Delay-Insensitive Circuits and Action Systems

Juha Plosila

University of Turku, Department of Applied Physics
Laboratory of Electronics and Information Technology
FIN-20014 Turku, Finland

Rimvydas Rukšėnas

Åbo Akademi University, Department of Computer Science
FIN-20520 Turku, Finland

Kaisa Sere

Åbo Akademi University, Department of Computer Science
FIN-20520 Turku, Finland

Turku Centre for Computer Science
TUCS Technical Report No 60
November 1996

ISBN 951-650-883-9
ISSN 1239-1891

Abstract

The action systems formalism has been successfully used as a formal framework for reasoning about concurrent behaviour. Circuits, both synchronous and asynchronous, exhibit concurrent behaviour. In this paper, we combine the design of asynchronous circuits with action systems. We show how (quasi) delay-insensitive circuits can be modelled as action systems. This is done by defining classes of action systems which possess properties allowing to implement them as (quasi) delay-insensitive circuits. In this way the considerable body of theory developed for action systems can be utilized in VLSI design. Circuits are usually designed in a modular fashion. Hence, we pay special attention to modularization facilities in action systems.

Keywords: action systems, delay-insensitive circuits, design, modularization, circuit element

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Asynchronous delay-insensitive (DI) circuits are devices, whose timing and correctness does not depend on delays in circuit elements, i.e., gates, and wires [5]. This property relies on the strict *acknowledgment requirement*: every transition on each boolean circuit variable must be acknowledged by a transition on some other circuit variable. In practise, however, the set of completely delay-insensitive circuits is extremely limited, and thus we compromise somewhat over the acknowledgment requirement by assuming that the *forks*, i.e., the branched wires, are *isochronic*. This means that the differences between the delays in such branches are considered negligible compared to the delays in the gates whose inputs are connected to the branches in question. Consequently, a transition on an entire isochronic fork is considered acknowledged, when a transition on a single branch has been acknowledged. Delay-insensitive circuits under the isochronicity assumption are called *quasi delay-insensitive (QDI)* circuits [7].

Asynchronous circuits are well-suited for formal specification and design which is essential for guaranteeing the correct behaviour of complex circuits. Moreover, Martin [12, 13] has showed that formal techniques familiar from parallel programming are usable for circuit design as well. Action systems have shown their strength in the construction of parallel and distributed systems [1, 3]. Their suitability in the initial steps of circuit design has also been demonstrated [2, 16].

In this paper, we show how action systems are used to model delay-insensitive circuits. We define a class of action systems, *quasi delay-insensitive action systems*, by restricting the way the actions in a system interact with each other. A computation of such an action system models a computation in a circuit. We also identify a subclass of quasi delay-insensitive systems, *delay-insensitive action systems*, which puts even stronger restrictions for the interaction between actions. Furthermore, we relate (quasi) delay-insensitive action systems to the digital gate-level circuit constructs. Quasi delay-insensitivity of an action system includes a stability requirement for the actions. This basically enables hazard-free construction of a circuit using standard components like AND-gates, OR-gates, inverters and wires as well as C-elements.

An important aspect of circuit design is modularity. We show how this is supported by the action systems framework by defining the concept of a (quasi) delay-insensitive *component* that interacts with other components following a handshaking protocol. We show that the parallel composition of two (quasi) delay-insensitive components will form a (quasi) delay-insensitive component provided that the interactions between the components conform to certain compatibility conditions.

In general when working with action systems, we do not put any restrictions on how the different action systems executing in parallel access variables. When considering delay-insensitive components, we assume that each shared variable is written by only one of them. It can, however, be read by several components in parallel. Moreover, our action systems are assumed to be weakly fair towards actions. This is a realistic assumption since the actions model gates and gates in practice are inherently fair.

Overview of the paper We proceed as follows. In section 2 we briefly introduce action systems. In section 3 a subclass of the action systems which corresponds to digital circuits is defined. In section 4 we give conditions under which an action system can be implemented as a (quasi) delay-insensitive circuit. In section 5, we describe the modularization mechanisms available for action systems and how these support the modularization idea in the design of digital circuits. In section 6, we show how action systems are mapped to the gate-level constructs of a circuit. We end in section 7 with some concluding remarks and references to related work.

2 Action systems

The action systems formalism is based on Dijkstra's language of *guarded commands* [8]. This language includes assignment, sequential composition, conditional choice, and iteration.

Actions An *action* is a guarded command, i.e. statement of the form

$$g \rightarrow S$$

where g is a boolean condition, the *guard*, and S is a program statement, the *body*. An action is said to be *enabled* when its guard is true. The guard of an action A often is written as gA and the body of A as sA. Furthermore, vA denotes the set of all variables accessed by the action A.

The program statements are defined using *weakest precondition* [8] predicate transformers in the usual manner. For an action A, the weakest precondition is defined as follows:

$$\mathrm{wp}(A, q) \quad \hat{=} \quad gA \Rightarrow \mathrm{wp}(sA, q).$$

The total correctness of the action A wrt *precondition* p and *postcondition* q is then defined as

$$p \ \langle A \rangle \ q \quad \hat{=} \quad p \Rightarrow \mathrm{wp}(A, q).$$

In the above definitions, p and q are predicates. Total correctness means that q holds after the execution of an action A in the initial state satisfying p.

Properties of actions The way in which actions can affect each other is captured by the following definitions:

$$\begin{array}{lcl} A \textit{ can enable } B & \hat{=} & \neg(\neg gB \ \langle A \rangle \ \neg gB) \\ A \textit{ cannot disable } B & \hat{=} & gB \ \langle A \rangle \ gB \\ A \textit{ excludes } B & \hat{=} & gA \Rightarrow \neg gB. \end{array}$$

Furthermore, an action that does not change the state is called a *stuttering action*. Namely, an action A is a stuttering action, if $(v = V)\ \langle A \rangle\ (v = V)$ holds for any V, where v represents the state variables and V values of these. An action A is called a *change action*, if it is not stuttering.

Action systems An *action system* is a statement of the form

$$\mathcal{A} :: \quad |[\ \mathbf{var}\ v.I;\ \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_m\ \mathbf{od}\ ]| : z$$

The identifiers v are the variables declared in $\mathcal{A}$, whereas z are the variables referenced, but not declared, in $\mathcal{A}$. Variables in v are called *exported* and superscripted by an asterisk *, if they are accessible outside $\mathcal{A}$. Otherwise, they are *local* variables. The system $\mathcal{A}$ is called a *closed* action system, if all variables in $\mathcal{A}$ are local. Otherwise it is an *open (reactive)* system. The variables z are called *imported* variables. The *global* variables of $\mathcal{A}$ consist of exported and imported ones. Within the loop, $A_1, \ldots, A_m$ are the actions of $\mathcal{A}$.

