Action Systems Synthesis of DI Circuits

Juha Plosila

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

Rimvydas Rukšėnas

Turku Centre for Computer Science (TUCS)
Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14, FIN-20520 Turku, Finland

Kaisa Sere

Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14, FIN-20520 Turku, Finland

Turku Centre for Computer Science
TUCS Technical Report No 149
December 1997
ISBN 952-12-0121-5
ISSN 1239-1891

Abstract

A sub-class of *action systems* has recently been shown to model the behaviour of asynchronous *delay-insensitive circuits*. Hence, the action systems formalism can be used to design such circuits. The set of action systems belonging to this sub-class is, however, rather limited and therefore it might be difficult to find such a system directly. In this paper we introduce a design method for delay-insensitive circuits. We show how an arbitrary action system can be stepwise transformed into a delay-insensitive form that allows a mapping to digital circuit elements. The transformations are *correctness preserving* and based on the *Refinement Calculus*.

Keywords: action systems, asynchronous circuits, refinement, handshaking expansion, circuit extraction

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Delay-insensitive (DI) circuits are devices, whose timing and correctness does not depend on delays in circuit elements [11]. 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. Since the set of DI circuits is extremely limited, we rely on a weaker acknowledgment requirement by assuming [18] that the forks are *isochronic*. A transition on an isochronic fork is considered acknowledged, when a transition on one of its branches has been acknowledged. DI circuits under such assumption are called *quasi* delay-insensitive (QDI) circuits [11].

The *action systems formalism* [4] has been recently extended so that it can be used to model the behavior of (Q)DI circuits [18]. In this paper, we continue on this subject by introducing a design method for (Q)DI circuits. We will show how to transform an arbitrary, abstract action system description stepwise into the delay-insensitive form which can be directly mapped into a network of digital circuit elements. The transformations are based on the *Refinement Calculus* [2] which guarantees the correctness of the design steps and hence, the correctness of the designed circuit with respect to the high-level action system description of the intended behaviour of the circuit.

The design process consists of four main phases: initial derivation, handshaking expansion, circuit extraction, and physical implementation. The design steps separate the most essential transformations from each other structuring the derivation process into manageable parts. In the *initial derivation* the description of a circuit is regarded as a parallel composition of action systems each describing some functional aspect of the target circuit. The reactive components communicate initially using shared variables. The *handshaking expansion*, where all communication variables needed in delay-insensitive interaction are introduced, brings the description closer to the circuit level. The abstract shared variable model of the initial derivation is transformed into an implementable communication protocol. We make a careful analysis of the different types of communication channels in the systems and propose transformations for each of them. The *circuit extraction* is then the design step, where an action system is transformed into the (quasi) delay-insensitive form. Here the circuit elements are brought about while respecting the communication protocol introduced in the previous step. The purpose of the last design step, *physical implementation*, is to find the most advantageous realization for the target circuit based on the different sets of basic circuit elements available for the circuit designer.

There are many formalisms described in the literature [1, 7, 9, 13, 15, 26] for asynchronous circuit design. Our synthesis method is based on the idea of using methods familiar from the design of parallel systems when deriving circuits. This idea of reasoning about circuits in a framework for parallel processes is not new. Our research is mainly inspired by the work of Martin [17, 16] who develops a CSP-based language, the *communicating hardware processes*, for the synthesis of asynchronous delay-insensitive circuits. The design phases basically follow those proposed by Martin. The main novelty in our approach is a fully formal basis, the Refinement Calculus, used in our synthesis method. The Refinement Calculus allows to carry out synthesis of the circuit simultaneously with the proof of its correctness using a tool [14] based on HOL theorem prover. Smith and Zwarico [24] have also given a correctness proof for the Martin's method. However, we feel that an approach like ours which allows to prove required transformation rules on demand is more flexible.

The design language *Tangram* [22], developed in Philips Research Laboratories, is based on CSP and OCCAM. A Tangram program is automatically compiled into an intermediate form, a network of *handshake circuits* [6], which is then implemented on silicon as a (quasi) delay-insensitive circuit. Our way of modelling the communication channels between two action systems at the handshaking expansion has been inspired by the handshake circuits.

Synchronized transitions [25], a UNITY-based [8] design framework, contains a refinement concept and a methodology to prove correctness of refinement steps. Eventhough action systems are closely related to UNITY programs, these two formalisms differ in the sense that action systems are derived within the refinement calculus using program-transformation rules rather than using temporal logic reasoning about specifications as done in UNITY.

Overview of the paper The paper is organized as follows. In section 2, we describe the action systems formalism to an extent needed in this paper. We also define a sub-class of action systems, the (quasi) delay-insensitive action systems which have been shown to correspond to delay-insensitive circuits. Moreover, the basic concepts for the derivation of provably correct action systems within the Refinement Calculus are introduced. Sections 3-6 describe the proposed synthesis method. In section 3, we discuss the initial derivation phase of our method. Here as well as in section 4, we concentrate on the communication structure in a parallel composition of action systems. In section 4 we show how a four-phase handshaking protocol between a pair of action systems is introduced as a program refinement step. Section 5 shows how an arbitrary handshake-expanded action system can be stepwise turned into a (quasi) delay-insensitive action system. Finally, in section 6 we discuss the circuit level implementation of (quasi) delay-insensitive action systems. We end in section 7 with some concluding remarks.

2 Action systems

In this section we give a brief overview of the action systems formalism and define a specific sub-class of action systems that can be compiled into circuits, the (quasi) delay-insensitive action systems. Finally, we show how action systems are refined within the Refinement Calculus framework.

2.1 Action systems

The action systems formalism is based on Dijkstra's language of *guarded commands* [12]. This language includes assignment, sequential composition, conditional choice, and iteration.

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

$$g \to S$$

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* [12] predicate transformers in the usual manner. For an action A, the weakest precondition is defined as follows:

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

Total correctness of an action A wrt a *precondition* p and a *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 An action is said to *disable* itself, if the condition

$$true \, \langle A \rangle \, \neg gA$$

holds. Two actions, A and B, *exclude* each other, if $gA \Rightarrow \neg gB$. If an action does not change the program state it is called a *stuttering action*. Namely, 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 is called a *change action*, if it is not stuttering one.

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

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

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 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 [4]. Their parallel execution is then equivalent to executing the actions one after the other, in either order.

Hiding Let x be an exported variable of the action system

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

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

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

Notation We often want to indicate that actions in an action system have to be executed sequentially. The notation of *sequential action systems* is introduced for such cases:

$$\mathcal{A} :: \ |[\ \textbf{do}\, A_1; \ \ldots \ A_i; \ \ldots \ A_m \ \textbf{od} \ ]| : z$$

The notation hides a local variable pc in the actions of $\mathcal{A}$ and actually stands for the following action system:

$$\begin{array}{lll}
\mathcal{A} :: & |[& \textbf{var}\, p \cdot pc := 0; \\
& & \textbf{do} \ \ pc = 0 \wedge gA_1 \rightarrow sA_1; \ pc := 1 \\
& & \ \ldots \ [\!] \ pc = i - 1 \wedge gA_i \rightarrow sA_i; \ pc := i \\
& & \ \ldots \ [\!] \ pc = m - 1 \wedge gA_m \rightarrow sA_m; \ pc := 0 \\
& & \textbf{od} \\
&]| : z &
\end{array}$$

The hidden variable pc is referred to as the *program counter*. Later on we will need a more abstract way to describe the control flow in action systems. Thus for an action system, $\mathcal{A}$, and a boolean condition, b, we define $at_{\mathcal{A}}(b)$ to be the *smallest* set of program counter values K such that the predicate $b \Rightarrow (pc \in K)$ is a local invariant of $\mathcal{A}$. By the definition, we have that $b \Rightarrow (pc \in at_{\mathcal{A}}(b))$ is a local invariant of $\mathcal{A}$.

We also use the notation $v\!\uparrow$ ($v\!\downarrow$) to represent an assignment of the value $true$ ($false$) to the variable (variables) v.

Composition of action systems Consider two action systems,

$$\begin{array}{ll}
\mathcal{A} :: & |[\ \textbf{var}\, v \cdot I; \ \textbf{do}\, A_1 \ [\!] \ \ldots \ [\!] \ A_m \ \textbf{od} \ ]| : x \\
\mathcal{B} :: & |[\ \textbf{var}\, w \cdot J; \ \textbf{do}\, B_1 \ [\!] \ \ldots \ [\!] \ B_n \ \textbf{od} \ ]| : y
\end{array}$$

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 $\mathcal{C}$:

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

The parallel composition operator is also used to decompose one action system into a number of those.

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 [4].

(Q)DI action systems An *elementary action*, E_v, is a guarded command of the form

$$b_1 \rightarrow v := V$$

where $v = v_1, \ldots, v_n$, and $V = V_1, \ldots, V_n$ are such that either $(\bigwedge_{i=1}^{n} V_i)$ or $(\bigwedge_{i=1}^{n} \neg V_i)$ holds. In sequential action systems, an elementary action may also contain an assignment to the program counter. The elementary action

$$b_2 \rightarrow v := \neg V$$

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 $\overline{E}_v$). A pair, consisting of an action and its complement, is called an *element*. If $n = 1$ then $(E_v, \overline{E}_v)$ is called a *single-output* element, otherwise it is a *multi-output* element. The element $(E_v, \overline{E}_v)$ is *stable*, if neither of the actions disables itself. The element is *non-interfering*, if the action E_v excludes its complement $\overline{E}_v$, i.e. $b_1 \Rightarrow \neg b_2$. Note that actions are non-interfering but unstable in sequential action systems. An element is called a *(quasi) delay-insensitive element* (a (Q)DI element), if it is both stable and non-interfering, and refers to boolean variables only.

