}(\mathbf{do}\ X_1\ [\!]\ \ldots\ [\!]\ X_m \mathbf{od}, true)$$

The lemma follows from the fact that the abstraction relation $R$ is local, i.e., it refers only to the local variables $a$ and $u$, and from the theory of trace refinement of action systems [3].

# 3 Synchronous action systems

In this section, we revise the concepts related to synchronous action systems. A more fundamental study on these subjects can be found in [19].

## 3.1 Two-phase actions

Consider an action $A$ of the form $< b \rightarrow v := v'.Q >$. The *read* and *write parts* of A, denoted $R_A$ and $W_A$, respectively are defined as

$$\begin{aligned} R_A &\ \widehat{=}\ < u := u'.(b \Rightarrow Q[u'/v']) \wedge (\neg b \Rightarrow u' = u) >, \\ W_A &\ \widehat{=}\ < v := u > \end{aligned}$$

where $u$ is an auxiliary intermediate variable of the same type as $v$. The *clock operator*, denoted '$\Diamond$' is then defined by

$$R_A \Diamond W_A \ \widehat{=}\ |[\ \mathbf{var}\ p : boolean;\ < \neg p \rightarrow R_A;\ p := true >\ [\!]\ < p \rightarrow W_A;\ p := false >\ ]|$$

where the sequencing boolean variable $p$ models the behavior of the clock. The *two-phase representation* of the original action $A$, or a *two-phase action* in short, is denoted by $T_A$ and given as:

$$T_A \widehat{=} |[\ \mathbf{var}\ u;\ R_A \Diamond W_A\ ]|$$

### 3.2 Synchronous composition of actions

Consider actions $A_i =< b_i \rightarrow v_i := v_i'.Q_i >,\ i = 1..n$, where $v_i$ are distinct variables. The *synchronous composition* of the actions $A_i$, denoted $A_1 \nabla \ldots \nabla A_n$, is defined in terms of the read and write parts of the component actions as

$$A_1 \nabla \ldots \nabla A_n \ \widehat{=}\ |[\ \mathbf{var}\ u_1, \ldots, u_n;\ (R_{A_1} \star\ \ldots\ \star R_{A_n}) \diamondsuit (W_{A_1} \star\ \ldots\ \star W_{A_n})\ ]|$$

where the read / write parts are extracted as before, for each involved action.

Hence, a synchronous composition of actions is a two-phase action, where the simultaneous execution of the read parts is sequenced with the simultaneous execution of the corresponding write parts of the component actions. By the properties of two-phase actions we also have that

$$A_1 \nabla \ldots \nabla A_n = T_{A_1} \nabla \ldots \nabla T_{A_n}$$

### 3.3 Synchronous action systems

A *synchronous action system* is a system, whose actions are synchronously composed by the $\nabla$- operator. Such a system never terminates, since the two-phase representations of the component actions are always enabled. The simplest possible synchronous system is composed of a single two-phase action.

**Synchronous composition of action systems** Consider two synchronous action systems $\mathcal{A}$ and $\mathcal{B}$ of the form

| | |
|---|---|
| $\mathbf{sys}\mathcal{A}\,(g_A) ::$ | $\mathbf{sys}\mathcal{B}\,(g_B) ::$ |
| $\|[\ \mathbf{var}\ l_A;\ \mathbf{init} g_A, l_A := g_{A0}, l_{A0};$ | $\|[\ \mathbf{var}\ l_B;\ \mathbf{init} g_B, l_B := g_{B0}, l_{B0};$ |
| $\quad \mathbf{do}\ A_1 \nabla \ldots \nabla A_m \mathbf{od}$ | $\quad \mathbf{do}\ B_1 \nabla \ldots \nabla B_n \mathbf{od}$ |
| $]\|$ | $]\|$ |

where $l_A \cap l_B = \emptyset$, and the initializations of the global variables $g_A \cap g_B \neq \emptyset$ in the systems $\mathcal{A}$ and $\mathcal{B}$ are consistent with each other. The *synchronous composition* of $\mathcal{A}$ and $\mathcal{B}$, denoted $\mathcal{A} \nabla \mathcal{B}$, synchronizes the actions of the constituent action systems creating the system

$$\begin{array}{l}
\mathbf{sys}\mathcal{C}\,(g_A \cup g_B) :: \\
|[\ \mathbf{var}\ l_A \cup l_B;\ \mathbf{init} g_A \cup g_B, l_A \cup l_B := g_{A0} \cup g_{B0}, l_{A0} \cup l_{B0}; \\
\quad \mathbf{do}\ A_1 \nabla \ldots \nabla A_m \nabla B_1 \nabla \ldots \nabla B_n \mathbf{od} \\
\ ]|
\end{array}$$

# 4 System synchronization

In this section, we study the conversion of an asynchronous action system into the corresponding synchronous two-phase form.

## 4.1 Asynchronous vs. synchronous systems

From the point of view of the action systems formalism, the fundamental difference between a synchronous action system, defined in Section 3, and an asynchronous action system is that the former never terminates, since a two-phase action is always enabled, while the latter terminates when none of the actions is enabled. However, an asynchronous system is usually composed in parallel with an environment, such that the composition never terminates. Synchronous action systems, in turn, are individually non-terminating.

Let us consider the differences between the two system types from a more general and practical viewpoint.

An asynchronous system interacts with its environment via an asynchronous *communication (*or *handshake) channel*, which is composed of two boolean *handshake variables* [5], denoted *req* and *ack* ( *request, acknowledge*), and *data variables* whose values are transferred from the system to its environment, or vice versa. The handshake variable *req* is updated by the *master* system which is the active party of communication requesting tasks from the *server* system which is the passive party of communication. Respectively, the variable *ack* is updated by the server whenever it has completed a task requested by the master. Furthermore, the master only reads *ack*, and the server only reads *req*.

A synchronous system does not usually need a communication channel with two handshake variables. Especially, the variable *ack* is often unnecessary, since the operation of a synchronous device is paced by a regular clock signal, and the completion of a task does not have to be acknowledged explicitly.

Generally, a system is considered synchronous if:

- the clock input to every edge-sensitive part is a function of the primary clock input.
- all storage elements are edge-sensitive.

## 4.2 Asynchronous specification

The asynchronous approach can be considered a natural way to describe system behavior. Moreover, the action systems formalism is well featured for asynchronous modeling. Based on these facts, the initial specification of the target system is often asynchronous. For a large category of descriptions we are able to provide a method for obtaining synchronous representations. At the end of this section we provide a generic format for those action systems that can be refined into synchronous representations.

We start the description by establishing an action system $\mathcal{A}sync$ that would implement our system specification, together with another system, $\mathcal{E}nv$, that models the environment in which the system $\mathcal{A}sync$ works. As $\mathcal{E}nv$ provides the *req* signal to our system, we consider it as the *master* in our design. It is then important that $\mathcal{E}nv$ prevails over some system $\mathcal{A}sync$, i.e. whenever an action in $\mathcal{E}nv$ is enabled, it is executed, even though there might be enabled actions in $\mathcal{A}sync$, too. This can be modeled in the action systems formalism by the prioritized composition operator [21].