The behaviour of the action system $\mathcal{A}$ is the following. The variables v are initialized by the action I. Then, repeatedly, an enabled action from $A_1, \ldots, A_m$ is nondeterministically selected and executed. If two actions are independent, i.e., they do not have any variables in common, they can be executed in parallel [3]. Their parallel execution is then equivalent to executing the actions one after the other, in either order.

Action systems do not have any fairness constraints associated with them. In this paper we, however, consider action systems that treat every enabled action in a weakly fair manner. Weak fairness means that every enabled action is eventually selected for execution. Fairness in action systems is treated in detail elsewhere [1].

Hiding Let x be an exported variable of the action system

$$\mathcal{A} :: \quad |[\ \mathbf{var}\ x^*, v.I;\ \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_m\ \mathbf{od}\ ]| : z$$

The variable x is *hidden* (made local) in the action system $|[\ \mathbf{var}\ x \bullet \mathcal{A}\ ]|$, defined as

$$|[\ \mathbf{var}\ x \bullet \mathcal{A}\ ]| \quad \widehat{=} \quad |[\ \mathbf{var}\ x, v.I;\ \mathbf{do}\ A_1 \ [\!]\ \ldots\ [\!]\ A_m\ \mathbf{od}\ ]| : z\ .$$

Composition of action systems Consider two action systems,

$$\begin{array}{l} \mathcal{A} ::\ \ |[\ \mathbf{var}\ v.I;\ \mathbf{do}\ A_1 \ [\!]\ \ldots\ [\!]\ A_m\ \mathbf{od}\ ]| : x \\ \mathcal{B} ::\ \ |[\ \mathbf{var}\ w.J;\ \mathbf{do}\ B_1 \ [\!]\ \ldots\ [\!]\ B_n\ \mathbf{od}\ ]| : y \end{array}$$

where $v \cap w = \emptyset$. We define the *parallel composition* of $\mathcal{A}$ and $\mathcal{B}$, written $\mathcal{A} \parallel \mathcal{B}$, to be the following action system $\mathcal{C}$:

$$\begin{array}{ll} \mathcal{C} ::\ \ |[& \mathbf{var}\ v.I; w.J; \\ & \mathbf{do}\ A_1 \ [\!]\ \ldots\ [\!]\ A_m\ [\!]\ B_1\ [\!]\ \ldots\ [\!]\ B_k\ \mathbf{od} \\ \ \ ]|: (x \cup y) \setminus (v \cup w) & \end{array}$$

The behaviour of a parallel composition of action systems is dependent on how the individual action systems interact with each other via the shared global variables. 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 elsewhere [3].

3 Elementary action systems

In this section, we identify a subclass of the general action systems. An action system from the subclass describes in detail the behaviour of a digital circuit. We call this subclass the elementary action systems.

An *elementary action* E_v is a guarded command of the form

$$b_1 \to v := V,$$

where all accessed variables are boolean, $v = v_1, \ldots, v_n$, $V = V_1, \ldots, V_n$ and either $\forall 1 \leq i \leq n.(V_i = \mathit{true})$ or $\forall 1 \leq i \leq n.(V_i = \mathit{false})$ holds. The elementary action

$$b_2 \to v := \neg V,$$

where $\neg V$ is defined to be $\neg V_1, \ldots, \neg V_n$, is denoted $\overline{E}_v$ and called the *complement* of E_v. Vice versa, E_v is the complement of $\overline{E}_v$. An action and its complement are called a *complementary pair*, written $(E_v, \overline{E}_v)$. If $n = 1$ the complementary pair $(E_v, \overline{E}_v)$ is called *unbranched*, otherwise it is *branched*.

For $(E_v, \overline{E}_v)$ of the form

$$(b_1 \to v_1, \ldots, v_n := U, \ldots, U, \quad b_2 \to v_1, \ldots, v_n := \neg U, \ldots, \neg U)$$

where $U \in \{true, false\}$, the complementary pair

$$(b_1 \to w_1, \dots, w_k := U, \dots, U, \quad b_2 \to w_1, \dots, w_k := \neg U, \dots, \neg U)$$

is a *duplicate* of $(E_v, \overline{E}_v)$, and vice versa. Furthermore, a complementary pair of the form

$$((u = U) \to v_1, \dots, v_n := U, \dots, U, \quad (u = \neg U) \to v_1, \dots, v_n := \neg U, \dots, \neg U)$$

is said to be *undelayed*. Any other complementary pair is *delayed*.

Note We use the notation $v \uparrow$ ($v \downarrow$) to represent an assignment of the value *true* (*false*) to all variables in the list v. Furthermore, we usually assume that the action E_v assigns the variables in v a value which differs from the initial one, and, consequently, its complement $\overline{E}_v$ restores the initial state of the variables. When there is no need to refer to a specific list of variables v, an elementary action and its complement are written as A and $\overline{A}$, respectively.

Definition 1 *Consider a weakly fair action system $\mathcal{E}$ of the form:*

$$\mathcal{E} :: \quad |[\ \mathbf{var}\ v.I;\ \mathbf{do}\ E_{v_1} \parallel \overline{E}_{v_1} \parallel \cdots \parallel E_{v_m} \parallel \overline{E}_{v_m}\ \mathbf{od}\ ]| : z$$

where the imported variables z are read-only, and v_i, $i = 1, \dots, m$, denote disjoint sets of boolean variables initialized by the action I. The system $\mathcal{E}$ is called an elementary action system.

Example 1 The system $\mathcal{E}x_1$ below is a closed elementary action system. We can notice that the pair $(E_{d,e}, \overline{E}_{d,e})$ is branched while the others are unbranched. Furthermore, the two duplicates $(E_{d,e}, \overline{E}_{d,e})$ and $(E_f, \overline{E}_f)$ as well as the pair $(E_a, \overline{E}_a)$ are delayed while $(E_b, \overline{E}_c)$ and $(E_c, \overline{E}_c)$ are undelayed complementary pairs.

$$
\begin{array}{lll}
\mathcal{E}x_1 :: \quad |[& \mathbf{var}\ a, b, c, d, e, f.\ a\downarrow, b\downarrow, c\downarrow, d\downarrow, e\downarrow, f\downarrow; & \\
 & \mathbf{do}\ d \wedge e \wedge f \to a\uparrow & [E_a] \\
 & \parallel\ a \to b\uparrow & [E_b] \\
 & \parallel\ b \to c\uparrow & [E_c] \\
 & \parallel\ c \to d\downarrow, e\downarrow & [\overline{E}_{d,e}] \\
 & \parallel\ c \to f\downarrow & [\overline{E}_f] \\
 & \parallel\ \neg d \wedge \neg e \wedge \neg f \to a\downarrow & [\overline{E}_a] \\
 & \parallel\ \neg a \to b\downarrow & [\overline{E}_b] \\
 & \parallel\ \neg b \to c\downarrow & [\overline{E}_c] \\
 & \parallel\ \neg c \to d\uparrow, e\uparrow & [E_{d,e}] \\
 & \parallel\ \neg c \to f\uparrow & [E_f] \\
 & \mathbf{od}\ ]| &
\end{array}
$$

Note The name of an action is given in the brackets to the right of it for ease of reference.