Consider an action system of the form:

$$\mathcal{E} ::\ \ |[\ \mathbf{var}\ w \cdot J;\ \mathbf{do}\ E_{w_1}\ [\!]\ \overline{E}_{w_1}\ [\!]\ \cdots\ [\!]\ E_{w_m}\ [\!]\ \overline{E}_{w_m}\ \mathbf{od}\ ]| : z$$

where the imported variables z are read-only in $\mathcal{E}$, $w = w_1, \ldots, w_m$, and w_i, $i = 1, \ldots, m$, denote pairwise disjoint sets of boolean variables initialized by the action J. If each pair $(E_{w_i}, \overline{E}_{w_i})$ is a (Q)DI element, then the system $\mathcal{E}$ is called a *(quasi) delay-insensitive action system* (a (Q)DI system).

Hence, if we have a parallel composition of two (Q)DI systems $\mathcal{E} : z$ and $\mathcal{F} : u$, they communicate via the shared variables z and u. Observe that these variables are accessed in a read-only manner in the importing action system.

In our earlier work [18] we have shown that (Q)DI systems model the behaviour of asynchronous delay-insensitive circuits. In the rest of the paper we develop a method for the stepwise derivation of provably correct (Q)DI systems from arbitrary action systems. Later in section 6 we show how (Q)DI action systems are related to the circuit gate-level constructs.

2.2 Refinement of action systems

A formal basis for the stepwise development of action systems is the *Refinement Calculus*. In the Refinement Calculus, program statements are identified with their weakest precondition predicate transformers. However, the predicate transformer framework is not sufficient to reason about reactive systems. An extension (trace refinement) of the framework to capture the reactive behaviour is described by Back and von Wright [5]. Our treatment of the action system refinement is based on the theory presented there.

Trace refinement The reactive behaviour of an action system is described in terms of *traces* (observable behaviours). For an action system, a trace t is a sequence of global states encountered during the execution of the action system: $t = \langle u_0, u_1, \ldots \rangle$. Traces can be finite or infinite. If $u_i = u_{i+1}$, then the state u_{i+1} is a *stuttering* state. All finite subsequences of stuttering states are removed from traces. A sequence with infinitely many stuttering states, $\langle u_0, \ldots, u_n, u_n, \ldots \rangle$, gives an *aborted* trace, denoted $\langle u_0, \ldots, u_n \rangle_\perp$. The set of all traces of an action system $\mathcal{A}$ is written as $tr(\mathcal{A})$.

We define an *approximation relation* on traces as follows: t approximates t', denoted $t \preceq t'$, if one of the following conditions hold:

(a) t is aborted and t is a prefix of t', or

(b) $t = t'$.

For action systems $\mathcal{A}$ and $\mathcal{C}$, the trace refinement is then defined as follows:

$$\mathcal{A} \sqsubseteq \mathcal{C} \quad \hat{=} \quad (\forall \tau \in tr(\mathcal{C}) \cdot \exists \sigma \in tr(\mathcal{A}) \cdot \sigma \preceq \tau)$$

The system $\mathcal{A}$ is usually referred to as the *abstract* and $\mathcal{C}$ as the *concrete* action system.

Local data refinement Data refinement is a technique for proving trace refinement of action systems. Let S be a statement on program variables a and u, and let S' be a statement on variables c and u. Assume that $R(a, c)$ is a relation between the variables a and c. Since $R(a, c)$ does not refer to the global variables u, it is a *local* relation. Then the abstract statement S is *data-refined* by the concrete statement S' using the abstraction relation $R(a, c)$, denoted $S \leq_R S'$, if for any predicate q

$$R \wedge \mathrm{wp}(S, q) \Rightarrow \mathrm{wp}(S', \exists a \cdot R \wedge q).$$

Note that $(\exists a \cdot R \wedge q)$ is a predicate on the program variables c and u.

From the action definition, we have that an action A is data-refined by an action A' using the relation $R(a, c)$, denoted $A \leq_R A'$, if the following two conditions hold for any predicate q:

(i) Guard: $R \wedge gA' \Rightarrow gA$,

(ii) Body: $R \wedge gA' \wedge \mathrm{wp}(sA, q) \Rightarrow \mathrm{wp}(sA', \exists a \cdot R \wedge q)$.

Refinement of action systems Consider action systems

$$\mathcal{A} :: \quad |[\ \mathbf{var}\ a \cdot a := a_0;\ \ \mathbf{do}\ A_1\ [\!]\ \ \ldots\ [\!]\ \ A_n\ \mathbf{od}\ ]|: u$$

$$\mathcal{C} :: \quad |[\ \mathbf{var}\ c \cdot c := c_0;\ \ \mathbf{do}\ C_1\ [\!]\ \ \ldots\ [\!]\ \ C_n\ [\!]\ \ X_1\ [\!]\ \ \ldots\ [\!]\ \ X_m\ \mathbf{od}\ ]|: u$$

where u is a list of global variables. The local variables a in $\mathcal{A}$ are replaced with new local variables c in $\mathcal{C}$. The actions A_i are replaced with C_i, and auxiliary actions X_j are added into $\mathcal{C}$. 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 $\mathcal{C}$ within any environment $\mathcal{E}$, if there exists an abstraction relation $R(a, c)$ such that:*

(i) Initialization*:* $R(a_0, c_0)$

(ii) Main actions*:* $A_i \leq_R C_i$*, for* $i = 1, \ldots, n$

(iii) Auxiliary actions*:* $\mathit{skip} \leq_R X_j$*, for* $j = 1, \ldots, m$

(iv) Continuation condition*:* $R \wedge g\mathcal{A} \Rightarrow g\mathcal{C}$

(v) Internal convergence*:* $R \Rightarrow \mathrm{wp}(\mathbf{do}\ X_1\ [\!]\ \ldots\ [\!]\ X_m\ \mathbf{od}, \mathit{true})$.

The lemma follows from the fact that the abstraction relation R is local, and from THEOREM 2 for trace refinement of action systems given by Back and von Wright [5]. Its first condition says that the abstraction relation is established by the initializations. The second condition requires that each action A_i is data-refined by the corresponding action C_i using $R(a, c)$. The third condition states that the auxiliary actions X_j behave like skip with respect to the global variables u while preserving $R(a, c)$. The fourth condition requires that an action in $\mathcal{C}$ is enabled whenever an action in $\mathcal{A}$ is enabled and $R(a, c)$ holds. The last condition stipulates that the execution of the auxiliary actions taken separately cannot continue forever whenever $R(a, c)$ holds.

3 Initial derivation

The design process starts from a high-level action system description of the basic functionality of the target circuit. The initial description is then decomposed within the Refinement Calculus framework into a number of action systems which can be viewed as *master-server* pairs. At this stage, shared variables are used to model signalling between circuit components. A master system is the *active* party — it sends a signal, whereas a server is the *passive* party — it receives a signal. Each master can be a server of some other system, and, respectively, each server system can be a master of some other system at the same time.

We have shown [20, 3] how arbitrary action systems are decomposed into a parallel composition of action systems, master-server pairs, where each component system models some functional entity of the target circuit. In this section we only give a generic pattern of such decomposition. Each component is then stepwise transformed into a (Q)DI system as shown in later sections.

Decomposition Let us consider a generic case of the system decomposition into a master-server pair. Let $\mathcal{A}$ be an action system of the form

$$\mathcal{A} :: \;\; |[\; \mathbf{do}\; g \to S_1; S_2 \;\; \mathbf{od} \;]| : u$$

where S_1 and S_2 are program statements, and S_1 does not assign to global variables. The system $\mathcal{A}$ can be decomposed into a master, $\mathcal{M}$, and server, $\mathcal{S}$, pair as follows:

$$\mathcal{M} :: \;\; |[\; \mathbf{chan}\; c \cdot c{\downarrow}; \mathbf{do}\; \neg c \to S_1; c{\uparrow} \;\; \mathbf{od} \;]| : u_1$$

$$\mathcal{S} :: \;\; |[\; \mathbf{do}\; c \to S_2; c{\downarrow} \;\; \mathbf{od} \;]| : c, u_2$$

where the fresh variable c, a channel variable, models communication between the systems. From now on we will use a special keyword, **chan**, to declare such variables and to denote the fact that they are exported. The variable c being hidden, the above master-server pair is a refinement of the original system:

$$\mathcal{A} \sqsubseteq |[\; \mathbf{var}\; c \bullet \mathcal{M} \parallel \mathcal{S} \;]|$$

Note that the decompositions are performed so that at most one channel variable is assigned in each action body.

The initial derivation includes also the separation of the *control systems* (*control path*) from the related *data processing systems* (*data path*). A data (processing) system is a program that receives a data word from some other data system, performs some computation with the received data and finally sends out the result of this computation. Control systems, in turn, do not manipulate data, their role is to master the progress of computation. The technical details how the control and data path are separated from each other depend on the data path model we want to implement [11].