Thus, the initial specification in our case will be $\mathcal{E}nv$ // $\mathcal{A}sync$.

Let us consider for the beginning an asynchronous system that updates a list of output variables in the following way:

$$
\begin{array}{l}
\mathbf{sys}\mathcal{A}sync\,(req, ack : boolean; u) ::\\
|[\ \mathbf{var}\ local\ variables; \mathbf{init} initialization\ of\ variables;\\
\quad \mathbf{do}\ < (req \rightarrow ((g_1 \rightarrow A_1)\ [\!]\ \ldots\ [\!]\ (g_n \rightarrow A_n))); ack := true >\\
\quad\ [\!]\ < \neg req \rightarrow ack := false >\\
\quad \mathbf{od}\\
\ ]|
\end{array}
$$

Actions $A_i,\ i = 1 \ldots n$ are of the following form:

$$
\begin{array}{rcl}
A_i & = & < v := v'.Q_i >\\
 & = & < w_1 \ldots w_m := w'_1 \ldots w'_m.(Q^i_1 \wedge \ldots \wedge Q^i_m) >,\\
Q_i & = & Q^i_1 \wedge \ldots \wedge Q^i_m
\end{array}
$$

$v$ is a list of variables (local and global) updated by actions $A_i$. We require that the guards of the actions to be mutually exclusive and that $\bigvee_{j=1}^{n} g_{A_j} = true$.

Considering the above, $\mathcal{A}sync$ can be refined in a manner that leads us to the following system:

$$
\begin{array}{l}
\mathbf{sys}\mathcal{A}sync'\,(req, ack : boolean; global\ variables) ::\\
|[\ \mathbf{var}\ local\ variables; \mathbf{init} initialization\ of\ variables\\
\quad \mathbf{do}\ < (req \rightarrow ((g_1 \rightarrow A_1)\ [\!]\ \ldots\ [\!]\ (g_n \rightarrow A_n))\ [\!]\ (\neg req \rightarrow skip);\\
\quad ((\neg req \rightarrow ack := false)\ [\!]\ (req \rightarrow ack := true)) >\\
\quad \mathbf{od}\\
]|
\end{array}
$$

We have: $\mathcal{A}sync \sqsubseteq \mathcal{A}sync'$ (Appendix A). The reason for this transformation was to produce a separation between actions that update variables $v$ and $ack$.

## 4.3 Synchronous interpretation of program constructs

Below we show the synchronous interpretations of the program constructs of the above system $\mathcal{A}sync'$ (Section 4.2). We consider the choice and the sequential compositions. Proofs related to the transformations presented here are showed in Appendix B.

**Choice composition** The loop of the $\mathcal{A}sync'$ system is composed of an atomic sequence of actions, which we denote $S_1$ and $S_2$:

$$\begin{aligned} S_1 &= (req \to ((g_1 \to A_1) \; [\!] \; \ldots \; [\!] \; (g_n \to A_n))) \; [\!] \; (\neg req \to skip), \\ S_2 &= (\neg req \to ack := false) \; [\!] \; (req \to ack := true) \end{aligned}$$

Through successive refinement steps we reach the final non-deterministic assignment format:

$$\begin{aligned} S_1 &\leq S_1', \\ S_1' &= < (v := v'.Q > \\ &= < w_1 \ldots w_m := w_1' \ldots w_m'.((g_1 \Rightarrow Q_1) \wedge \ldots \wedge (g_n \Rightarrow Q_n)) > \\ &= < w_1 := w_1'.((g_1 \Rightarrow Q_1^1) \wedge \ldots \wedge (g_n \Rightarrow Q_1^n) > \star \ldots \\ &\quad \star < w_m := w_m'.((g_1 \Rightarrow Q_m^1) \wedge \ldots \wedge (g_n \Rightarrow Q_m^n) > \end{aligned}$$

Similarly,

$$\begin{aligned} S_2 &\leq S_2', \\ S_2' &= < ack := ack'.((req \Rightarrow ack' = true) \wedge (\neg req \Rightarrow ack' = false)) > \end{aligned}$$

**Sequential composition** The precedent paragraph showed that a choice construct could be refined, in a stepwise manner, into a non-deterministic assignment format.

The **do** $-$**od** loop of the system becomes now:

$$\mathbf{do} \; < S_1'; S_2' > \mathbf{od},$$

Actions $S_1'$ and $S_2'$ have no read/write conflicts. They can be then executed in a simultaneous manner. Thus, after a trivial proof, we have the refinement:

$$< S_1'; S_2' > \leq < S_1' \star S_2' >$$

Following the steps used in obtaining the two-phase representation of actions [19], we transform ( for simplicity reasons we will write only the **do** $-$**od** loops of the systems ):

$$\mathbf{do} \; < S_1' \star S_2' > \mathbf{od} \sqsubseteq \mathbf{do} \; S_1' \; \nabla \; S_2' \mathbf{od},$$

obtaining, thus, a synchronous system:

$\mathbf{sys}\mathcal{S}ync\,(req, ack : boolean; variables...) ::$
$|[$ **var** $variables...;$ **init**$initialization\ of\ state\ variables$
  **do** $S_1' \; \nabla \; S_2'$ **od**
$]|$

## 4.4 Generic asynchronous system

We studied above the synchronous transformation of different program constructs that were present in the generic asynchronous system $\mathcal{A}sync$. The refinement relations extend also to the $\mathcal{E}nv$ // $\mathcal{A}sync$ composition [21]:

$$(\mathcal{E}nv \; // \; \mathcal{A}sync) \sqsubseteq \ldots \sqsubseteq (\mathcal{E}nv \; // \; \mathcal{S}ync)$$

A more general format for the $\mathcal{A}sync$ system can be considered the following:

$$
\begin{array}{l}
\mathbf{sys}\mathcal{A}sync_G\,(req, ack : boolean; variables...) :: \\
|[\; \mathbf{var}\; variables...;\; \mathbf{init} initialization\ of\ state\ variables; \\
\quad \mathbf{do}\; < (req \rightarrow ((choice_1); \ldots; (choice_n)); ack := true > \\
\quad [\!]\; < \neg req \rightarrow ack := false > \\
\quad \mathbf{od} \\
]|
\end{array}
$$

The choice constructs individually comply with the requirements originally specified for system $\mathcal{A}sync$. For $\mathcal{A}sync_G$ to be a valid subject for transformation into a synchronous format, one choice construct must not refer to variables updated by any other previous choice constructs in the sequence:

$$
\begin{array}{rcl}
choice_j & \leq & < v_j := v'_j.Q_j > \\
 & = & < w^j_1 \ldots w^j_{m_j} := w^{j'}_1 \ldots w^{j'}_{m_j}.(Q^j_1 \wedge \ldots \wedge Q^j_{m_j}) >, \\
 & \text{where} & \forall (i, j = 1 \ldots n, i < j, k = 1 \ldots m_j, l = 1 \ldots m_i), \\
 & & Q^j_k \text{ does not mention } w^i_l.
\end{array}
$$

Thus the sequence can be transformed at its turn into a non-deterministic assignment (Appendix C).