Immediate successors The set of all actions in the elementary action system $\mathcal{E}$ is denoted by $\mathit{Action}(\mathcal{E})$.

Definition 2 *Assume,* $A, B \in \mathit{Action}(\mathcal{E})$ *and* A *can enable* B*. Then the action* B *is called an* immediate successor *of* A*. The set of immediate successors of action* A *is written as* $\mathit{Succ}_1(A)$*.*

Example 2 Consider the elementary action system from Example 1:

$$\mathit{Succ}_1(E_a) = \{E_b\} \qquad\qquad \mathit{Succ}_1(\overline{E}_a) = \{\overline{E}_b\}$$

Computation paths Assume, $A, B \in \mathit{Action}(\mathcal{E})$. A *computation path* from action A to action B is a sequence of immediate successors which leads from A to B. The loops in those sequences are omitted. Formally, for $i \geq 1$, we consider the following sets of actions whose execution follow or precede A:

$$\begin{aligned} \mathit{Succ}_{i+1}(A) &\;\widehat{=}\; \mathit{Succ}_i(A) \cup \{B \in \mathit{Action}(\mathcal{E}) \mid \exists C \in \mathit{Succ}_i(A). B \in \mathit{Succ}_1(C)\} \\ \mathit{Pre}_i(A) &\;\widehat{=}\; \{B \mid A \in \mathit{Succ}_i(B)\} \end{aligned}$$

The set of actions from all computation paths between A and B, denoted $\mathit{Path}(A, B)$, is defined as follows:

$$\begin{aligned} \mathit{Path}(A, B) \;\widehat{=}\; & \{C \in \mathit{Action}(\mathcal{E}) \mid \exists i, j > 0. \\ & \quad A \in \mathit{Pre}_i(C) \wedge B \in \mathit{Succ}_j(C) \wedge \mathit{Pre}_i(C) \cap \mathit{Succ}_j(C) = \emptyset\} \end{aligned}$$

The set $\mathit{Path}(A, B)$ is empty, when B is an immediate successor of A.

Example 3 Consider the elementary action system from Example 1:

$$\mathit{Path}(E_a, \overline{E}_a) = \{E_b, E_c, \overline{E}_{d,e}, \overline{E}_f\} \qquad \mathit{Path}(\overline{E}_a, E_a) = \{\overline{E}_b, \overline{E}_c, E_{d,e}, E_f\}$$

Real and pseudo successors A *real successor* of an elementary action $A \in \mathit{Action}(\mathcal{E})$ is an action $B \in \mathit{Path}(A, \overline{A})$ whose execution follows A in such a way, that there is a real gate-delay between the completions of the actions A and B at the circuit level. However, not all successors of the action A are necessarily real ones in the physical sense. For example, the action $B \in \mathit{Succ}_1(A)$ is not considered a real successor when the corresponding complementary pair $(B, \overline{B})$ is undelayed. The reason for this is that such a pair is implemented physically as a wire which, provided it is short enough, has a negligible propagation delay compared to delays in transistor-based gates. Such an action B is called *pseudo successor* of A. Formally, we have the following definitions:

DEFINITION 3 *Assume* $A, B \in Action(\mathcal{E})$. *Then* B *is an* immediate pseudo successor *of* A, *if* $B \in Succ_1(A)$ *and* $(B, \overline{B})$ *is an undelayed complementary pair. The set of immediate pseudo successors of* A *is denoted* $PSucc_1(A)$.

The set of the pseudo successors of the order $i > 1$ for the action A is defined as

$$PSucc_i(A) \;\; \widehat{=} \;\; \{B \in Action(\mathcal{E}) \mid \exists C \in PSucc_{i-1}(A).\; B \in PSucc_1(C)\}.$$

Then the set of all pseudo successors of the action A is denoted $PSucc(A)$ and defined as follows:

DEFINITION 4 $\quad PSucc(A) \;\; \widehat{=} \;\; \bigcup_{i=1}^{\infty} PSucc_i(A).$

LEMMA 1 $PSucc(\overline{A}) = \{\overline{B} \mid B \in PSucc(A)\}$.

PROOF Use induction on the order of a successor then the definitions of $Succ_1$ and undelayed complementary pair. $\square$

The set of the real successors of A is denoted $RSucc(A)$ and defined as:

DEFINITION 5 $\quad RSucc(A) \;\; \widehat{=} \;\; Path(A, \overline{A}) \setminus PSucc(A)$.

Note that any pseudo successor B can be enforced physically real at the circuit level by implementing the pair $(B, \overline{B})$ as a delayed identity gate. We assume that such a trick is not used. Later on the circuit level interpretation of complementary pairs is considered in more detail.

EXAMPLE 4 Consider again the elementary action system from Example 1:

$$\begin{array}{ll} PSucc(E_a) = \{E_b, E_c\} & \qquad RSucc(E_a) = \{\overline{E}_{d,e}, \overline{E}_f\} \\ PSucc(\overline{E}_a) = \{\overline{E}_b, \overline{E}_c\} & \qquad RSucc(\overline{E}_a) = \{E_{d,e}, E_f\} \end{array}$$

4 Delay-insensitive action systems

In this section, we introduce conditions (stability, noninterference and acknowledgment) which make it possible to implement an elementary action system as a delay-insensitive circuit. The conditions are formulated for closed action systems.

SNI property The necessary prerequisite for the delay-insensitive implementation of an elementary action system $\mathcal{E}$ is the *stability* and *non-interference* requirement. Stability means that (1) an elementary action A is not enabled prematurely, i.e., before all actions in $Path(\overline{A}, A)$ have been executed, (2) it can be disabled only by its real successors. Non-interference, in turn, means that complementary actions A and $\overline{A}$ exclude each other, i.e, that the condition $\neg gA \vee \neg g\overline{A}$ holds invariantly.