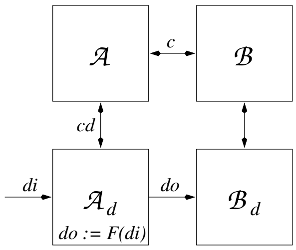

Figure 1: Communication channels

In this paper, we focus on generic communication between a control and either a control or a data system. Note, however, that the proposed method applies equally well to data paths to be implemented as (Q)DI circuits (dual-rail data). In the case of design using single-rail data [11], the data path is no longer a (Q)DI circuit, therefore, other techniques should be applied.

Communication channels Consider the following control system:

$$\begin{array}{ll} \mathcal{A} :: & |[\ \textbf{chan}\ \ldots\ cd \cdot \ldots\ cd{\downarrow}; \\ & \quad \textbf{do}\ \ldots\ g_i \rightarrow S_i[cd{\uparrow}];\ \ \ldots\ g_k[\neg cd] \rightarrow S_k;\ \ \ldots\ \textbf{od} \\ &]| : u \end{array}$$

The notation $S_i[cd{\uparrow}]$ means that the body S_i includes the assignment $cd{\uparrow}$, while $g_k[\neg cd]$ says that the guard g_k contains $\neg cd$. We assume that these are the only occurrences of the variable cd within the system $\mathcal{A}$. Note that such assumptions are made for all channel variables. Using the variable cd the system $\mathcal{A}$ controls the computation of a related data system

$$\mathcal{A}_d ::\ \ |[\ \textbf{var}\ do^* \cdot \textbf{do}\ cd \rightarrow do := F(di); cd{\downarrow}\ \ \textbf{od}\ ]|\ : cd, di$$

Here the input and the output data of the computation F is represented by di and do, respectively. The computation is initiated by the transition $cd{\uparrow}$ in $\mathcal{A}$, whereas its completion is acknowledged by the transition $cd{\downarrow}$ in $\mathcal{A}_d$. The variable cd is called a *control channel*. In principal, the computation F can be implemented either as a combinational circuit or a sequential circuit. In the former case, we say that F is a *combinational* computation, whereas in the latter case it is called a *sequential* computation.

Typically, the result do of the computation in $\mathcal{A}_d$ is used by another data system, $\mathcal{B}_d$, controlled by a control system $\mathcal{B}$ as it is shown in Fig. 1. The communication between systems $\mathcal{A}$ and $\mathcal{B}$ is modelled by means of a boolean variable, say c. The variables c and do are then said to form a *data channel*, written $c(do)$, between the systems $\mathcal{A}$ and $\mathcal{B}$. The states in which the data do is valid on the channel $c(do)$ are denoted by the predicate $vld(c, do)$. Depending on which of the parties is active in the communication, we can distinguish two cases:

(i) The system $\mathcal{A}$ is the active party and of the form

$$
\begin{array}{lll}
\mathcal{A} :: & |[& \textbf{chan} \ \dots \ c, cd \cdot \dots \ c{\downarrow}; cd{\downarrow}; \\
& & \textbf{do} \ \ \dots \ g_i \to S_i[cd{\uparrow}]; \ \ \dots \ g_j[\neg cd] \to S_j; \\
& & \quad\ \ \dots \ g_k \to S_k[c{\uparrow}]; \ \ \dots \ g_l[\neg c] \to S_l; \ \ \dots \\
& & \textbf{od} \\
&]|: u &
\end{array}
$$

The data channel $c(do)$ is then called a *push* channel. The data do on the channel is valid in the states where c is true:

$$vld(c, do) \quad \hat{=} \quad c$$

Note that the predicate $c \Rightarrow \neg cd$ is an invariant of the system $\mathcal{A}$.

(ii) The system $\mathcal{A}$ is the passive party (i.e. $\mathcal{B}$ is active) and of the form

$$
\begin{array}{lll}
\mathcal{A} :: & |[& \textbf{chan} \ \dots \ cd \cdot \dots \ cd{\downarrow}; \\
& & \textbf{do} \ \ \dots \ g_i \to S_i[cd{\uparrow}]; \ \ \dots \ g_j[\neg cd] \to S_j; \\
& & \quad\ \ \dots \ g_k \to S_k[c{\downarrow}]; \ \ \dots \ g_l[c] \to S_l; \ \ \dots \\
& & \textbf{od} \\
&]|: c, u &
\end{array}
$$

In this case, the data channel $c(do)$ is called a *pull* channel. The data do on the channel is valid in the states where c is false:

$$vld(c, do) \quad \hat{=} \quad \neg c$$

The predicate $\neg c \Rightarrow \neg cd$ is an invariant of the system $\mathcal{A}$.

Next, we consider generic control systems. Such a system is responsible for some computation, therefore, it has *input* and *output* channels. Assume that c_1 is an input channel, while c_2 is an output channel which depends on c_1. If the computation of the data for c_2 is combinational, the control system must ensure that the input data is not corrupted before the output produced by the computation is no longer needed. Formally, this means that a control system must preserve the following invariant, called the *synchronization constraint*:

$$vld(c_2, do) \Rightarrow vld(c_1, di)$$

Since data variables are shared by several data systems, the input variable of one system is the output variable for another system. Therefore, the validity condition $vld(c_1, di)$ is defined with respect to the system which controls the computation of di. The actual form of a control system depends on the type of its communication channels. Generic systems for different combinations of communication channel types are given below:

(i) Let both the input and the output channel be push channels. The generic form of such a system is as follows:

$$
\begin{array}{ll}
\mathcal{A} :: & |[\ \textbf{chan}\ \ldots\ c_2, cd \cdot \ldots\ c_2\!\downarrow; cd\!\downarrow; \\
& \quad \textbf{do}\ \ \ldots\ g_i[c_1] \to S_i[cd\!\uparrow];\ \ \ldots\ g_j[\neg cd] \to S_j; \\
& \quad \quad \ldots\ g_k \to S_k[c_2\!\uparrow];\ \ \ldots\ g_l[\neg c_2] \to S_l;\ \ \ldots \\
& \quad \textbf{od} \\
&]|: c_1, u
\end{array}
$$

The system additionally contains the transition $c_1\!\downarrow$, which is not indicated above, since its exact place in the system depends on the computation type. For a combinational computation, the synchronization constraint requires $c_1\!\downarrow$ to be executed after $c_2\!\downarrow$. In the case of sequential computation, it can precede $c_2\!\downarrow$ as well.

(ii) Let next c_1 and c_2 be, respectively, push and pull channels:

$$
\begin{array}{ll}
\mathcal{A} :: & |[\ \textbf{chan}\ \ldots\ cd \cdot \ldots\ cd\!\downarrow; \\
& \quad \textbf{do}\ \ \ldots\ g_i[c_1] \to S_i[cd\!\uparrow];\ \ \ldots\ g_j[\neg cd] \to S_j; \\
& \quad \quad \ldots\ g_k \to S_k[c_2\!\downarrow];\ \ \ldots\ g_l[c_2] \to S_l;\ \ \ldots \\
& \quad \textbf{od} \\
&]|: c_1, c_2, u
\end{array}
$$

Again, the system contains the transition $c_1\!\downarrow$. For a combinational computation, it must occurr after the transition $c_2\!\uparrow$ due to the synchronization constraint. Whereas in the case of sequential computation, it is allowed to be executed at any time after the transition $cd\!\downarrow$.

(iii) Let c_1 and c_2 be, respectively, pull and push channels:

$$
\begin{array}{ll}
\mathcal{A} :: & |[\ \textbf{chan}\ \ldots\ c_1, c_2, cd \cdot \ldots\ c_1\!\uparrow; c_2\!\downarrow; cd\!\downarrow; \\
& \quad \textbf{do}\ \ \ldots\ g_i \to S_i[c_1\!\uparrow];\ \ \ldots\ g_j[\neg c_1] \to S_j[cd\!\uparrow];\ \ \ldots\ g_k[\neg cd] \to S_k; \\
& \quad \quad \ldots\ g_l \to S_l[c_2\!\uparrow];\ \ \ldots\ g_m[\neg c_2] \to S_m;\ \ \ldots \\
& \quad \textbf{od} \\
&]|: u
\end{array}
$$

In this case the only sequencing of the transitions $c_1\!\uparrow$ and $c_2\!\uparrow$ is possible.

(iv) Let finally both c_1 and c_2 be pull channels:

$$
\begin{array}{ll}
\mathcal{A} :: & |[\ \textbf{chan}\ \ldots\ c_1, cd \cdot \ldots\ c_1\!\uparrow; cd\!\downarrow; \\
& \quad \textbf{do}\ \ \ldots\ g_i \to S_i[c_1\!\uparrow];\ \ \ldots\ g_j[\neg c_1] \to S_j[cd\!\uparrow];\ \ \ldots\ g_k[\neg cd] \to S_k; \\
& \quad \quad \ldots\ g_l \to S_l[c_2\!\downarrow];\ \ \ldots\ g_m[c_2] \to S_m;\ \ \ldots \\
& \quad \textbf{od} \\
&]|: c_2, u
\end{array}
$$

Again there is the only way to sequence the transitions $c_1\!\uparrow$ and $c_2\!\downarrow$.