The corresponding resulting synchronous system is then represented by

$$
\begin{array}{l}
\mathbf{sys}\mathcal{S}ync_G\,(req, ack : boolean; variables...) :: \\
|[\; \mathbf{var}\; variables; \mathbf{init} initialization\ of\ state\ variables \\
\quad \mathbf{do}\; T_{choice_1} \, \nabla \ldots \, \nabla T_{choice_n} \, \nabla \, T_{S'_2} \; \mathbf{od} \\
]|
\end{array}
$$

where action $S_2$ is defined as previously.

We have shown the details of translating a specific asynchronous description into its synchronous correspondent. The method can be resumed in completing the following steps:

1. The (possibly necessary) refinement of the initial asynchronous description in order to match the presented generic format.

2. The separation of the actions that update the $req/ack$ signals from the choice construct(s).

3. Refinement into the synchronous format.

The proposed method is to be used in section 6.

# 5 Implementation aspects

Circuit design is the transformation of a specification describing the required behavior into an equivalent one that is suitable for direct implementation in hardware. A synchronous machine is defined as consisting of a combinational network, with feedback from outputs to input via edge-sensitive devices [13]. In this section we will present the immediate hardware reflection of some basic constructs.

**Elementary cell.** As a particular implementation issue we will consider the assignment for a one-bit variable. In this case, the design would describe a common digital circuit: a storage element, the D-type flip-flop (Figure 1). We

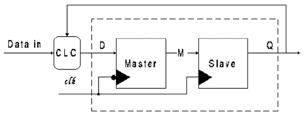


Figure 1: D Flip- flop

will refer to this circuit as the *elementary cell*.

The elementary cell represents the lowest level of a synchronous system description. A collection of elementary cells creates a register, or a *basic cell* [19]. For a bottom-up design, the elementary cell should be the starting point when describing a synchronous system.

**Choice.** The previous section studied the formal aspects of synchronous interpretation of the $[\!]$ -operator. On the hardware level choice can be represented with the help of a multiplexer placed in the combinational circuit of a register (Figure 2).

**Sequence of choice constructs.** The hardware representation of the sequence is a repetition of the schematic in Figure 2, for as many times as choice constructs in the sequence. The placement of choice blocks is dictated by the input/output variables of each construct.

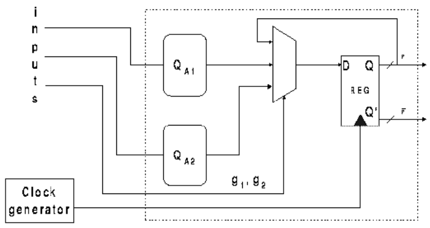


Figure 2: Choice as a multiplexer

# 6 Example: A 4-bit binary counter

In this section, we give an example of deriving a synchronous digital circuit from an asynchronous action system specification. For our case study we have selected a simple electronic circuit, a synchronous 4-bit up counter with parallel load.

The initial specification is: $\mathcal{E}nv \; // \; \mathcal{C}ount$, where

$$
\begin{array}{l}
\mathbf{sys}\mathcal{C}ount\,(req, ack, mode : boolean; din, dout : int) :: \\
|[\ \mathbf{init} req, ack := false, false; \\
\quad \mathbf{do}\ < req \rightarrow ((mode \rightarrow dout := dout'.(dout' = (dout + 1)\ mod\ 16\ ) \\
\qquad [\!]\ \neg mode \rightarrow dout := dout'.(dout' = din)); ack := true) > \\
\qquad [\!]\ < \neg req \rightarrow ack := false > \\
\quad \mathbf{od} \\
\ ]|
\end{array}
$$

$$
\begin{array}{l}
\mathbf{sys}\mathcal{E}nv\,(req, ack, mode : boolean;\ \ din, dout : int) :: \\
|[\ \mathbf{var}\ \ d : int;\ \mathbf{init} req, ack := false, false; \\
\quad \mathbf{do}\ < \neg ack \wedge \neg req \rightarrow\ \ d := dout;\ mode := mode'.(mode' \in boolean); \\
\qquad din := din'.(din' \in \{0, \ldots, 15\});\ req := true > \\
\qquad [\!]\ < ack \wedge req \rightarrow req := false > \\
\quad \mathbf{od} \\
\ ]|
\end{array}
$$

where $\mathcal{C}ount$ is the system we are considering, and $\mathcal{E}nv$ models the behavior of the environment. The following variables are used:

- $req$ ("request"), $ack$ ("acknowledge") are boolean handshake variables that constitute an asynchronous communication channel between thereactive components.
- $mode$ is a flag that sets the operating mode: load ($mode = false$) or count up ($mode = true$).
- $din$, $dout$ are integers representing input and output data of the counter.

It is easy to see that $\mathcal{E}nv$ is the active party (master) which activates the counter by sending a request signal $req = true$. Hence, $\mathcal{C}ount$ is the passive party (server) which responds by sending an acknowledgement signal $ack = true$ after completing the requested operation. Then the handshake variables are initialized to $false$, starting from $req$.

The system $\mathcal{C}ount$ maps directly to the generic structure presented in Section 4.2. We denote:

$$\begin{aligned} A_1 &= \; < mode \rightarrow dout := dout'.(dout' = (dout+1) \; mod \; 16 \;) >, \\ A_2 &= \; < \neg mode \rightarrow dout := dout'.(dout' = din) > \end{aligned}$$

As shown in Section 4.2, the initial specification with two separate actions can be straightforwardly refined into a single-action form. We have that $\mathcal{C}ount \sqsubseteq \mathcal{C}ount^1$ with

$$\begin{array}{l} \mathbf{sys} \mathcal{C}ount^1 \, (req, ack, mode : boolean; \; din, dout : int) :: \\ |[ \; \mathbf{init} \, req, ack := false, false; \\ \quad \mathbf{do} \; < ((req \rightarrow (A_1 \; [] \; A_2)) \; [] \; (\neg req \rightarrow skip); \\ \qquad (req \rightarrow ack := true \; [] \; \neg req \rightarrow ack := false) > \\ \quad \mathbf{od} \\ ]| \end{array}$$

The synchronization operation requires that the choices be first turned into non-deterministic assignment format.