Stability and non-interference basically depend on the effective executions of the actions. To exploit this fact we split an elementary action E_v into two parts, the change action E_v^c and the stuttering action E_v^s defined respectively as:

$$\begin{array}{lcl} gE_v \wedge \neg \bigwedge_{i=1}^{n} (v_i = V_i) & \rightarrow & v := V \\ gE_v \wedge \bigwedge_{i=1}^{n} (v_i = V_i) & \rightarrow & v := \neg V. \end{array}$$

Definition 6 *Let $\mathcal{E}$ be a closed elementary action system. For the elementary pair $(E_v, \overline{E}_v)$ from the system $\mathcal{E}$, the* stability *and* non-interference *condition, denoted $SNI(v)$, is defined as*

$$SNI(v) \;\; \widehat{=} \;\; S(E_v) \wedge NI(E_v) \wedge S(\overline{E}_v) \wedge NI(\overline{E}_v)$$

where stability condition $S(A)$ and non-interference condition $NI(A)$ for an action A are the following:

$$\begin{array}{lcl} S(A) & \widehat{=} & (\forall B \in Path(\overline{A}, A).\ B^c \textit{ excludes } A) \\ & \wedge & (\forall B \notin RSucc(A).\ B^c \textit{ cannot disable } A) \\ NI(A) & \widehat{=} & (\overline{A} \textit{ excludes } A). \end{array}$$

The system $\mathcal{E}$ has the stability and non-interference *(SNI) property when the condition $SNI(v)$ is satisfied for each complementary pair $(E_v, \overline{E}_v)$.*

Example 5 Let us check the SNI condition for the system $\mathcal{E}x_1$ from Example 1. We start from the complementary pair $(E_a, \overline{E}_a)$:

$$\begin{array}{ll} Path(E_a, \overline{E}_a) = \{E_b, E_c, \overline{E}_{d,e}\} & PSucc(E_a) = \{E_b, E_c\} \\ Path(\overline{E}_a, E_a) = \{\overline{E}_b, \overline{E}_c, E_{d,e}, E_f\} & PSucc(\overline{E}_a) = \{\overline{E}_b, \overline{E}_c\} \end{array}$$

Now, (i) the change parts of actions in $Path(\overline{E}_a, E_a)$ exclude E_a, and the change parts of actions in $Path(E_a, \overline{E}_a)$ exclude $\overline{E}_a$, (ii) E_a^c, E_b^c, E_c^c cannot disable E_a, and $\overline{E}_a^c$, $\overline{E}_b^c$, $\overline{E}_c^c$ cannot disable $\overline{E}_a^c$. From (i) and (ii) follows that the stability conditions $S(E_a)$ and $S(\overline{E}_a)$ hold for the pair $(E_a, \overline{E}_a)$. Furthermore, E_a and $\overline{E}_a$ exclude each other. Thus the non-interference conditions $NI(E_v)$ and $NI(\overline{E}_v)$ are satisfied as well. Hence, the SNI condition holds for the pair $(E_a, \overline{E}_a)$. In a similar way we can show that the other complementary pairs satisfy the SNI condition as well, and therefore the system $\mathcal{E}x_1$ has the SNI property.

Acknowledgment requirement Recall that informally the acknowledgment requirement is a requirement of every transition on each circuit variable to be acknowledged by a transition on some other circuit variable. In our model, this translates to the requirement that each action has a real successor. We define two acknowledgment requirements: the weak and the strong one. The latter requires each branch in a fork to have a real successor. The former assumes that all forks are isochronic and, therefore, does not contain such requirement.

DEFINITION 7 *A closed elementary action system $\mathcal{E}$ satisfies the* (weak) acknowledgment requirement*, if the following conditions hold:*

(a) $RSucc(A) \neq \emptyset$*, for each action* $A \in Action(\mathcal{E})$ *such that* $(A, \overline{A})$ *is a delayed complementary pair.*

(b) $\bigcup_{i=1}^{k} RSucc(A_i) \neq \emptyset$*, for each action* $A_1 \in Action(\mathcal{E})$ *such that* $(A_1, \overline{A}_1)$ *is undelayed and the pairs* $(A_2, \overline{A}_2), \ldots, (A_k, \overline{A}_k)$ *are its duplicates.*

When additionally the condition

$$\forall A \in Action(\mathcal{E}).\ Succ_1(\overline{A}) = \{\overline{B} \mid B \in Succ_1(A)\}$$

holds for the system $\mathcal{E}$ it is said to satisfy the strong acknowledgment requirement.

Note that the two cases of the weak acknowledgment requirement are due to the fact that a fork can be represented in two different ways in our model.

EXAMPLE 6 For the system $\mathcal{E}x_1$ of Example 1 we have

$$\begin{array}{ll}
\overline{E}_{d,e} \in RSucc(E_a) & E_{d,e} \in RSucc(\overline{E}_a) \\
E_a \in RSucc(E_{d,e}) & \overline{E}_a \in RSucc(\overline{E}_{d,e}) \\
E_a \in RSucc(E_f,) & \overline{E}_a \in RSucc(\overline{E}_f)
\end{array}$$

So the weak acknowledgment requirement is satisfied. Since

$$\begin{array}{ll}
Succ_1(E_c) = \{\overline{E}_{d,e}, \overline{E}_f\} & Succ_1(\overline{E}_c) = \{E_{d,e}, E_f\} \\
Succ_1(E_{d,e}) = \{E_a\} & Succ_1(\overline{E}_{d,e}) = \{\overline{E}_a\} \\
Succ_1(E_f) = \{E_a\} & Succ_1(\overline{E}_f = \{\overline{E}_a\}
\end{array}$$

the strong acknowledgment condition holds as well.

Delay-insensitive action systems Now we are ready to define a (quasi) delay-insensitive action system.

Definition 8 *Let $\mathcal{E}$ be an elementary action system. An action A is called a* final *action, if no action in $\mathcal{E}$ can disable A:*

$$\forall B \in \mathit{Action}(\mathcal{E}).\ B \text{ cannot disable } A.$$

Definition 9 *Let $\mathcal{E}$ be a closed elementary action system. Let all actions in $\mathcal{E}$ be non-final. The system $\mathcal{E}$ is called a* quasi delay-insensitive (QDI) *action system, if it has the SNI property. Furthermore, a QDI action system that satisfies the strong acknowledgment requirement is called a* delay-insensitive (DI) *action system.*

In the QDI action system, the absence of the strong acknowledgment requirement is balanced by the fork isochronicity assumption.

Theorem 1 *Any (quasi) delay-insensitive action system satisfies the (weak) acknowledgment requirement.*

Proof Follows from the SNI condition and the fact that all actions in $\mathcal{E}$ are non-final. □

Example 7 It was shown in Examples 5, 6 that the system $\mathcal{E}x_1$ has the SNI property and satisfies the strong acknowledgment requirement. Since all actions in $\mathcal{E}x_1$ are non-final it is a delay-insensitive action system. In fact, $\mathcal{E}x_1$ models the behavior of C-oscillator.