4 Handshaking expansion

Since we are developing (Q)DI systems, which are not allowed to write onto the same variables, each communication variable introduced in the previous design step is to be implemented by two separate variables. A *request* variable is declared and assigned within the master system and an *acknowledgment* variable within the server, respectively. The idea is to transform the abstract one-variable signalling stepwise into *four-phase handshaking* which is well-suited for hardware implementation. Potentially very efficient *two-phase handshaking*, where only one half of the four phases (up-going or down-going part) is used at the time, is also a possibility, but it requires somewhat more complex circuitry than the four-phase model. In this paper, we use four-phase handshaking.

Handshaking cycles In one-variable signalling, the communication cycle is a sequence of transitions $c\uparrow; c\downarrow$ which starts in the initial state $c = \mathit{false}$ in the case of a push channel. Similarly, the communication cycle for a pull channel is a sequence of transitions $c\downarrow; c\uparrow$ which starts from $c = \mathit{true}$. The handshaking expansion replaces the communication variable c with two handshaking variables, req and ack. The one-variable communication cycle is replaced by the following sequence of transitions, called the *handshaking cycle*, on the variables req and ack:

$$req\uparrow;\ ack\uparrow;\ req\downarrow;\ ack\downarrow\ .$$

The relationship between the variables c, req and ack depends on the type of the channel under consideration. Since there are four transitions in the handshaking cycle, we can choose which of the transitions $ack\uparrow$ and $ack\downarrow$ actually stands for the acknowledgment. Thus for each type of data channels, we have two data validity models on the handshaking variable level. The corresponding *handshaking relations* are given below:

(i) On a *narrow push* channel $c(do)$, the data do is valid in the states $req \wedge \neg ack$. Thus, the handshaking relation is defined as follows:

$$H1(c, req, ack) \quad \hat{=} \quad c = req \wedge \neg ack$$

(ii) On a *broad push* channel $c(do)$, the data do is valid in the states $req \vee ack$. We then have the following handshaking relation:

$$H2(c, req, ack) \quad \hat{=} \quad c = req \vee ack$$

(iii) On a *narrow pull* channel $c(do)$, the data do is valid in the states $req \wedge ack$. This gives the following handshaking relation:

$$H3(c, req, ack) \quad \hat{=} \quad \neg c = req \wedge ack$$

(iv) On a *broad pull* channel $c(do)$, the data do is valid in the states $\neg req \vee ack$. The handshaking relation is then defined as follows:

$$H4(c, req, ack) \quad \hat{=} \quad \neg c = \neg req \vee ack$$

For control channels, the handshaking relation is identical to the relation for narrow push channels ($H1$). As we show below, the handshaking relations constitute an essential part of the abstraction relation which is used to prove the handshaking expansion as a refinement of action systems.

Refinement Consider the following parallel composition of action systems

$$\mathcal{C} :: \quad |[\text{ } \mathbf{var} \; c_1, \ldots, c_m \bullet \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n \text{ }]|$$

where $c_j(d_j)$ is a channel between some systems $\mathcal{A}$ and $\mathcal{B}$ from $\mathcal{C}$. In the handshaking expansion, each variable c_j is replaced with two handshaking variables, req_j and ack_j. This gives a new composition of action systems:

$$\mathcal{C}' :: \quad |[\text{ } \mathbf{var} \; req_1, \ldots, ack_m \bullet \mathcal{A}'_1 \parallel \ldots \parallel \mathcal{A}'_n \text{ }]|$$

Below we show the generic forms of the new systems $\mathcal{A}'_1, \ldots, \mathcal{A}'_n$. We also define an abstraction relation, R, and an invariant, Inv, of the system $\mathcal{C}'$ such that $\mathcal{C}'$ is a trace refinement of the original system $\mathcal{C}$, proved using the abstraction relation $R \wedge Inv$:

$$\mathcal{C} \sqsubseteq \mathcal{C}'$$

The predicates R and Inv are written as the conjunctions,

$$\begin{array}{rcl} R & \hat{=} & (R_1 \wedge \ldots \wedge R_m) \wedge PC \\ Inv & \hat{=} & Inv_1 \wedge \ldots \wedge Inv_m \end{array}$$

where each pair R_j, Inv_j, to be defined below, relates to the corresponding channel $c_j(d_j)$. Note that the relation PC associates the program counter values in the abstract programs with those of the concrete programs for all actions whose bodies remain unaffected by the handshaking expansion. Let us assume that A_i^k is such an action from the system $\mathcal{A}_k$. Let us assume further that the corresponding main action in the refined system $\mathcal{A}'_k$ has the index i'. Then the abstract program counter value i must be associated with the concrete program counter value i':

$$(pc_{\mathcal{A}_k} = i) \Leftrightarrow (pc_{\mathcal{A}'_k} = i')$$

The abstraction relation PC is then defined to be the conjunction of the above equivalences for all unaffected actions A_i^k in all the systems $\mathcal{A}_k$.

Let us now consider the refinements and the corresponding parts of the abstraction relation for each type of communication channels:

(i) Assume that c_j is a push channel. This means that the aforementioned systems $\mathcal{A}$ and $\mathcal{B}$ are of the form:

$$
\begin{array}{ll}
\mathcal{A} :: & |[\ \textbf{chan}\ \ldots\ c_j \cdot \ldots\ c_j\!\downarrow; \\
& \quad \textbf{do}\ \ldots\ g_{l_1} \to S_{l_1}[c_j\!\uparrow];\ \ldots\ g_{l_2}[\neg c_j] \to S_{l_2};\ \ldots\ \textbf{od} \\
&]|: u_A
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{B} :: & |[\ \textbf{chan}\ \ldots \cdot \ldots \\
& \quad \textbf{do}\ \ldots\ h_{l_5}[c_j] \to T_{l_5};\ \ldots\ h_{l_6} \to T_{l_6}[c_j\!\downarrow];\ \ldots\ \textbf{od} \\
&]|: c_j, u_B
\end{array}
$$

Note that the actions have indexes which are the same as the values assigned to the program counter by the actions. The handshaking expansion gives a refined system, $\mathcal{B}'$, of the form:

$$
\begin{array}{ll}
\mathcal{B}' :: & |[\ \textbf{var}\ \ldots\ ack_j \cdot \ldots\ ack_j\!\downarrow; \\
& \quad \textbf{do}\ \ \ldots\ h_{l'_5}[req_j] \to T_{l'_5};\ \ldots\ h_{l'_6} \to T_{l'_6}[ack_j\!\uparrow]; \\
& \qquad\quad \ldots\ \neg req_j \to skip;\ \ldots\ true \to ack_j\!\downarrow;\ \ldots \\
& \quad \textbf{od} \\
&]|: req_j, u_B
\end{array}
$$

For the system $\mathcal{A}$ we have two cases. If $c_j(d_j)$ is a narrow channel, then the acknowledgment in the refined system starts with the transition $ack_j\!\uparrow$. The refined system $\mathcal{A}'$ is then as follows:

$$
\begin{array}{ll}
\mathcal{A}' :: & |[\ \textbf{var}\ \ldots\ req_j \cdot \ldots\ req_j\!\downarrow; \\
& \quad \textbf{do}\ \ \ldots\ g_{l'_1} \to S_{l'_1}[req_j\!\uparrow];\ \ldots\ g_{l'_2}[ack_j] \to S_{l'_2}; \\
& \qquad\quad \ldots\ true \to req_j\!\downarrow;\ \ldots\ \neg ack_j \to skip;\ \ldots \\
& \quad \textbf{od} \\
&]|: ack_j, u_A
\end{array}
$$

The corresponding conjunct R_j in the abstraction relation R includes the handshaking relation $H1(c_j, req_j, ack_j)$, and an abstraction relation, PC_j, for program counters:

$$R_j \ \ \hat{=}\ \ H1(c_j, req_j, ack_j) \wedge PC_j$$

The relation PC_j is needed for technical reasons in the refinement proofs. It guarantees that the values of the program counters in the original and refined systems reflect the main relation $H1$. In this case, PC_j is defined as follows:

$$PC_j \ \ \hat{=}\ \ \begin{array}{l} (pc_A \in at_{\mathcal{A}}(c)) \Leftrightarrow (pc_{A'} \in at_{\mathcal{A}'}(req_j \wedge \neg ack_j)) \\ \wedge\ (pc_B \in at_{\mathcal{B}}(\neg c)) \Leftrightarrow (pc_{B'} \in at_{\mathcal{B}'}(\neg req_j \vee ack_j)) \end{array} \tag{1}$$

The corresponding conjunct Inv_j in the concrete invariant consists of two parts:

$$Inv_j \ \ \hat{=}\ \ I1(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \wedge I_{push}(pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$$

The predicate $I1(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$ reflects the sequence of transitions in the handshaking cycle. It consists of local invariants of the systems $\mathcal{A}'$, $\mathcal{B}'$, and is defined as follows:

$$I1(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \quad \hat{=} \quad \begin{array}{l} (req_j \Rightarrow pc_{A'} \in at_{\mathcal{A}'}(req_j)) \\ \wedge\ (ack_j \Rightarrow pc_{A'} \in at_{\mathcal{A}'}(ack_j)) \\ \wedge\ (req_j \Rightarrow pc_{B'} \in at_{\mathcal{B}'}(req_j)) \\ \wedge\ (ack_j \Rightarrow pc_{B'} \in at_{\mathcal{B}'}(ack_j)) \end{array} \tag{2}$$