We refine $\mathcal{C}ount^1 \sqsubseteq \mathcal{C}ount^2$, where

$$\begin{array}{l} \mathbf{sys} \mathcal{C}ount^2 \, (req, ack, mode : boolean; \; din, dout : int) :: \\ |[ \; \mathbf{init} \, req, ack := false, false; \\ \quad \mathbf{do} \; < dout := dout'.(req \wedge mode \Rightarrow dout' = (dout+1) \; mod \; 16) \; \wedge \\ \qquad (req \wedge \neg mode \Rightarrow dout' = din) \; \wedge (\neg req \Rightarrow dout' = dout); \\ \qquad ack := ack'.(req \Rightarrow ack' = true) \wedge (\neg req \Rightarrow ack' = false) > \\ \quad \mathbf{od} \\ ]| \end{array}$$

Because the two non-deterministic assignments in $\mathcal{C}ount^2$ do not have a read-write conflict, they can be executed simultaneously. Hence, we refine $\mathcal{C}ount^2 \sqsubseteq \mathcal{C}ount^3$, where

$$
\begin{array}{l}
\mathbf{sys}\mathcal{C}ount^3\,(req, ack, mode : boolean;\; din, dout : int) :: \\
|[\ \mathbf{init} req, ack := \mathit{false}, \mathit{false}; \\
\quad \mathbf{do}\ \ < dout := dout'.(req \wedge mode \Rightarrow dout' = (dout+1)\ mod\ 16)\ \wedge \\
\qquad (req \wedge \neg mode \Rightarrow dout' = din)\ \wedge \\
\qquad (\neg req \Rightarrow dout' = dout) > \\
\qquad \star < ack := ack'.(req \Rightarrow ack' = \mathit{true}) \wedge (\neg req \Rightarrow ack' = \mathit{false} > \\
\quad \mathbf{od} \\
\ ]|
\end{array}
$$

According to Section 4.3 a simultaneous composition can be transformed straightforwardly into a synchronous composition. Hence, we have the refinement $\mathcal{C}ount^3 \sqsubseteq \mathcal{C}ount^4$, where

$$
\begin{array}{l}
\mathbf{sys}\mathcal{C}ount^4\,(req, ack, mode : boolean;\; din, dout : int) :: \\
|[\ \mathbf{init} req, ack := \mathit{false}, \mathit{false}; \\
\quad \mathbf{do}\ \ < dout := dout'.(req \wedge mode \Rightarrow dout' = (dout+1)\ mod\ 16)\ \wedge \\
\qquad (req \wedge \neg mode \Rightarrow dout' = din)\ \wedge (\neg req \Rightarrow dout' = dout) > \\
\qquad \nabla < ack := ack'.(req \Rightarrow ack' = \mathit{true}) \wedge (\neg req \Rightarrow ack' = \mathit{false} > \\
\quad \mathbf{od} \\
\ ]|
\end{array}
$$

The system is now in a synchronous two-phase form and could be automatically transformed into VHDL code. However, we continue the formal derivation process further into bit-level description. This means that the integer-type data variables $din$, $dout$ and $d$ are replaced by the boolean vectors $din[3..0]$, $dout[3..0]$, and $d[3..0]$, where the index '0' corresponds to the least significant bit and the index '3' to the most significant bit, respectively. Naturally, the intermediate data variable hidden into $\nabla$ has to be transformed as well. We *hide* those variables that we change, and refine

$$
|[\ \mathcal{E}nv\ //\ \mathcal{C}ount^4\ ]| : req, ack, mode \quad \sqsubseteq \quad |[\ \mathcal{E}nv^1\ //\ \mathcal{C}ount^5\ ]| : req, ack, mode
$$

where the notation of the form $|[\ \mathcal{A}\ ]| : v$ means that the variables $v$ of the system $\mathcal{A}$ are observable, while the others are hidden. The actions of the composition $\mathcal{E}nv\ //\ \mathcal{C}ount^4$ are data-refined using an abstraction relation $R$ of the form

$$
\begin{array}{rcl}
R(dat, datv) & \widehat{=} & (\forall i \in \{0,\ldots,3\}.(datv[i] \Rightarrow x[i] = 1) \wedge (\neg datv[i] \Rightarrow x[i] = 0)) \wedge \\
 & & (dat = \sum_{j=0}^{3} x[j] * 2^j)
\end{array}
$$

where $dat$ represents an original integer-type variable and $datv[3..0]$ the corresponding boolean vector. The vector $x[3..0]$ is an auxiliary vector needed in the relation $R$ only. The resulting systems are:

$$\begin{array}{l}
\textbf{sys}\mathcal{C}ount^5\,(req, ack, mode : boolean;\;\; din[3..0], dout[3..0] : boolean) ::\\
|[\; \textbf{init} req, ack := \mathit{false}, \mathit{false};\\
\quad \textbf{do} \;\; < dout := dout'.(req \wedge mode \Rightarrow F) \;\; \wedge\\
\qquad (req \wedge \neg mode \Rightarrow (\forall i \in \{0,\ldots,3\}.dout'[i] = din[i])) \;\; \wedge\\
\qquad (\neg req \Rightarrow (\forall i \in \{0,\ldots,3\}.dout'[i] = dout[i])) >\\
\qquad \nabla < ack := ack'.(req \Rightarrow ack' = \mathit{true}) \wedge (\neg req \Rightarrow ack' = \mathit{false} >\\
\quad \textbf{od}\\
\;]|
\end{array}$$

$$\begin{array}{l}
\textbf{sys}\mathcal{E}nv^1\,(req, ack, mode : boolean;\;\; din[3..0], dout[3..0] : boolean) ::\\
|[\; \textbf{var}\;\, d[3..0] : boolean;\;\; \textbf{init} req, ack := false, false;\\
\quad \textbf{do} \;\; < \neg ack \wedge \neg req \rightarrow \;\; d := d'.(\forall i \in \{0,\ldots,3\}.d'[i] = dout[i])\\
\qquad mode := mode'.(mode' \in boolean);\\
\qquad din := din'.(\forall i \in \{0,\ldots,3\}.din[i]' \in boolean);\\
\qquad req := \mathit{true} >\\
\qquad \; [\!] \;\; < ack \wedge req \rightarrow req := \mathit{false} >\\
\quad \textbf{od}\\
\;]|
\end{array}$$

where $F \mathrel{\widehat{=}} F_0 \wedge F_1 \wedge F_2 \wedge F_3$, and

$$\begin{array}{llll}
F_0 & \widehat{=} & & (dout'[0] = \neg dout[0]),\\
F_1 & \widehat{=} & & (dout[0] \Rightarrow dout'[1] = \neg dout[1]) \wedge (\neg dout[0] \Rightarrow dout'[1] = dout[1]),\\
F_2 & \widehat{=} & & (dout[0] \wedge dout[1] \Rightarrow dout'[2] = \neg dout[2])\\
 & & \wedge & (\neg(dout[0] \wedge dout[1]) \Rightarrow dout'[2] = dout[2]),\\
F_3 & \widehat{=} & & (dout[0] \wedge dout[1] \wedge dout[2] \Rightarrow dout'[3] = \neg dout[3])\\
 & & \wedge & (\neg(dout[0] \wedge dout[1] \wedge dout[2]) \Rightarrow dout'[3] = dout[3])
\end{array}$$