Example 8 Consider another system

$$
\begin{array}{lll}
\mathcal{E}x_2 :: & |[\ \textbf{var}\ l1, l2, r1, r2, m1, m2.\ l1{\downarrow}, l2{\downarrow}, r1{\downarrow}, r2{\downarrow}, m1{\downarrow}, m2{\downarrow}; & \\
 & \quad \textbf{do}\ l1 \wedge \neg l2 \wedge \neg r2 \rightarrow m1{\uparrow} & [E_{m1}] \\
 & \quad\ \ [\!]\ \ m1 \rightarrow m2{\uparrow} & [E_{m2}] \\
 & \quad\ \ [\!]\ \ m2 \wedge m1 \rightarrow r1{\uparrow} & [E_{r1}] \\
 & \quad\ \ [\!]\ \ r1 \rightarrow r2{\uparrow} & [E_{r2}] \\
 & \quad\ \ [\!]\ \ r2 \rightarrow m1{\downarrow} & [\overline{E}_{m1}] \\
 & \quad\ \ [\!]\ \ \neg m1 \rightarrow m2{\downarrow} & [\overline{E}_{m2}] \\
 & \quad\ \ [\!]\ \ \neg m1 \wedge r1 \rightarrow l2{\uparrow} & [E_{l2}] \\
 & \quad\ \ [\!]\ \ l2 \rightarrow l1{\downarrow} & [\overline{E}_{l1}] \\
 & \quad\ \ [\!]\ \ l2 \rightarrow r1{\downarrow} & [\overline{E}_{r1}] \\
 & \quad\ \ [\!]\ \ \neg r1 \rightarrow r2{\downarrow} & [\overline{E}_{r2}] \\
 & \quad\ \ [\!]\ \ \neg l1 \wedge \neg r1 \wedge \neg m2 \rightarrow l2{\downarrow} & [\overline{E}_{l2}] \\
 & \quad\ \ [\!]\ \ \neg l2 \rightarrow l1{\uparrow} & [E_{l1}] \\
 & \quad \textbf{od}\ ]| &
\end{array}
$$

which models the behavior of a circuit that synchronizes three 4-phase handshaking cycles on the 'ports' $(l1, l2)$, $(m1, m2)$ and $(r1, r2)$. The operation of $\mathcal{E}x_2$ starts after the initialization, when the variable $l1$ is set to *true* by the action E_{l1}.

In the same way as in Example 5 we can show that the system $\mathcal{E}x_2$ has the SNI property. Moreover, all actions in $\mathcal{E}x_2$ are non-final. Hence, it is a quasi delay-insensitive action system. However, the strong acknowledgment requirement is violated. As an example, consider the complementary pair $(E_{m2}, \overline{E}_{m2})$. We have:

$$Succ_1(E_{m2}) = \{E_{r1}\} \qquad\qquad Succ_1(\overline{E}_{m2}) = \{\overline{E}_{l2}\}$$

Thus, $\mathcal{E}x_2$ is not a delay-insensitive system.

5 Modularity

Delay-insensitivity as described above is a property of a closed action system. However, in practice circuits are designed in a modular fashion. Therefore, it is important to formulate (Q)DI property for open systems as well. An open elementary action system is called an *elementary component*. Recall that all imported variables are read-only in such a system.

(Q)DI components We assume that the communication between elementary components is based on four-phase handshaking protocol. This means that each component contains global variables that correspond to request and acknowledgment signals in a circuit.

DEFINITION 10 *Let $\mathcal{A}$ be an elementary component. Assume that one of the variables a, b is exported, whereas the other is imported variable in $\mathcal{A}$. Let a be a request variable and let b be an acknowledgment variable. Then the variables a, b form a* communication port *of the component $\mathcal{A}$. If the request variable a is exported, the port is an* active *port, denoted (a^*, b). Otherwise, it is a* passive *port, denoted (a, b^*).*

In the environment of an elementary component $\mathcal{A}$, only transitions on the request and acknowledgment variables can affect the delay-insensitivity of $\mathcal{A}$. Therefore, we define an abstract environment of $\mathcal{A}$ which follows the handshaking protocol.

DEFINITION 11 *Let $\mathcal{A}$ be an elementary component. Assume that the communication ports of $\mathcal{A}$ are $(v_1^*, x_1), \ldots, (v_m^*, x_m)$ and $(y_1, w_1^*), \ldots, (y_n, w_n^*)$. For each active port (v_i^*, x_i), define the following complementary pair:*

$$(P_{x_i}, \overline{P}_{x_i}) \quad \widehat{=} \quad (v_i \rightarrow x_i\!\uparrow, \neg v_i \rightarrow x_i\!\downarrow)$$

For each passive port (y_j, w_j^*), define the following complementary pair:

$$(P_{y_j}, \overline{P}_{y_j}) \quad \widehat{=} \quad (\neg w_j \rightarrow y_j\!\uparrow, w_j \rightarrow y_j\!\downarrow)$$

Then the action system

$$\begin{array}{lll} \mathcal{P}_{\mathcal{A}} :: & |[& \mathbf{var}\ x^*, y^*.\ x\!\downarrow; y\!\downarrow; \\ & & \mathbf{do}\ P_{x_1} \parallel \overline{P}_{x_1} \parallel \ \ldots \ \parallel P_{x_m} \parallel \overline{P}_{x_m} \\ & & \parallel \quad P_{y_1} \parallel \overline{P}_{y_1} \parallel \ \ldots \ \parallel P_{y_n} \parallel \overline{P}_{y_n} \\ & & \mathbf{od} \\ &]|\!: v, w & \end{array}$$

where $x = x_1, \ldots, x_m$, $y = y_1, \ldots, y_n$, $v = v_1, \ldots, v_m$ and $w = w_1, \ldots, w_n$, is the protocol system *(abstract environment) of $\mathcal{A}$. The actions of $\mathcal{P}_{\mathcal{A}}$ are called the* protocol actions *of $\mathcal{A}$.*

The following lemma holds for the parallel composition of an elementary component $\mathcal{A}$ and its protocol system $\mathcal{P}_{\mathcal{A}}$:

LEMMA 2 $|[\ \mathbf{var}\ v, w, x, y \bullet \mathcal{A} \parallel \mathcal{P}_{\mathcal{A}}\ ]|$ *is a closed elementary action system.*

PROOF Follows from the construction of $\mathcal{P}_{\mathcal{A}}$ and the definition of an elementary action system. □

The system $|[\ \mathbf{var}\ v, w, x, y \bullet \mathcal{A} \parallel \mathcal{P}_{\mathcal{A}}\ ]|$ is called the *closure* of $\mathcal{A}$ and denoted $\mathcal{C}(\mathcal{A})$. The notion of protocol system allows us to define (Q)DI property for an elementary component in terms of (Q)DI property for a closed action system.

DEFINITION 12 *Let $\mathcal{A}$ be an elementary component. Consider every complementary pair in the protocol system $\mathcal{P}_{\mathcal{A}}$ to be delayed. Then $\mathcal{A}$ is a* (Q)DI component*, if its closure $\mathcal{C}(\mathcal{A})$ is a (Q)DI action system.*

Note Let E_v be an action from $\mathcal{A}$. Considering an elementary pair from the protocol system $\mathcal{P}_{\mathcal{A}}$ to be delayed ensures that the action E_v has a real successor in $\mathcal{C}(\mathcal{A})$. Otherwise, the weak acknowledgment requirement would be violated. The assumption reflects the fact that the action E_v necessarily has a real successor in any specific (Q)DI component replacing $\mathcal{P}_{\mathcal{A}}$.