The second predicate $I_{push}(pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$ is again needed for technical reasons in the refinement proofs. It is the same for both narrow and broad push channels and relates the program counters $pc_{A'}$ of $\mathcal{A}'$ and $pc_{B'}$ of $\mathcal{B}'$:

$$I_{push}(pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \quad \hat{=} \quad \begin{array}{l} (pc_{A'} \in at_{\mathcal{A}'}(\neg req_j \wedge ack_j)) \\ \Rightarrow (pc_{B'} \in at_{\mathcal{B}'}(\neg req_j \wedge ack_j)) \end{array} \tag{3}$$

If the channel $c_j(d_j)$ is broad, then the acknowledgment $c_j\downarrow$ is identified with the transition $ack_j\downarrow$. The refined system $\mathcal{A}'$ is then of the form:

$$\begin{array}{ll} \mathcal{A}' :: & |[\ \textbf{var}\ \ldots\ req_j \cdot \ldots\ req_j\downarrow; \\ & \quad \textbf{do}\ \ldots\ g_{l'_1} \to S_{l'_1}[req_j\uparrow];\ \ldots\ true \to req_j\downarrow; \\ & \quad\quad \ldots\ \neg ack_j \to skip;\ \ldots\ g_{l'_2}[\neg ack_j] \to S_{l'_2};\ \ldots \\ & \quad \textbf{od} \\ &]|: ack_j, u_A \end{array}$$

The conjunct R_j in the abstraction relation includes the handshaking relation $H2(c_j, req_j, ack_j)$:

$$R_j \quad \hat{=} \quad H2(c_j, req_j, ack_j) \wedge PC_j$$

Here PC_j is defined as bellow:

$$PC_j \quad \hat{=} \quad \begin{array}{l} (pc_A \in at_{\mathcal{A}}(c)) \Leftrightarrow (pc_{A'} \in at_{\mathcal{A}'}(req_j \vee ack_j)) \\ \wedge\ (pc_B \in at_{\mathcal{B}}(\neg c)) \Leftrightarrow (pc_{B'} \in at_{\mathcal{B}'}(\neg req_j \wedge \neg ack_j)) \end{array} \tag{4}$$

The corresponding predicate Inv_j is the following:

$$Inv_j \quad \hat{=} \quad I2(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \wedge I_{push}(pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$$

where the invariant $I2(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$ is defined as bellow:

$$I2(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \quad \hat{=} \quad \begin{array}{l} (req_j \Rightarrow pc_{A'} \in at_{\mathcal{A}'}(req_j)) \\ \wedge\ (\neg ack_j \Rightarrow pc_{A'} \in at_{\mathcal{A}'}(\neg ack_j)) \\ \wedge\ (req_j \Rightarrow pc_{B'} \in at_{\mathcal{B}'}(req_j)) \\ \wedge\ (\neg ack_j \Rightarrow pc_{B'} \in at_{\mathcal{B}'}(\neg ack_j)) \end{array} \tag{5}$$

(ii) Next, assume that $c_j(d_j)$ is a pull channel. Then the systems $\mathcal{A}$ and $\mathcal{B}$ are of the form:

$$
\begin{array}{ll}
\mathcal{A} :: & |[\ \textbf{chan}\ \ldots\ \cdot \ldots \\
& \quad \textbf{do}\ \ldots\ g_{l_1} \rightarrow S_{l_1}[c_j\!\downarrow];\ \ \ldots\ g_{l_2}[c_j] \rightarrow S_{l_2};\ \ \ldots\ \textbf{od} \\
&]|: c_j, u_A
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{B} :: & |[\ \textbf{chan}\ \ldots\ c_j \cdot \ldots\ c_j\!\uparrow; \\
& \quad \textbf{do}\ \ldots\ h_{l_5} \rightarrow T_{l_5}[c_j\!\uparrow];\ \ \ldots\ h_{l_6}[\neg c_j] \rightarrow T_{l_6};\ \ \ldots\ \textbf{od} \\
&]|: u_B
\end{array}
$$

After the handshaking expansion, we get the following system $\mathcal{B}'$:

$$
\begin{array}{ll}
\mathcal{B}' :: & |[\ \textbf{var}\ \ldots\ req_j \cdot \ldots\ req_j\!\uparrow; \\
& \quad \textbf{do}\ \ \ldots\ h_{l'_5} \rightarrow T_{l'_5}[req_j\!\uparrow];\ \ \ldots\ h_{l'_6}[ack_j] \rightarrow T_{l'_6}; \\
& \qquad\quad \ldots\ true \rightarrow req_j\!\downarrow;\ \ \ldots\ \neg ack_j \rightarrow skip;\ \ \ldots \\
& \quad \textbf{od} \\
&]|: ack_j, u_B
\end{array}
$$

Again, we consider two possible refinements of the system $\mathcal{A}$. If c_j is a narrow channel, then the request for the refined system $\mathcal{A}'$ starts with the transition $req_j\!\downarrow$:

$$
\begin{array}{ll}
\mathcal{A}' :: & |[\ \textbf{var}\ \ldots\ ack_j \cdot \ldots\ ack_j\!\downarrow; \\
& \quad \textbf{do}\ \ \ldots\ req_j \rightarrow skip;\ \ \ldots\ g_{l'_1} \rightarrow S_{l'_1}[ack_j\!\uparrow]; \\
& \qquad\quad \ldots\ g_{l'_2}[\neg req_j] \rightarrow S_{l'_2};\ \ \ldots\ true \rightarrow ack_j\!\downarrow;\ \ \ldots \\
& \quad \textbf{od} \\
&]|: req_j, u_A
\end{array}
$$

The corresponding conjunct in the abstraction relation R is defined as bellow:

$$
R_j \ \ \hat{=}\ \ H3(c_j, req_j, ack_j) \wedge PC_j
$$

where PC_j is the following:

$$
PC_j \ \ \hat{=}\ \ \begin{array}{l}
(pc_A \in at_{\mathcal{A}}(\neg c)) \Leftrightarrow (pc_{A'} \in at_{\mathcal{A}'}(req_j \wedge ack_j)) \\
\wedge\ (pc_B \in at_{\mathcal{B}}(c)) \Leftrightarrow (pc_{B'} \in at_{\mathcal{B}'}(\neg req_j \vee \neg ack_j))
\end{array}
$$

As before, the corresponding predicate Inv_j in the concrete invariant Inv consists of two parts:

$$
Inv_j \ \ \hat{=}\ \ I3(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \wedge I_{pull}(pc_{\mathcal{A}'}, pc_{\mathcal{B}'})
$$

The invariant $I3(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$ is defined as follows:

$$
I3(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \ \ \hat{=}\ \ \begin{array}{l}
(\neg req_j \Rightarrow pc_{A'} \in at_{\mathcal{A}'}(\neg req_j)) \\
\wedge\ (ack_j \Rightarrow pc_{A'} \in at_{\mathcal{A}'}(ack_j)) \\
\wedge\ (\neg req_j \Rightarrow pc_{B'} \in at_{\mathcal{B}'}(\neg req_j)) \\
\wedge\ (ack_j \Rightarrow pc_{B'} \in at_{\mathcal{B}'}(ack_j))
\end{array}
$$

Like I_{push}, the predicate $I_{pull}(pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$ is the same for both narrow and broad pull channels:

$$I_{pull}(pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \;\hat{=}\; \begin{array}{l}(pc_{B'} \in at_{\mathcal{B}'}(\neg req_j \wedge ack_j)) \\ \Rightarrow (pc_{A'} \in at_{\mathcal{A}'}(\neg req_j \wedge ack_j))\end{array}$$

For a broad channel, the request $c_j\uparrow$ is identified with the transition $req_j\uparrow$ in the system $\mathcal{A}'$:

$$\begin{array}{ll}\mathcal{A}' :: & |[\ \textbf{var}\ \ldots\ ack_j \cdot \ldots\ ack_j\downarrow; \\ & \quad \textbf{do}\ \ \ldots\ g_{l'_1} \rightarrow S_{l'_1}[ack_j\uparrow];\ \ \ldots\ g_{l'_2}[\neg req_j] \rightarrow skip; \\ & \quad\quad \ldots\ true \rightarrow ack_j\downarrow\,;\ \ \ldots\ req_j \rightarrow S_{l'_2};\ \ \ldots \\ & \quad \textbf{od} \\ &]|: req_j, u_A\end{array}$$

The corresponding conjunct in the abstraction relation is given below:

$$R_j \;\hat{=}\; H4(c_j, req_j, ack_j) \wedge PC_j$$

where PC_j is follows:

$$PC_j \;\hat{=}\; \begin{array}{l}(pc_A \in at_{\mathcal{A}}(\neg c)) \Leftrightarrow (pc_{A'} \in at_{\mathcal{A}'}(req_j \wedge ack_j)) \\ \wedge\ (pc_B \in at_{\mathcal{B}}(c)) \Leftrightarrow (pc_{B'} \in at_{\mathcal{B}'}(\neg req_j \vee \neg ack_j))\end{array}$$

In this case, we use the following predicate Inv_j in the concrete invariant:

$$Inv_j \;\hat{=}\; I4(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \wedge I_{pull}(pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$$

Here, the invariant $I4(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'})$ is defined as follows:

$$I4(req_c, ack_c, pc_{\mathcal{A}'}, pc_{\mathcal{B}'}) \;\hat{=}\; \begin{array}{l}(req_j \Rightarrow pc_{A'} \in at_{\mathcal{A}'}(req_j)) \\ \wedge\ (ack_j \Rightarrow pc_{A'} \in at_{\mathcal{A}'}(ack_j)) \\ \wedge\ (req_j \Rightarrow pc_{B'} \in at_{\mathcal{B}'}(req_j)) \\ \wedge\ (ack_j \Rightarrow pc_{B'} \in at_{\mathcal{B}'}(ack_j))\end{array}$$

In this paper, we omit the verification that the abstraction relation R proves the trace refinement. The related proofs for handshaking expansion can be found in the earlier paper [21].