By the definition of the simultaneous composition of actions the system $\mathcal{C}ount^5$ is equivalent to

$$\begin{array}{l}
\textbf{sys}\mathcal{C}ount^6\,(req, ack, mode : boolean;\;\; din[3..0], dout[3..0] : boolean) ::\\
|[\; \textbf{init} req, ack := \mathit{false}, \mathit{false};\\
\quad \textbf{do}\;\, (DOUT_0 \star DOUT_1 \star DOUT_2 \star DOUT_3) \nabla ACK \textbf{od}\\
\;]|
\end{array}$$

where, for $i = 0..3$:

$$
\begin{aligned}
DOUT_i \;\;&\widehat{=}\;\; < dout[i] := dout'[i].((req \wedge mode \Rightarrow F_i)\;\wedge \\
&\qquad\qquad (req \wedge \neg mode \Rightarrow dout'[i] = din[i])\;\wedge \\
&\qquad\qquad (\neg req \Rightarrow dout'[i] = dout[i])) >, \\
ACK \;\;&\widehat{=}\;\; < ack := ack'.(req \Rightarrow ack' = true) \wedge (\neg req \Rightarrow ack' = false >
\end{aligned}
$$

The last formal derivation step is to transform the simultaneous composition into the synchronous composition by refining $\mathcal{C}ount^6 \sqsubseteq \mathcal{C}ount^7$, where

$$
\begin{array}{l}
\textbf{sys}\mathcal{C}ount^7\,(req, ack, mode : boolean;\;\; din[3..0], dout[3..0] : boolean) :: \\
|[\ \textbf{init} req, ack := false, false; \\
\quad \textbf{do}\ DOUT_0 \nabla DOUT_1 \nabla DOUT_2 \nabla DOUT_3 \nabla ACK \textbf{od} \\
\ ]|
\end{array}
$$

The basic-cell representation of the final system $\mathcal{C}ount^7$ is shown in Figure 3.

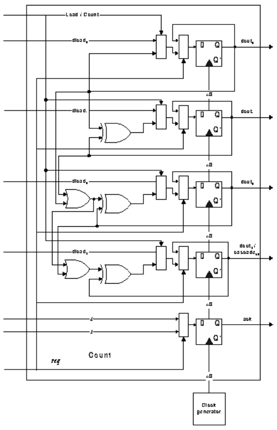


Figure 3: Basic-cell representation of the 4-bit counter

If we analyse the system $\mathcal{E}nv$, we notice that, with minor changes, it maps also on the format of the generic $\mathcal{A}sync$ system presented in Section 4. Applying the same method, we can translate it into a synchronous representation, too. This means that we have another starting point for our design, which would be $\mathcal{C}ount \;\nabla\; \mathcal{E}nv$, instead of $\mathcal{C}ount \;\; // \;\; \mathcal{E}nv$. This leads us to the fact that the environment is also a synchronous system. The idea of a *synchronous observer* was presented also in [8]. This also shows that not only the *server*-type systems can be subject to synchronization, but the *master*-type systems, too.

The presence of the variables *req*, *ack* is an aspect that we would rather avoid in a synchronous implementation. If we start from a description as presented above, $\mathcal{C}ount \nabla \mathcal{E}nv$, then the *req-ack* pair is not needed. Anyway, the *synchronous* environment is not a necessity for our description, as shown in this paper. The design flow we presented here also allows delaying the moment of deciding the actual architecture for the implementation (asynchronous or synchronous).

# 7 Conclusions

We have presented the basics of synchronous action systems design. Design is based on translating asynchronous action systems, a natural way of describing system behavior, into synchronous action systems, while preserving the correctness of the design. In [22] the authors defined a class of implementable relations. We defined a subset of asynchronous descriptions which can be transformed into their synchronous representation within the refinement calculus. We might call this subset as a synchronous implementable class of descriptions.

Most of the asynchronous data processing action systems can be refined to this generic format we take into account, as previous work proves it [15, 16, 17, 18].

We also provided immediate schematic representation for the constructs we analyzed. The example studied in Section 6 is an illustration of the presented design methodology. Either by custom design or automated means, the whole design process, from the abstract asynchronous specification to the concrete gate-level, is covered.

**Related work.** *Synchronized Transitions* is a UNITY-based formalism which has a synchronous composition operator [24]. The synchronous approach is somehow similar to ours, being a timeless one. The verification of the design implies a different analysis from the one used for asynchronous designs.

$\mu$FP [11, 10] also presents *untimed* descriptions for synchronous circuit design. It provides programs suitable for *direct implementation* in hardware.

A quite similar approach (to ours) towards clocked circuit descriptions is presented in [14]. The starting point is an asynchronous process that can be formally related to a synchronous hardware implementation. The analysis, however, uses distinct models for high and low level design.

In our framework we are able to use exactly the same rules for asynchronous or synchronous descriptions. Hence, the action systems formalism offers a powerful unifying basis for hardware design generally. It offers the possibility to work in a stepwise manner with systems that contain both synchronous and asynchronous components, leading to correctly derived descriptions and implementations. As showed in Section 5, our synchronous descriptions can be mapped to predefined hardware structures.

**Future work.** In continuing the study on synchronous design using the formal framework of action systems immediate steps will be taken towards the analysis of systolic arrays. As the programmable logic devices become common targets for hardware implementations, we will concentrate our efforts in order to improve the applicability of synchronous action systems for covering this specific area of design, too.


## Acknowledgements

Authors are grateful to Professor Kaisa Sere for fruitful discussions on the topic.


## References


[1] 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.

[2] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic 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*, pages 42–66. Springer–Verlag, 1990.

[3] R.-J. R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 1994*, volume 836 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[4] R.J.R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17-30, 1991.

[5] G. Birtwistle and A. Davis, editors. *Asynchronous Digital Circuit Design*. Springer, 1995.

[6] C. Brown, G. Hutton. *Categories, Allegories and Circuit Design*. IEEE Symposium on Logic in Computer Science, July 1994.

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

[8] N. Halbwachs, F. Lagnier and P. Raymond. *Synchronous observers and the verification of reactive systems.* Third Int. Conf. On Algebraic Methodology and Software Technology, AMAST'93, Workshops in Computing, Springer Verlag. Twente, June 1993.

[9] S. Hauck *Asynchronous Design Methodologies: An Overview* . Proceedings of the IEEE, Vol. 83, No. 1, pp. 69-93, January 1995.

[10] G. Jones, M.Sheeran. *Timeless Truths about Sequential Circuits* . Concurrent Computations: Algorithms, Architecture and Technology, ed. S.K. Tewksbury, B.W. Dickinson, S.C. Schwartz, Plenum Press, New York, 1988.

[11] G. Jones and M. Sheeran. *Relations and refinement in circuit design* . in 3rd Refinement Workshop, 1990, eds. Carroll Morgan and Jim Woodcock, Springer Workshops in Computing, 1991.