EXAMPLE 9 Consider the elementary component $\mathcal{S}ync$ with two communication ports:

$$\begin{array}{lll} \mathcal{S}ync :: & |[& \mathbf{var}\ l2^*, r1^*.\ l2\!\downarrow; r1\!\downarrow; \\ & & \mathbf{do}\ r2 \rightarrow l2\!\uparrow \ \parallel \ \neg r2 \wedge \neg l1 \rightarrow l2\!\downarrow \\ & & \parallel \quad l1 \wedge \neg l2 \rightarrow r1\!\uparrow \ \parallel \ \neg l1 \vee l2 \rightarrow r1\!\downarrow \\ & & \mathbf{od} \\ &]|\!: l1, r2 & \end{array}$$

The left port $(l1, l2^*)$ is passive whereas the right port $(r1^*, r2)$ is active. Hence, the protocol system for $\mathcal{S}ync$ is as follows:

$$
\begin{array}{ll}
\mathcal{P}_{\mathcal{S}ync} :: & |[\ \ \mathbf{var}\ l1^*, r2^*.\ l1{\downarrow}; r2{\downarrow}; \\
& \quad \mathbf{do}\ \neg l2 \to l1{\uparrow}\ \ [\!]\ \ l2 \to l1{\downarrow} \\
& \quad\ \ [\!]\ \ r1 \to r2{\uparrow}\ \ [\!]\ \ \neg r1 \to r2{\downarrow} \\
& \quad \mathbf{od} \\
&]|\!: l2, r1
\end{array}
$$

Since the closure $\mathcal{C}(\mathcal{S}ync)$ is a (Q)DI system, the component $\mathcal{S}ync$ is a (Q)DI component. In fact, it synchronizes two handshaking cycles on the left and right communication ports, respectively.

Composition of (Q)DI components (Q)DI components can be composed giving a new (Q)DI component, provided they are 'compatible'.

DEFINITION 13 *Let $\mathcal{A}$ and $\mathcal{B}$ be (Q)DI components and let p and q be communication ports of $\mathcal{A}$ and $\mathcal{B}$, respectively. We say that p and q form a* channel *between $\mathcal{A}$ and $\mathcal{B}$, if $\{p, q\} = \{(a^*, b), (a, b^*)\}$ for some variables a, b from $\mathcal{A}$ and $\mathcal{B}$. The channel is written as (a, b).*

Let $\mathcal{A}$ be a (Q)DI component with the communication ports $p_1, \ldots, p_m$ and let $\mathcal{B}$ be another (Q)DI component with the ports $q_1, \ldots, q_n$. The notion of compatibility is formally defined as follows:

DEFINITION 14 *We say that the (Q)DI components $\mathcal{A}$ and $\mathcal{B}$ are* compatible, *if for each pair (p_i, q_j) either of the following holds: (i) p_i and q_j form a channel, or (ii) p_i and q_j do not refer to the same variables.*

The parallel composition is only defined for compatible (Q)DI components $\mathcal{A}$ and $\mathcal{B}$. Let $(a_1, b_1), \ldots, (a_k, b_k)$ be the channels between $\mathcal{A}$ and $\mathcal{B}$. Then the system $\mathcal{A} \parallel\!| \mathcal{B}$, defined as

$$\mathcal{A} \parallel\!| \mathcal{B} \quad \hat{=} \quad |[\ \mathbf{var}\ a, b \bullet \mathcal{A} \parallel \mathcal{B}\ ]|,$$

where $a = a_1, \ldots, a_k$ and $b = b_1, \ldots, b_k$, is the parallel composition of the compatible (Q)DI components $\mathcal{A}$ and $\mathcal{B}$.

We have that the (Q)DI property is preserved by the parallel composition operator:

THEOREM 2 *Let $\mathcal{A}$ and $\mathcal{B}$ be (Q)DI components. Then their parallel composition $\mathcal{A} \parallel\!| \mathcal{B}$ is a (Q)DI component.*

PROOF Assume (v, x) is a channel between $\mathcal{A}$ and $\mathcal{B}$. Let P_x be a protocol action related to the port (v^*, x) of $\mathcal{A}$. Let $(E_u, \overline{E}_u)$ be a complementary pair from $\mathcal{A}$. Assume, $P_x \in Path^{\mathcal{C}(\mathcal{A})}(E_u, \overline{E}_u)$. In the path $Path^{\mathcal{A} \,|||\, \mathcal{B}}(E_u, \overline{E}_u)$, the action P_x is replaced by a number of actions $Path^{\mathcal{A} \,|||\, \mathcal{B}}(E_v, E_x) \cup \{E_x\}$ from the component $\mathcal{B}$. However, actions from $Path^{\mathcal{A} \,|||\, \mathcal{B}}(E_v, E_x)$ do not assign to the variables of $\mathcal{A}$. The actions E_x and P_x assign the same value to x since they follow the same handshaking protocol. Therefore, SNI(u) in $\mathcal{A} \,|||\, \mathcal{B}$ is equivalent to the SNI(u) in $\mathcal{C}(\mathcal{A})$. Hence, SNI(u) holds in $\mathcal{A} \,|||\, \mathcal{B}$. $\square$

Note The superscript in $Path^{\mathcal{E}}(A, B)$ indicates the system in which the path between A and B is calculated.

6 Circuit elements

In this section we show how elementary components are related to the gate-level constructs of a circuit.

Circuit elements Depending on the component library, different sets of basic circuit elements are available to a circuit designer. Therefore, we first introduce a notion of an abstract circuit element.

DEFINITION 15 *Consider an elementary component*

$$\mathcal{E} :: \quad |[\ \mathbf{var}\ v^*.v{\downarrow}; \mathbf{do}\ E_v \ [\!] \ \overline{E}_v \ \mathbf{od} \ ]| : (vE_v \cup v\overline{E}_v) \setminus \{v\}$$

Let $(E_v, \overline{E}_v)$ be either unbranched or undelayed branched complementary pair. Then the component $\mathcal{E}$ is called a circuit element.

When the guards gE_v and $g\overline{E}_v$ are complements of each other, so that

$$g\overline{E}_v \equiv \neg gE_v$$

holds invariantly, the circuit element is *combinational*. If the pair $(E_v, \overline{E}_v)$ is unbranched, the corresponding combinational element is called a *gate* and denoted $Gate(gE_v; v)$.

On the other hand, when gE_v and $g\overline{E}_v$ can be *false* simultaneously, so that

$$g\overline{E}_v \neq \neg gE_v$$

holds, the circuit element is *sequential*.