EXAMPLE 1 Let us consider a two-channel handshake distributor shown in Fig. 2. The distributor is activated via a left channel. Then it sequences the communication cycles on the middle and right channels starting from the middle. We assume that the middle channel m controls a combinational computation, $do := F(di)$, implemented in the data part of the circuit. The left and right channels, $l(di)$ and $r(do)$, are both push channels, narrow and broad, respectively. On the one-variable communication level the distributor is specified as the following action system:

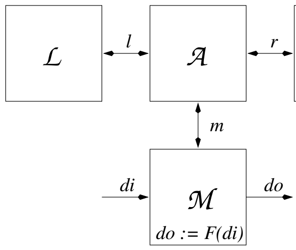

Figure 2: Two-channel handshake distributor

$$\begin{array}{rl} \mathcal{A} :: & |[\ \textbf{chan}\ m, r \cdot m{\downarrow}; r{\downarrow}; \\ & \quad \textbf{do}\ l \to m{\uparrow}\ ;\ \neg m \to r{\uparrow}\ ;\ \neg r \to l{\downarrow}\ \ \textbf{od} \\ &]|: l \end{array}$$

The environment of the system can be viewed as two action systems, one for the communication on each data channel:

$$\mathcal{L} :: \quad |[\ \textbf{chan}\ l \cdot l{\downarrow}; \textbf{do}\ g \to l{\uparrow}\ ;\ \neg l \to S\ \textbf{od}\ ]|$$

$$\mathcal{R} :: \quad |[\ \textbf{do}\ r \to T;\ h \to r{\downarrow}\ \ \textbf{od}\ ]|\ :r$$

The control part of the system is then the following parallel composition:

$$\mathcal{S} :: \quad |[\ \textbf{var}\ l, r \bullet \mathcal{L} \parallel \mathcal{A} \parallel \mathcal{R}\ ]|$$

In the handshaking expansion, we introduce handshaking cycles on the data channels $l(di)$ and $r(do)$. The control channel m is left unexpanded:

$$\begin{array}{rl} \mathcal{A}' :: & |[\ \textbf{chan}\ m\ \textbf{var}\ ack_l, req_r \cdot m{\downarrow}; ack_l{\downarrow}; req_r{\downarrow}; \\ & \quad \textbf{do}\ \ req_l \to m{\uparrow} \\ & \quad\ \ \textbf{;}\ \neg m \to req_r{\uparrow} \\ & \quad\ \ \textbf{;}\ ack_r \to req_r{\downarrow} \\ & \quad\ \ \textbf{;}\ \neg ack_r \to ack_l{\uparrow} \\ & \quad\ \ \textbf{;}\ \neg req_l \to ack_l{\downarrow} \\ & \quad \textbf{od} \\ &]|: req_l, ack_r \end{array}$$

The refined left and right environment systems are as follows:

$$\begin{array}{rl} \mathcal{L}' :: & |[\ \textbf{var}\ req_l \cdot req_r{\downarrow}; \\ & \quad \textbf{do}\ g \wedge \neg ack_l \to req_l{\uparrow}\ ;\ ack_l \to S; req_l{\downarrow}\ \ \textbf{od} \\ &]|: ack_l \end{array}$$

$$\begin{array}{rl} \mathcal{R}' :: & |[\ \textbf{var}\ ack_r \cdot ack_r{\downarrow} \\ & \quad \textbf{do}\ req_r \to T; ack_r{\uparrow}\ ;\ h \wedge \neg req_r \to ack_r{\downarrow}\ \ \textbf{od} \\ &]|: req_r \end{array}$$

They basically specify the order of transitions on the handshaking variables which must be preserved by any potential environment. Note that to keep the presentation simpler we did not precisely follow the method described earlier. Namely, no separate actions related to the transitions $req_l \downarrow$ and $ack_r \uparrow$ were added to the environment systems. Since the names pc_L and $pc_{L'}$ (pc_R and $pc_{R'}$) in such case stand for the same variable, we do not need an abstraction relation for them.

The handshaking relation $H1(l, req_l, ack_l)$ is used for the left channel, while the relation $H2(r, req_r, ack_r)$ is used for the right channel. The predicates PC_l, PC_r are instances of (1), (4):

$$
\begin{aligned}
PC_l &\;\hat{=}\; (pc_A = 0) \Leftrightarrow (pc_{A'} = 4 \vee pc_{A'} = 0) \\
PC_r &\;\hat{=}\; (pc_A = 2) \Leftrightarrow (pc_{A'} = 2 \vee pc_{A'} = 3)
\end{aligned}
$$

Since only the body of the first action in the system $\mathcal{A}$ is unaffected by the handshaking expansion, the abstraction relation PC is simply:

$$PC \;\hat{=}\; (pc_A = 1) \Leftrightarrow (pc_{A'} = 1)$$

We have then the following abstraction relation:

$$R \;\hat{=}\; (H1(l, req_l, ack_l) \wedge PC_l) \wedge (H2(r, req_r, ack_r) \wedge PC_r) \wedge PC$$

The concrete invariants related to the channels $l(di)$, $r(do)$ are defined by (2) and (5), respectively:

$$
\begin{aligned}
I1(req_l, ack_l, pc_{L'}, pc_{A'}) \;\hat{=}\; & (req_l \Rightarrow pc_{L'} = 1) \wedge (ack_l \Rightarrow pc_{L'} = 0) \\
& \wedge (req_l \Rightarrow pc_{A'} \in \{1, 2, 3, 4\}) \wedge (ack_l \Rightarrow pc_{A'} = 4) \\
I2(req_r, ack_r, pc_{A'}, pc_{R'}) \;\hat{=}\; & (req_r \Rightarrow pc_{A'} = 2) \wedge (\neg ack_r \Rightarrow pc_{A'} \in \{0, 1, 2, 4\}) \\
& \wedge (req_r \Rightarrow pc_{R'} = 1) \wedge (\neg ack_r \Rightarrow pc_{R'} = 0)
\end{aligned}
$$

Finally, the definition (3) gives the following invariants on the program counters:

$$
\begin{aligned}
I_{push}(pc_{L'}, pc_{A'}) &\;\hat{=}\; (pc_{L'} = 0) \Rightarrow (pc_{A'} = 4) \\
I_{push}(pc_{A'}, pc_{R'}) &\;\hat{=}\; (pc_{A'} = 2) \Rightarrow (pc_{R'} = 0)
\end{aligned}
$$

The complete invariant is then defined as follows:

$$
\begin{aligned}
Inv \;\hat{=}\; & I1(req_l, ack_l, pc_{L'}, pc_{A'}) \wedge I_{push}(pc_{L'}, pc_{A'}) \\
\wedge \; & I2(req_r, ack_r, pc_{A'}, pc_{R'}) \wedge I_{push}(pc_{A'}, pc_{R'})
\end{aligned}
$$

The handshaking expansion using the relation $R \wedge Inv$ is the refinement

$$\mathcal{S} \sqsubseteq \mathcal{S}'$$

where the refined system $\mathcal{S}'$ is the composition

$$\mathcal{S}' :: \; |[\; \mathbf{var}\; req_l, ack_l, req_r, ack_r \bullet \mathcal{L}' \parallel \mathcal{A}' \parallel \mathcal{R}' \;]|$$

5 Circuit extraction

In this section, we show how (quasi) delay-insensitive action systems are derived by refining handshake-expanded action systems described in the previous section.

The handshaking expansion produces systems where actions contain at most one transition on handshaking (global) variables. Transitions on local variables can be splitted into separate actions using the following lemma:

LEMMA 2 *Let $\mathcal{A}$ be a sequential action system of the form*

$$\begin{array}{ll} \mathcal{A} :: & |[\ \textbf{var}\ \ldots\ v^*, x \cdot J; \\ & \quad \textbf{do}\ \ldots\ (g_i \to S_i[v := V]; x := X); \ldots\ \ \textbf{od} \\ &]|: z \end{array}$$

Then $\mathcal{A} \sqsubseteq \mathcal{A}'$, where the refined sequential system $\mathcal{A}'$ is as follows:

$$\begin{array}{ll} \mathcal{A}' :: & |[\ \textbf{var}\ \ldots\ v^*, x \cdot J; \\ & \quad \textbf{do}\ \ldots\ (g_i \to S_i[v := V]); (\mathit{true} \to x := X); \ldots\ \ \textbf{od} \\ &]|: z \end{array}$$

The lemma allows to refine handshake-expanded action systems into systems of elementary pairs. Now we introduce a strategy for transforming such pairs into (Q)DI elements. The main objective here is to remove the explicit sequencing enforced by the program counter control which appears in the guards of the elementary actions as shown above. The basic idea is to find equivalent boolean expressions on circuit variables for conditions on the program counter. If such a replacement cannot be found, then the condition in question must be implemented using an auxiliary boolean variable. The result of the transformations will be a (Q)DI action system.