[12] M. Larsson. *An Engineering Approach to Formal Digital System Design.* The Computer Journal, Vol. 38, No. 1, 1995.

[13] P. Naish, P. Bishop. *Designing Asics.* Ellis Horwood Limited, Ellis Horwood Series In Electrical And Electronic Engineering, 1988.

[14] J. O'Leary, G. Brown, W. Luk. *Verified Compilation of Communicating Processes into Clocked Circuits.* Formal Aspects of Computing. The International Journal of Formal Methods, Volume 9, Numbers 5 & 6, 1997. Editors C. Jones, J. Cooke.

[15] J. Plosila, R.Rukšėnas, and K. Sere. *Delay-Insensitive Circuits and Action Systems.* TUCS Technical Report No 60, November 1996.

[16] J. Plosila, R.Rukšėnas, and K. Sere. *Action Systems Synthesis of DI Circuits.* Proc. of the International Refinement Workshop and Formal Methods Pacific 1998, September 1998, Canberra, Australia.

[17] J. Plosila and K. Sere. Action systems in pipelined processor design. In *Proc. of the 3rd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 156 – 166, 1997.

[18] J. Plosila and T. Seceleanu. *An Asynchronous Linear Predictive Analyzer.* TUCS Technical Report No 142, 1997.

[19] J. Plosila and T. Seceleanu. *Synchronous Action Systems* . TUCS Technical Report No 192, 1998.

[20] T. Seceleanu. *Implementation of Synchronous Action Systems* . TUCS Technical Report No 204, 1998.

[21] E. Sekerinski and K. Sere. *A Theory of Prioritizing Composition*. The Computer Journal, VOL. 39, No 8, pp. 701-712. The British Computer Society. Oxford University Press.

[22] R. Sharp and O. Rasmussen. *The T-Ruby Design System*. Computer Hardware Description Languages and Their Applications, CHDL '95. In S. Johnson, editor, CHDL'95, pages 587-596. IFIP WG10.5, 1995.

[23] E. P. Stabler, M. P. Nassif, R. J. Paragi. *Extending the Design Process with Formal Verification Technology*. VHDL International Users Forum, Santa Clara, 1995.

[24] J. Staunstrup. *A Formal Approach to Hardware Design*. Kluwer Academic Publishers 1994.

# Appendix: Proofs

## A. Refinement of the generic asynchronous system.

As the *main actions* refinement requirement of the above lemma, the relation we have to prove is:

$$
\begin{array}{rl}
& < (request \rightarrow (g_1 \rightarrow A_1) \; [\!] \; \ldots \; [\!] \; (g_n \rightarrow A_n)); acknowledge := true > \\
& [\!] \; < \neg request \rightarrow acknowledge := false > \\
\leq & < (request \rightarrow ((g_1 \rightarrow A_1) \; [\!] \; \ldots \; [\!] \; (g_n \rightarrow A_n))) \; [\!] \; \neg request \rightarrow skip; \\
& \quad (\neg request \rightarrow acknowledge := false \; [\!] \; request \rightarrow acknowledge := true) >,
\end{array}
$$

where $A_i =< v_i := v_i'.Q_i >, \; i = 1 \ldots n$. The refinement proof is done as follows:

- *Guards.* The resulting composition, as well as the initial one have the overall guard $true$, i.e. they are always enabled:

  $$
  \begin{array}{rcl}
  g_i & = & ((request \wedge (g_1 \vee \ldots \vee g_n)) \wedge true) \vee \neg request \\
  & = & ((request \wedge true) \wedge true) \vee \neg request \\
  & = & true,
  \end{array}
  $$

  and

  $$
  \begin{array}{rcl}
  g_r & = & ((request \wedge (g_1 \vee \ldots \vee g_n)) \vee \neg request) \wedge (request \vee \neg request) \\
  & = & ((request \wedge true) \vee \neg request) \wedge true \\
  & = & true,
  \end{array}
  $$

  where $g_i$ and $g_r$ are the guards of the initial and resulting constructs, respectively. The above means also that $g_r \Rightarrow g_i$.

- *Bodies.* We start by considering only one action in the *choice* composition. We will also consider the form of the involved action ( $g \rightarrow v := v'.Q$ ), and we want to show that:

  $$
  \begin{array}{l}
  \mathrm{wp}((request \rightarrow A; acknowledge := true) \\
  \quad [\!] \; (\neg request \rightarrow acknowledge := false), P) \Rightarrow \\
  \mathrm{wp}((request \rightarrow A \; [\!] \; \neg request \rightarrow skip); \\
  \quad ((request \rightarrow acknowledge := true) \\
  \quad [\!] \; (\neg request \rightarrow acknowledge := false)), P)
  \end{array}
  $$

  By applying the definitions for the weakest precondition of involved actions, we have:

  $$
  \mathrm{wp}((request \rightarrow v := v'.Q; acknowledge := true)
  $$

$$
\begin{aligned}
& \quad [\!]\ (\neg request \rightarrow acknowledge := false), P) \\
= & \quad \ldots \\
= & \quad (\forall v'.(request(\Rightarrow Q \Rightarrow P[true, v'/acknowledge, v])) \\
& \quad \wedge(\neg request \Rightarrow P[false/acknowledge])
\end{aligned}
$$

Similarly:

$$
\begin{aligned}
& \mathrm{wp}((request \rightarrow g \rightarrow v := v'.Q\ [\!]\ \neg request \rightarrow skip); \\
& \quad (request \rightarrow acknowledge := true) \\
& \quad\ [\!]\ (\neg request \rightarrow acknowledge := false), P) \\
= & \quad \ldots \\
= & \quad (\forall v'.(request(\Rightarrow Q \Rightarrow (request \Rightarrow P[true/acknowledge]) \\
& \quad \wedge(\neg request \Rightarrow P[false/acknowledge])) \\
& \quad \wedge(\neg request \Rightarrow (request \Rightarrow P[true/acknowledge]) \\
& \quad \wedge(\neg request \Rightarrow P[false/acknowledge]))
\end{aligned}
$$

Making the following notations ( for simplicity / readability reasons ):

$$
\begin{aligned}
a &= request, \\
b &= Q, \\
c &= P[true, v'/acknowledge, v], \\
d &= P[false/acknowledge], \\
e &= P[false, v'/acknowledge, v], \\
f &= P[true/acknowledge],
\end{aligned}
$$

the above relation becomes:

$$
\begin{aligned}
& (a \Rightarrow b \Rightarrow c) \wedge (\neg a \Rightarrow d) \\
\Rightarrow & \quad (a \Rightarrow b \Rightarrow (\neg a \Rightarrow e) \wedge (a \Rightarrow c)) \wedge (\neg a \Rightarrow (\neg a \Rightarrow d) \wedge (a \Rightarrow f)),
\end{aligned}
$$

which, through simple logical operations proves $true$.

The above show correctness of the presumed relation about main actions. Based on this and on the definition of the weakest precondition for *choice* composition, the relation is easily proven for any number of $A$ actions, and so, as we add no additional actions, the refinement $Async \sqsubseteq Async'$ is proven.