Basic circuit elements Below we show how typical circuit elements, called *basic elements*, are derived as specific instances of an abstract circuit element. Other basic elements could be defined in a similar way. Basic combinational gates are as follows:

$$\begin{array}{lcl}
wire(a;v) & \widehat{=} & Gate(a;v) \\
invert(a;v) & \widehat{=} & Gate(\neg a;v) \\
nand(a_1,\ldots,a_n;v) & \widehat{=} & Gate(\neg \bigwedge\limits_{i=1}^{n} a_i;v) \\
nor(a_1,\ldots,a_n;v) & \widehat{=} & Gate(\neg \bigvee\limits_{i=1}^{n} a;v) \\
and(a_1,\ldots,a_n;v) & \widehat{=} & Gate(\bigwedge\limits_{i=1}^{n} a_i;v) \\
or(a_1,\ldots,a_n;v) & \widehat{=} & Gate(\bigvee\limits_{i=1}^{n} a;v) \\
xor(a,b;v) & \widehat{=} & Gate(a \neq b;v).
\end{array}$$

The input parameters $a_1, \ldots, a_n$ above represent the imported variables, while the output parameter v is the exported variable in the corresponding action system.

If $(E_v,\overline{E}_v)$ is an undelayed branched pair of the form

$$(\ a \rightarrow v_1{\uparrow}, \ldots, v_n{\uparrow},\quad \neg a \rightarrow v_1{\downarrow}, \ldots, v_n{\downarrow}\ )$$

the corresponding combinational circuit element is called a *fork* and denoted $fork(a; v_1, \ldots, v_n)$.

Different types of C-elements are the basic sequential elements. Let the complementary pair $(E_v, \overline{E}_v)$ be

$$(\ a \wedge b \rightarrow v{\uparrow},\quad \neg a \wedge \neg b \rightarrow v{\downarrow}\ ).$$

Then the corresponding circuit element $\mathcal{E}$, denoted $C(a, b; v)$, is a normal (symmetric) C-element. Down-asymmetric and up-asymmetric C-elements, written as $dC(a, b; v)$ and $uC(a, b; v)$, correspond to the following complementary pairs

$$\begin{array}{ll}
(\ a \wedge b \rightarrow v{\uparrow}, & \neg b \rightarrow v{\downarrow}\ ) \\
(\ a \rightarrow v{\uparrow}, & \neg a \wedge \neg b \rightarrow v{\downarrow}\ )
\end{array}$$

respectively.

Implementing (Q)DI components Any (Q)DI component is trivially implemented as a parallel composition of (abstract) circuit elements. If these are available as library components, we have a 'physical' implementation which preserves the (Q)DI property. Note that undelayed duplicates are supposed to be implemented as a single fork.

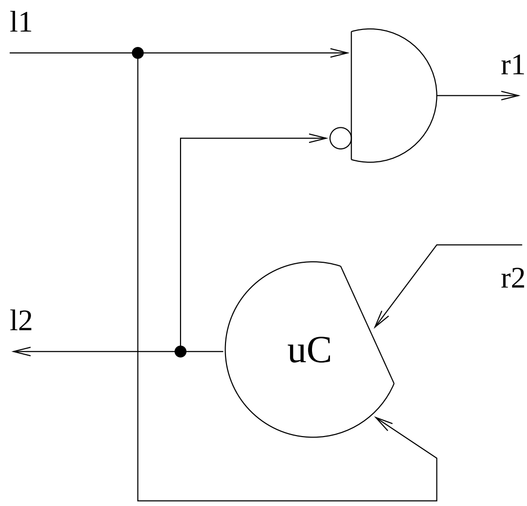

Figure 1: Implementation of the synchronizer $\mathcal{S}ync$.

EXAMPLE 10 Consider the (Q)DI component $\mathcal{S}ync$ from Example 9. It is decomposed into two circuit elements $\mathcal{E}_1$ and $\mathcal{E}_2$ defined as follows:

$$\begin{array}{ll} \mathcal{E}_1 :: & |[\ \textbf{var}\ l2^*.l2{\downarrow};\ \textbf{do}\ r2 \rightarrow l2{\uparrow}\ \talloblong\ \neg r2 \wedge \neg l1 \rightarrow l2{\downarrow}\ \ \textbf{od}\ ]| : l1, r2 \\ \mathcal{E}_2 :: & |[\ \textbf{var}\ r1^*.r1{\downarrow};\ \textbf{do}\ l1 \wedge \neg l2 \rightarrow r1{\uparrow}\ \talloblong\ \neg l1 \vee l2 \rightarrow r1{\downarrow}\ \ \textbf{od}\ ]| : l1, l2 \end{array}$$

The former is implemented as up-asymmetric C-element, the later as AND-gate. Formally, we have:

$$\begin{array}{rcl} \mathcal{S}ync & = & \mathcal{E}_1 \parallel \mathcal{E}_2 \\ & = & uC(r2, l1; l2) \parallel and(l1, \neg l2; r1). \end{array}$$

An abstract circuit element is mapped to a number of the basic elements when it is not available as a library component. However, the mapping and its physical implementation must ensure that hazard-free behaviour of the original element is preserved. For more on this topic, see [14].

EXAMPLE 11 Consider the following (abstract) sequential element:

$$\begin{array}{ll} \mathcal{S}eq :: & |[\ \ \textbf{var}\ v^*.\ v{\downarrow}; \\ & \qquad \textbf{do}\ a \wedge b \rightarrow v{\uparrow}\ \talloblong\ \neg b \wedge c \rightarrow v{\downarrow}\ \ \textbf{od} \\ &]|: a, b \end{array}$$

Assume that $\mathcal{S}eq$ is not available as a library component. Nevertheless, it can be implemented as the parallel composition $\mathcal{E}_1 \parallel \mathcal{E}_2 \parallel \mathcal{E}_3$ of simpler circuit elements defined as:

$$\begin{array}{ll} \mathcal{E}_1 :: & |[\textbf{ var } x.x{\downarrow};\ \textbf{do } a \wedge b \to x{\uparrow} \ \| \ \neg a \vee \neg b \to x{\downarrow} \ \textbf{ od }]| : a, b \\ \mathcal{E}_2 :: & |[\textbf{ var } y.y{\downarrow};\ \textbf{do } b \vee \neg c \to y{\uparrow} \ \| \ \neg b \wedge c \to y{\downarrow} \ \textbf{ od }]| : b, c \\ \mathcal{E}_3 :: & |[\textbf{ var } v.v{\downarrow};\ \textbf{do } x \wedge y \to v{\uparrow} \ \| \ \neg x \wedge \neg y \to v{\downarrow} \ \textbf{ od }]| : x, y \end{array}$$

The circuit elements $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$ correspond to AND-gate, NAND-gate and C-element, respectively. Hence, the element $\mathcal{S}eq$ is implemented as follows:

$$and(a, b; x) \parallel nand(\neg b, c; y) \parallel C(x, y; v).$$

The AND-gate and NAND-gate are inputs of the C-element which keeps the state of v stable in the situations, where the condition $(\neg b \wedge c) \neq \neg(a \wedge b)$ holds.