Stability properties A handshake-expanded action system, $\mathcal{A}$, is *up-stable* with respect to the boolean variable v, written $up(v)$, if the following total correctness assertion holds for any action A from $\mathcal{A}$:

$$(v = \mathit{true})\ \langle A \rangle\ (v = \mathit{true})$$

Intuitively this means that no action in $\mathcal{A}$ can change the value of v from true to false. Likewise, the system $\mathcal{A}$ is *down-stable* with respect to the variable v, written $down(v)$, if for any action A from $\mathcal{A}$ the following holds:

$$(v = \mathit{false})\ \langle A \rangle\ (v = \mathit{false})$$

Again, down-stability means that the variable v cannot be set to true in $\mathcal{A}$, if it was equal to false. An action system is *stable* with respect to the variable v, if it is both up-stable and down-stable:

$$\mathit{stable}(v) \quad \hat{=} \quad up(v) \wedge down(v)$$

Handshaking protocol The handshaking expansion described in the previous section guarantees that the corresponding system, say $\mathcal{E}$, of elementary pairs and its environment, another system of elementary pairs $\mathcal{P}$, obey the following order of transitions (the handshaking cycle) on each channel between them:

$$req\uparrow, ack\uparrow, req\downarrow, ack\downarrow$$

The handshaking cycle can be expressed as the following property $HP(req, ack)$:

$$HP(req, ack) \quad \hat{=} \quad HP_{ack}(req) \wedge HP_{req}(ack)$$

where $HP_{ack}(req)$ is a property of the passive party in the communication on a channel, while $HP_{req}(ack)$ is a property of the active party. These properties are formally defined as follows:

$$\begin{array}{lcl} HP_{ack}(req) & \hat{=} & stable(req) \wedge (\neg req \Rightarrow down(ack)) \wedge (req \Rightarrow up(ack)) \\ HP_{req}(ack) & \hat{=} & stable(ack) \wedge (\neg ack \Rightarrow up(req)) \wedge (ack \Rightarrow down(req)) \end{array}$$

Now assume that the pairs (req_j, ack_j), $j = 1, \ldots, m$ model the control signals on the channels between the systems $\mathcal{E}$ and $\mathcal{P}$. Then the handshaking protocol, denoted HP, between the systems is defined as:

$$HP \quad \hat{=} \quad \bigwedge_{j=1}^{m} HP(req_j, ack_j)$$

Extracting elements Now let us consider in isolation a sequential system of elementary pairs of the form

$$\mathcal{E} :: \ |[\ \mathbf{var}\ v \cdot J; \mathbf{do}\ g_{v_1} \rightarrow v_1\uparrow; \ldots \ \overline{g}_{v_n} \rightarrow v_n\downarrow \ \mathbf{od} \ ]| : z$$

where $v = v_1, \ldots, v_n$, while assuming that the system and its environment follow the handshaking protocol HP. Expressions involving program counters are replaced in a series of transformations. Each transformation either replaces the program counter expressions in the guards of an element or introduces a new element (an auxiliary variable) into the system. Let us consider in detail a single step in the series. Assume that at the k-th step the intermediate system, $\mathcal{E}^k$, is as follows:

$$\begin{array}{ll} \mathcal{E}^k :: & |[\ \mathbf{var} \ \ldots \ pc \cdot \ldots \ pc := 0; \\ & \quad \mathbf{do} \\ & \quad \ldots \ [\!] \ pc = i - 1 \wedge g_{v_k} \rightarrow v_k\uparrow; pc := i \\ & \quad \ldots \ [\!] \ pc = j - 1 \wedge \overline{g}_{v_k} \rightarrow v_k\downarrow; pc := j \\ & \quad \ldots \\ & \quad \mathbf{od} \\ &]| : z \end{array}$$

We are aiming to replace the program counter in the guards of the element related to the variable v_k, i.e., $(E_{v_k}, \overline{E}_{v_k})$, with suitable conditions on the other variables of the system. There are two possible cases:

(1) We are able to find two boolean expressions (conjunctions), b_1 and b_2, on the variables $(v \cup z) \setminus \{v_k\}$, such that (assuming the protocol HP is obeyed) the following conditions hold for some invariants, I_1 and I_2, of $\mathcal{E}^k$:

$$(I_1 \wedge g_{v_k}) \quad \Rightarrow \quad (pc = i - 1 \Leftrightarrow (b_1 \wedge \neg v_k)) \tag{6}$$

$$(I_2 \wedge \overline{g}_{v_k}) \quad \Rightarrow \quad (pc = j - 1 \Leftrightarrow (b_2 \wedge v_k)) \tag{7}$$

Using (6) and (7) we can replace the program counter expressions in the two guards. The new system is as follows:

$$
\begin{array}{ll}
\mathcal{E}^{k+1} :: & |[\ \textbf{var}\ \ldots pc \cdot \ldots\ pc := 0; \\
& \quad \textbf{do} \\
& \quad \ldots \ [\!]\ g_{v_k} \wedge b_1 \wedge \neg v_k \rightarrow v_k\!\uparrow; pc := i \\
& \quad \ldots \ [\!]\ \overline{g}_{v_k} \wedge b_2 \wedge v_k \rightarrow v_k\!\downarrow; pc := j \\
& \quad \ldots \\
& \quad \textbf{od} \\
&]|: z
\end{array}
$$

Since the transformation replaces the two guards with equivalent guards, the set of traces is the same for both systems, $\mathcal{E}^k$ and $\mathcal{E}^{k+1}$. Therefore, the following refinement holds:

$$\mathcal{E}^k \sqsubseteq \mathcal{E}^{k+1}$$

The main practical problem in the above transformation is to find the expressions b_1 and b_2 for the element $(E_{v_k}, \overline{E}_{v_k})$. Some heuristics restricting and directing the search can be used to solve this task. However, we omit the technical details of the search process in this paper.

(2) The boolean expressions b_1 and b_2 satisfying the conditions (6) and (7) do not exist or we are unable to find them. In such a case, a fresh local variable, e_k, and two auxiliary actions, modelling the transitions $e_k\uparrow$ and $e_k\downarrow$, are inserted into the system $\mathcal{E}^k$. Assuming that $0 \leq l_1 < i$, and $i < l_2 < j + 2$, the new system

$$
\begin{array}{ll}
\mathcal{E}^{k'} :: & |[\ \textbf{var}\ \ldots\ e_k, pc \cdot \ldots\ e_k\!\downarrow; pc := 0; \\
& \quad \textbf{do} \\
& \quad \ldots \ [\!]\ pc = l_1 - 1 \rightarrow e_k\!\uparrow; pc := l_1 \\
& \quad \ldots \ [\!]\ pc = i \wedge g_{v_k} \wedge e_k \rightarrow v_k\!\uparrow; pc := i + 1 \\
& \quad \ldots \ [\!]\ pc = l_2 - 1 \rightarrow e_k\!\downarrow; pc := l_2 \\
& \quad \ldots \ [\!]\ pc = j + 1 \wedge \overline{g}_{v_k} \wedge \neg e_k \rightarrow v_k\!\downarrow; pc := j + 2 \\
& \quad \ldots \\
& \quad \textbf{od} \\
&]|: z
\end{array}
$$

is a refinement of $\mathcal{E}^k$:

$$\mathcal{E}^k \sqsubseteq \mathcal{E}^{k'}$$

After this a new search of the expressions b_1 and b_2 is attempted for the variable v_k in the system $\mathcal{E}^{k'}$.

Actually, the strengthening conjunction or the auxiliary variable is introduced only for the guard of the action that needs strengthening rather than for the whole pair $(E_{v_k}, \overline{E}_{v_k})$.

At the end, the program counter is not mentioned in the guards of the actions and as a result, a new system, $\mathcal{E}^n$, of the form

$$
\begin{array}{lll}
\mathcal{E}^n :: & |[& \mathbf{var}\ w, pc \cdot J'; pc := 0; \\
 & & \mathbf{do} \\
 & & [\!]\ g'_{w_1} \wedge \neg w_1 \rightarrow w_1\!\uparrow; pc := 1 \\
 & & \ldots \\
 & & [\!]\ \overline{g}'_{w_{n+l}} \wedge w_{n+l} \rightarrow w_{n+l}\!\downarrow; pc := 0 \\
 & & \mathbf{od} \\
 &]|: z &
\end{array}
$$

is derived. Here, l is the number of auxiliary variables e_k, w consists of the old variables v and the auxiliary variables, whereas J' includes both the original initialization J and the initializations of the auxiliary variables e_k.

Since the values of the program counter are never used in the system $\mathcal{E}^n$, the variable itself and all assignments of the form $pc := i$ are removed from the system giving a new system $\mathcal{E}'$:

$$\mathcal{E}^n \sqsubseteq \mathcal{E}'$$

Since the elements are non-interfering in the original system $\mathcal{E}$, the same property holds for the refined system $\mathcal{E}'$ as well. However, the elements of $\mathcal{E}'$ are still unstable, because actions can disable themselves. To establish the stability property we need an assumption concerning fairness in the execution of action systems. Thus we say that an action system is *weak-fair*, if no stuttering action is forever executed in it. This means that sequences of states with infinite stuttering are excluded from the set of traces of the system. Note that the weak-fairness assumption exactly corresponds to the reality: any circuit produces an output signal in finite time after receiving input signals.