Thus, as we make no initializations and we add no auxiliary actions, the other requirements of *Lemma* 1 are also satisfied, and we have the relation:

$$
\mathcal{A}sync \sqsubseteq \mathcal{A}sync'
$$

## B. Synchronous interpretation of *choice* construct.

**Non-deterministic assignment format for the choice composition** First we concentrate on showing that the choice composition of actions of the type $< g_j \rightarrow v := v'.Q_j >$ can be refined to an assignment format.

For simplicity reasons we don't analyze the case when the output variable $v$ is actually a list of variables. The proofs are absolutely similar is that case.

Let us first considering a two-action construct only:

$$\begin{array}{l} < (g_1 \rightarrow v := v'.Q_1) \; [\!] \; (g_2 \rightarrow v := v'.Q_2) >, \text{ where:} \\ g_1 \otimes g_2 = true, \end{array}$$

We refine the composition as follows:

$$\begin{array}{ll} & < (g_1 \rightarrow v := v'.Q_1) \; [\!] \; (g_2 \rightarrow v := v'.Q_2) > \\ \leq & < v := v'.((g_1 \Rightarrow Q_1) \wedge (g_2 \Rightarrow Q_2)) > \\ & =< v := v'.Q >, \\ & Q = (g_1 \Rightarrow Q_1) \wedge (g_2 \Rightarrow Q_2) \end{array}$$

The proof is made in two steps:

1. *Guards.* It is obvious that the guards of the both actions are the same.
2. *Bodies.* Applying the definitions of the weakest preconditions for the choice composition and for the guarded assignment we have:

$$\begin{array}{lll} & \mathrm{wp}((g_1 \rightarrow v := v'.Q_1) \; [\!] \; (g_2 \rightarrow v := v'.Q_2), P) & \\ \Rightarrow & \mathrm{wp}(v := v'.((g_1 \Rightarrow Q_1) \wedge (g_2 \Rightarrow Q_2)), P) & \Leftrightarrow \\ & (\forall v'.(g_1 \Rightarrow Q_1 \Rightarrow P[v'/v])) \wedge (\forall v'.(g_2 \Rightarrow Q_2 \Rightarrow P[v'/v])) & \\ \Rightarrow & \forall v'.(((g_1 \Rightarrow Q_1) \wedge (g_2 \Rightarrow Q_2)) \Rightarrow P[v'/v]) & \Rightarrow \\ & (\forall v'.(g_1 \Rightarrow Q_1 \Rightarrow P[v'/v])) \wedge (\forall v'.(g_2 \Rightarrow Q_2 \Rightarrow P[v'/v])) & \\ \Rightarrow & (\forall v'.((g_1 \Rightarrow Q_1) \Rightarrow P[v'/v])) \wedge (\forall v'.((g_2 \Rightarrow Q_2) \Rightarrow P[v'/v])) & \Leftrightarrow \\ & (\forall v'.(g_1 \Rightarrow Q_1 \Rightarrow P[v'/v])) \wedge (\forall v'.(g_2 \Rightarrow Q_2 \Rightarrow P[v'/v])) & \\ \Rightarrow & (\forall v'.((g_1 \Rightarrow Q_1) \Rightarrow P[v'/v])) \wedge (\forall v'.((g_2 \Rightarrow Q_2) \Rightarrow P[v'/v])) & \Leftrightarrow \\ & true, & \end{array}$$

Generalizing,

$$< (g_1 \rightarrow A_1) \; [\!] \; \ldots \; [\!] \; (g_n \rightarrow A_n) > \leq < v := v'.Q >$$

The final conclusion is that the original choice construct, with the known properties regarding the guards of the involved actions, can be refined into a multiple non-deterministic format. When $v = \{w_1, \ldots, w_m\}$, we have:

$$\begin{array}{rcl} A & \widehat{=} & < v := v'.Q > \\ & = & < w_1 \ldots w_m := w'_1 \ldots w'_m.(Q'_1 \wedge \ldots \wedge Q'_m) > \\ & = & < w_1 := w'_1.Q'_1 > \star \ldots \star < w_m := w'_m.Q'_m >, \\ Q'_j & = & (g_1 \Rightarrow Q^j_1) \wedge \ldots \wedge (g_n \Rightarrow Q^j_n), j = 1 \ldots m. \end{array}$$

Based on the monotonicity of the choice composition w.r.t. refinement, the initial construct is refined into:

$$\begin{array}{rcl} S_1 & \leq & S_1', \\ S_1 & = & <(req \rightarrow (g_1 \rightarrow A_1) \,[\!]\, \ldots \,[\!]\, (g_n \rightarrow A_n)) \,[\!]\, (\neg req \rightarrow skip)>, \\ S_1' & = & <(req \rightarrow A) \,[\!]\, (\neg req \rightarrow skip)> \end{array}$$

Interpreting the *skip* action as preserving the old values of all the output variables involved in the composition, we rewrite the above action as:

$$S_1' = <(req \rightarrow A) \,[\!]\, (\neg req \rightarrow V)>,$$

and again, analyzing the logical expressions for the weakest preconditions, we end up proving that

$$\begin{array}{rcl} & & S_1' \leq S_1'', \\ V & = & (v' = v), \\ S_1'' & = & < v := v'.((req \Rightarrow Q) \wedge (\neg req \Rightarrow V) \end{array}$$

Considering $v$ a list of variables, as before, we have:

$$\begin{array}{rcl} v & = & \{w_1, \ldots, w_m\} \\ V & = & (w_1' = w_1) \wedge \ldots \wedge (w_m' = w_m) \\ & = & V_1 \wedge \ldots \wedge V_m \\ S_1'' & = & < v := v'.Q'' > \\ & = & < w_1, \ldots, w_m := w_1', \ldots, w_m'.((req \Rightarrow Q_1' \wedge \ldots \wedge Q_m') > \\ & & \wedge(\neg req \Rightarrow V_1 \wedge \ldots \wedge V_m)) > \\ & = & < w_1, \ldots, w_m := w_1', \ldots, w_m'.(Q_1'' \wedge \ldots \wedge Q_m'') > \\ & = & < w_1 := w_1'.Q_1'' > \star \ldots \star < w_m := w_m'.Q_m'' >, \\ Q'' & = & (req \Rightarrow Q_1' \wedge \ldots \wedge Q_m'), \\ Q_j'' & = & (req \Rightarrow Q_j') \wedge (\neg req \Rightarrow V_j) \end{array}$$

A similar treatment applied to action $S_2$ leads us to the refined version:

$$\begin{array}{rcl} S_2 & \leq & S_2' \\ S_2' & = & < ack := ack'.((req \Rightarrow ack' = true) \wedge (\neg req \Rightarrow ack' = false)) > \\ & = & < ack := ack'.Q_{ack} > \end{array}$$

The transformation of the sequence $< S_1''; S_2 >$ into $< S_1 \star S_2 >$ is proven as follows:

- *Guards.* The guards of both compositions are $true$, so this requirement is satisfied.