7 Conclusions

We have shown how action systems can be used to describe asynchronous delay-insensitive circuits. We started by defining the concept of an elementary action system. Then we introduced the properties an elementary system must possess in order to be (quasi) delay-insensitive. The concept of a (Q)DI component was introduced which makes it possible to design a circuit in modular style. We defined the notion of a protocol system which models on an abstract level the communication behaviour of the actual environment. The basic circuit elements used in the designs were defined as elementary action systems of a special form. We also demonstrated how elementary action systems are mapped to basic circuit elements.

The main advantage we gain by using action systems is the possibility to develop and reason about circuits within one unifying framework, i.e., one language, in all levels of the development activity. Furthermore, the *refinement calculus* [3] for action systems gives us a solid foundation to carry out the development of a circuit in a stepwise, transformational manner. Every step in the design process can be formally proven correct within this calculus, even mechanically [10].

A (Q)DI action system is a low level description of a circuit. Therefore, it might be difficult to directly find such system for a specific application. In an accompanying paper [14], we develop a method to synthesize delay-insensitive circuits from high-level action systems specifications. The synthesis method is based on the refinement calculus, which supports the idea of program refinements and transformations. The method is exemplified by a case study on a pipeline controller [15]. Unlike several other approaches, we have explicit wires in our model. This allows us to check 'communication delay-insensitivity' of two or more interacting systems. In this way, both quasi delay-insensitive (or speed-independent) and delay-insensitive decomposition, are supported by our model.

Related work The action systems framework is similar to the UNITY approach of Chandy and Misra [6]. Action systems are derived within the refinement calculus whereas UNITY comes with an associated, temporal logic based specification language. Moreover, a subset of action systems [3] has been shown to correspond to CSP and occam. Hence, circuit design methods based on these formalisms are related to the approach put forward in this paper.

The *communicating hardware processes* of Martin [12, 13] is a CSP-based design language for asynchronous quasi delay-insensitive circuits. This language has been the original inspiration to our work and hence, the reader familiar with this approach can find many similarities with our work. Unlike us, Martin defines several languages that are used in different phases of the design process. The main difference is, however, in the way the formalism is intended to be used in the design process: our language is designed with the refinement calculus in mind whereas no such formal basis is given for communicating hardware processes.

Smith and Zwarico [18] give a formal basis and correctness proof for the design method of Martin. Like we do here, also they use a single, CSP-based language for the description of the entire design process. Their main motivation is, however, to show the correctness of the compilation method of Martin. Our motivation is to put forward a very general refinement calculus based design method.

The design language *Tangram* [20] is inspired by CSP and occam. The language supports automatic compilation of Tangram programs into quasi delay-insensitive handshake circuits. To our knowledge, the translation from these to basic circuit elements is not supported by a fully formal framework. The handshake algebra [11] is a formalism suited for CSP-style algebraic manipulation of expressions modelling circuits. Ebergen's program notation *commands* [9] is inspired by CSP and Dijkstra's guarded commands. The mathematical background of this formalism is the *trace theory* [17]. Labeled Petri nets are an example of a different approach based on the trace theory. To some extent, the language of Petri nets provides a unified framework [21] for both, abstract synthesis of the control circuit and logic synthesis at the gate level.

Synchronized transitions of Staunstrup [19] has its roots in the UNITY-formalism [6]. It is meant for modelling and verifying hardware in general, and it contains a refinement concept and a methodology to prove correctness of refinement steps. A pure UNITY-approach for modelling asynchronous hardware systems has been introduced by Bickford [4]. His main objective is to define composable specifications of circuits along the lines we propose in section 5 and 6 of this paper.

Acknowledgments

The authors like to thank Alain Martin for providing the initial inspiration to this work. Also, discussions with Ralph Back, Tom Kuusela and Joakim von Wright have been very fruitful. The work reported here was supported by the Academy of Finland.

References

[1] R. J. R. Back and R. Kurki-Suonio. Distributed Co-operation with Action Systems. *ACM Transactions on Programming Languages and Systems*, Vol 10, No 4, 1998, pp. 513–554.

[2] R. J. R. Back, A. J. Martin, and K. Sere. Specifying the Caltech asynchronous microprocessor. *Science of Computer Programming* 26(1996), pp. 79–97, Elsevier.

[3] R. J. R. Back and K. Sere. From Action Systems to Modular Systems. *Software - Concepts and Tools* (1996) 17, pp. 26–39, Springer Verlag

[4] M. Bickford. Composable specifications for asynchronous systems using UNITY. In *Proceedings of the 1st International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 216–227, 1994.

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

[6] K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison–Wesley, 1988.

[7] A. Davis and S. M. Nowick. Asynchronous circuit design: motivation, background and methods. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, pages 1–49. Springer, 1995.

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

[9] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5:107–119, 1991.

[10] T. Långbacka, R. Rukšėnas, and J. von Wright. TkWinHOL: A Tool for Window Inference in HOL. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications – 8th International Workshop, Aspen Grove, Utah, USA*, volume 971 of *LNCS*, pages 245–260, Springer–Verlag, 1995.

[11] M. B. Josephs, J. T. Udding, and J. T. Yantchev. Handshake Algebra, South Bank University Technical Report SBU-CISM-93-1, 1993.

[12] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226–234, 1986.

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

[14] J. Plosila, R. Rukšėnas, K. Sere, and T. Kuusela Action Systems Synthesis of DI Circuits. Manuscript, 1996.

[15] J. Plosila, R. Rukšėnas, K. Sere, and T. Kuusela Action Systems Synthesis of DI Circuits – A Case Study. Manuscript, 1996.

[16] J. Plosila and K. Sere. Action Systems in Pipelined Processor Design. Technical Report 54, Turku Centre for Computer Science, 1996. Available from `http://www.tucs.abo.fi/publications/techreports/TR54.ps.gz`.

[17] M. Rem, J. L. A. van de Snepscheut, and J. T. Udding. Trace theory and the definition of hierarchical components. In R. Bryant, editor, *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 225–239. CSP, Inc., 1983.

[18] S. F. Smith and A. E. Zwarico. Correct compilation of specifications to deterministic asynchronous circuit. *Formal Methods in System Design*, 7:155–226, 1995.

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

[20] K. van Berkel. *Handshake circuits: an asynchronous architecture for VLSI programming*. International Series on Parallel Computation. Cambridge University Press, 1993.

[21] A. V. Yakolev, A. M. Koelmans, and L. Lavagno. High level modelling and design of asynchronous interface logic. *IEEE Design & Test of Computers*, 12(1):32–40, 1995.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**