- *Bodies.*

$$
\begin{array}{lll}
 & \mathrm{wp}(S_1''; S_2', P) & \\
\Rightarrow & \mathrm{wp}(S_1'' \star S_2', P) & \Leftrightarrow \\
 & (\forall v'.(Q'' \Rightarrow (\forall ack'.Q_{ack} \Rightarrow P[v', ack'/v, ack]))) & \\
\Rightarrow & \forall v', ack'.(Q'' \wedge Q_{ack} \Rightarrow P[v', ack'/v, ack]) & \Rightarrow \\
 & \{ack' \text{ is not mentioned in } \}Q'' & \\
 & (\forall v', ack'.(Q'' \Rightarrow (Q_{ack} \Rightarrow P[v', ack'/v, ack]))) & \\
\Rightarrow & (\forall v', ack'.(Q'' \wedge Q_{ack} \Rightarrow P[v', ack'/v, ack])) & \Rightarrow \\
 & \{ \text{ laws of implication } \} & \\
 & (\forall v', ack'.(Q'' \wedge Q_{ack} \Rightarrow P[v', ack'/v, ack])) & \\
\Rightarrow & (\forall v', ack'.(Q'' \wedge Q_{ack} \Rightarrow P[v', ack'/v, ack])) & \Leftrightarrow \\
 & true &
\end{array}
$$

As no auxiliary actions are added, the requirements of trace refinement Lemma are thus accomplished, and we have the relation:

$$\mathbf{do}\ \ < S_1''; S_2' > \mathbf{od} \sqsubseteq \mathbf{do}\ \ < S_1'' \star S_2' > \mathbf{od}$$

We can consider now to transform the action in the loop of the above system into its two-phase representation, as shown in [19]:

$$
\begin{array}{rcl}
T_A & = & |[\ \mathbf{var}\ u, u_{ack};\ < u, u_{ack} := u', u'_{ack}.((Q'' \wedge Q_{ack}) > \\
 & & \qquad \Diamond\ < v, ack := u, u_{ack} >\ ]| \\
 & = & |[\ \mathbf{var}\ u, u_{ack};\ < R \star R_{ack} >\ \Diamond\ < W \star W_{ack} >\ ]| \\
R & = & < u := u'.Q'' >, \\
R_{ack} & = & < u_{ack} := ack'.Q_{ack} >, \\
W & = & < v := u >, \\
W_{ack} & = & < ack := u_{ack} >
\end{array}
$$

Supplementary, considering the possible list format for $v$, and the corresponding representation of $Q''$:

$$
\begin{array}{rcl}
T_A & = & |[\ \mathbf{var}\ u_1, \ldots, u_m, u_{ack}; < R_1 \star \ldots \star R_m \star R_{ack} > \\
 & & \quad \Diamond\ < W_1 \star \ldots \star W_m \star W_{ack} >\ ]| \\
 & = & |[\ \mathbf{var}\ u_1, \ldots, u_m; < R_1 \Diamond W_1 >\ \nabla \ldots \nabla\ < R_m \Diamond W_m > \\
 & & \quad \nabla\ < R_{ack} \Diamond W_{ack} >\ ]|,\ \text{where:} \\
R_i & = & < u_i = u_i'.Q_i'' >, \\
W_i & = & < w_i = u_i >,\ i = 1 \ldots m, \\
R_{ack} & = & < u_{ack} = u'_{ack}.Q_{ack} >, \\
W_{ack} & = & < ack = u_{ack} >
\end{array}
$$

Thus we showed the detailed process of translating the original system into its synchronous correspondent:

$$
\begin{array}{l}
\mathbf{sys} Sys\,(req, ack : boolean; variables...) :: \\
|[\ \mathbf{var}\ b, variables...; \mathbf{init} initialization\ of\ state\ variables \\
\quad \mathbf{do}\ T_{w_1}\ \nabla\ \ldots\ \nabla\ T_{w_m}\ \nabla\ T_{S'_2}\ \mathbf{od} \\
\ ]|
\end{array}
$$

( where $T_{w_j} = |[\ \mathbf{var}\ u_j;\ < R_j \diamondsuit W_j >\ ]|,\ j = 1 \ldots m.$ )

## C. The sequence of choice constructs.

It has already been shown ( Appendix B ) that, if the requirements considering the actions in tone choice composition and their guards are respected, then a choice construct can be represented as a (multiple) non-deterministic assignment. What is left to prove, then, is the fact that a sequence of non-deterministic assignments can be represented as a single multiple assignment (provided that the requirement relative to the appearance of updated variables in one of the following assignments in the sequence). In order to prove this, we will analyze a sequence of two non-deterministic assignments.

Consider the composition:

$$< v_1 := v'_1.Q_1; v_2 := v'_2.Q_2 >,$$

where $Q_2$ does not mention $v_1$. In this situation we can show that

$$
\begin{array}{lrcl}
\text{denoting:} & A & \widehat{=} & < v_1 := v'_1.Q_1 >, \\
 & B & \widehat{=} & < v_2 := v'_2.Q_2 >, \\
 & A \star B & \widehat{=} & < v_1, v_2 := v'_1, v'_2.Q_1 \wedge Q_2 > \\
\text{we have:} & A; B & \leq & A \star B,
\end{array}
$$

The proof goes on in two steps, as follows:

- *Guards.* The guards of both actions are $true$.
- *Bodies.*

$$
\begin{array}{lll}
 & \mathrm{wp}(A; B, P) & \\
\Rightarrow & \mathrm{wp}(A \star B, P) & \Leftrightarrow \\
 & (\forall v'_1.(Q_1 \Rightarrow (\forall v'_2.Q_2 \Rightarrow P[v'_1, v'_2/v_1, v_2]))) & \\
\Rightarrow & \forall v'_1, v'_2.(Q_1 \wedge Q_2 \Rightarrow P[v'_1, v'_2/v_1, v_2]) & \Rightarrow \\
 & \{v'_1 \text{ is not mentioned in } Q_2\} & \\
 & (\forall v'_1, v'_2.(Q_1 \Rightarrow (Q_2 \Rightarrow P[v'_1, v'_2/v_1, v_2]))) & \\
\Rightarrow & (\forall v'_1, v'_2.(Q_1 \wedge Q_2 \Rightarrow P[v'_1, v'_2/v_1, v_2])) & \Rightarrow \\
 & \{ \text{ laws of implication } \} & \\
 & (\forall v'_1, v'_2.(Q_1 \wedge Q_2 \Rightarrow P[v'_1, v'_2/v_1, v_2])) & \\
\Rightarrow & (\forall v'_1, v'_2.(Q_1 \wedge Q_2 \Rightarrow P[v'_1, v'_2/v_1, v_2])) & \Leftrightarrow \\
 & true &
\end{array}
$$