Now for each action, $g'_{w_i} \wedge \neg w_i \rightarrow w_i\!\uparrow$ ($g'_{w_j} \wedge w_j \rightarrow w_j\!\downarrow$), in $\mathcal{E}'$, we add a stuttering action, $g'_{w_i} \wedge w_i \rightarrow w_i\!\uparrow$ ($g'_{w_j} \wedge \neg w_j \rightarrow w_j\!\downarrow$) into the system. Assuming the resulting system $\mathcal{E}''$ is weak-fair, we have:

$$tr(\mathcal{E}') = tr(\mathcal{E}'').$$

Indeed, a finite execution of stuttering actions in $\mathcal{E}''$ does not add new traces to the set $tr(\mathcal{E}'')$. On the other hand, weak-fairness of $\mathcal{E}''$ excludes infinite executions of such actions. As the last step in the circuit extraction, we merge each action $g'_{w_i} \wedge \neg w_i \rightarrow w_i\!\uparrow$ ($g'_{w_j} \wedge w_j \rightarrow w_j\!\downarrow$) and its stuttering counterpart into a single action using the following property of action systems:

LEMMA 3 *A pair of actions of the form $g \wedge b \rightarrow S$ and $g \wedge \neg b \rightarrow S$ is equivalent to the single action $g \rightarrow S$ in any action system $\mathcal{A}$.*

Such a merging gives the system

$$\mathcal{E}''' :: \ |[\ \mathbf{var}\ w \cdot J';\ \mathbf{do}\ g'_{w_1} \rightarrow w_1\!\uparrow\ \|\ \ldots\ \|\ \overline{g}'_{w_{n+l}} \rightarrow w_{n+l}\!\downarrow\ \mathbf{od}\ ]| : z$$

which is a refinement of the original system:

$$\mathcal{E} \sqsubseteq \mathcal{E}'''$$

Here, the resulting action system, $\mathcal{E}'''$, is a (Q)DI system and hence, models the behaviour of a (Q)DI circuit.

6 Implementation

In this section we show how (Q)DI action systems are related to the circuit gate-level constructs.

Circuit elements Depending on the component library, different sets of basic circuit elements are available to a circuit designer. Therefore, we introduce a notion of an abstract circuit element, and show how a (Q)DI system is implemented using such elements. Thus, a *circuit element* is a (Q)DI action system of the form

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

where J_v is an initialization of v.

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 a single-output element, the corresponding combinational element is called a *gate*, denoted $Gate(gE_v; v)$. If the above pair is a multi-output element, then the corresponding circuit element is called a *fork*.

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*. Different types of C-elements are basic sequential elements. As an example, let the pair $(E_v, \overline{E}_v)$ be of the form

$$(\ a \wedge b \rightarrow v\!\uparrow,\quad \neg a \wedge \neg b \rightarrow v\!\downarrow\ )$$

Then the corresponding circuit element $\mathcal{E}^v$, denoted $C(a, b; v)$, is a normal (symmetric) C-element.

Any (Q)DI system can be written as a parallel composition of abstract circuit elements. Let $\mathcal{E}$ be a (Q)DI system of the form

$\mathcal{E}$ :: $|[$ **var** $w \cdot J;$ **do** E_{w_i} $\| \ \ldots \ \|$ $\overline{E}_{w_n}$ **od** $]| : z$

where $w = w_1, \ldots, w_n$. Let $\mathcal{E}^{w_i}$ be the following circuit element:

$\mathcal{E}^{w_i}$ :: $|[$ **var** $w_i^* \cdot J_{w_i};$ **do** E_{w_i} $\|$ $\overline{E}_{w_i}$ **od** $]| : (vE_{w_i} \cup v\overline{E}_{w_i}) \setminus \{w_i\}$

Then the system $\mathcal{E}$ is implemented as a composition of the circuit elements $\mathcal{E}^{w_i}$:

$$\mathcal{E} = \mathcal{E}^{w_1} \parallel \ldots \parallel \mathcal{E}^{w_n}$$

Implementation of circuit elements We assume that the circuit elements are implemented using a component library which contains a sufficient selection of different combinational gates and C-elements.

An arbitrary combinational element is realized with a standard combinational gate (and, or, nand, nor, not, xor, wire etc.) or a combinational network of such gates.

An arbitrary sequential element of the form

$\mathcal{E}^v$ :: $|[$ **var** $v^* \cdot J_v;$ **do** E_v $\|$ $\overline{E}_v$ **od** $]| : (vE_v \cup v\overline{E}_v) \setminus \{v\}$

can be transformed into the composition

$$Gate(gE_v; a) \parallel Gate(\neg g\overline{E}_v; b) \parallel C(a, b; v)$$

as shown in [18]. Here the system $Gate(gE_v; v)$ $(Gate(\neg g\overline{E}_v; v))$ performs the continuous assignment of the boolean expression gE_v $(\neg g\overline{E}_v)$ to the boolean variable v, whereas $C(a, b; v)$ specifies the operation of the two-input C-element with the inputs a and b and the output v. The above composition is implemented using separate combinational gate networks for $Gate(gE_v; a)$ and $Gate(\neg g\overline{E}_v; b)$, and a two-input C-element of our component library.

Reset-supplement We usually need to add the possibility of controlled initialization into an elementary action system describing a digital circuit. This is done by arranging a universal reset-signal, say rst, for those sequential elements that need to be at the certain initial state but are not initialized automatically by the initial state of the other variables in the system. The signal rst is controlled by the global environment and it is meant to be activated only once, before the standard operation of the system is initiated.

Let $\mathcal{E}$ be a sequential element of the form

$\mathcal{E}^v$ :: $|[$ **var** $v^* \cdot J_v;$ **do** E_v $\|$ $\overline{E}_v$ **od** $]| : (vE_v \cup v\overline{E}_v) \setminus \{v\}$

If the initialization of the variables in $vE_v \cup v\overline{E}_v$ makes the guard $g\overline{E}_v$ *false*, then an active-high type reset-variable, rst, is added into the guards of the actions as follows:

$$\begin{aligned} gE_v' &= gE_v \wedge \neg rst \\ g\overline{E}_v' &= g\overline{E}_v \vee rst \end{aligned}$$

This transformation preserves the properties of the system, because rst is constantly false, and thus ineffective, during the normal operation.

Isochronic forks We must also ensure that the isochronicity assumption does not introduce problems, i.e., hazards, at the circuit-level. This means that the isochronic forks are made local by isolating the problematic connections between systems from the internal connections using appropriate buffering [10]. Also some internal forks may have to be manipulated by speeding up critical slow branches with respect to fast branches. To achieve this, we can apply the *De Morgan's laws* and introduce some auxiliary inverters to enforce the needed order of transitions on the branches.

7 Conclusions

We have introduced an action systems-based design framework for asynchronous (quasi) delay-insensitive circuits. The emphasis has been on the correctness of the transformations rather than on the performance, power consumption, and size features of the target circuit. Naturally, high-speed and low-power design are important issues and they will be addressed in future work. Also the concept of isochronicity will be formalized in a more detailed manner and included explicitly into the framework. Furthermore, some notational enhancements are necessary to make the action systems formalism more attractive and practical for a circuit designer. In this respect, a tool supporting graphical representation of and manipulation with action systems looks very promising. An example of such representation can be found in [23]. Finally, several substantial case studies are needed to evaluate the practical use of our method. Such case studies are under way [19].

Acknowledgments

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

References

[1] V. Akella and G. Gopalakrishnan. SHILPA: A high-level synthesis system for self-timed circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 587–591, 1992.

[2] R.-J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[3] R.-J. R. Back, A. J. Martin, and K. Sere. An action system specification of the Caltech asynchronous microprocessor. In B. Moller, editor, *Mathematics of Program Construction'95, Kloster Irsee, Germany, July 1995*, volume 947 of *Lecture Notes in Computer Science*. Springer–Verlag, 1995.

[4] R.-J. R. Back and K. Sere. From modular systems to action systems. *Formal Methods Europe'94, Spain, October 1994*, *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[5] 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.

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

[7] E. Brunvand and R. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 262–265, 1989.

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

[9] T. A. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. Technical Report MIT-LCS-TR-393, Massachusetts Institute of Technology, 1987. Ph.D. Thesis.

[10] A. Davis. Synthesizing asynchronous circuits: practise and experience. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, pages 104–150. Springer–Verlag, 1995.

[11] 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.

[12] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, 1976.

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

[14] J. Grundy, M. J. Butler, T. Långbacka, R. Rukšėnas, and J. von Wright. The Refinement Calculator: A tool support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, *Discrete Mathematics and Theoretical Computer Science* series, pages 40–61. Springer-Verlag, July 1997.

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

[16] A. J. Martin. Synthesis of asynchronous VLSI circuits. Technical Report, CalTech, 1993.

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

[18] J. Plosila, R. Rukšėnas, and K. Sere. Delay-insensitive circuits and action systems. TUCS Technical Report 60, Turku Centre for Computer Science, November 1996. Available from `http://www.tucs.abo.fi/publications/techreports/TR60.ps.gz`.

[19] J. Plosila and T. Seceleanu. An asynchronous linear predictive analyzer. TUCS Technical Report 142, Turku Centre for Computer Science, November 1997. Available from `http://www.tucs.abo.fi/publications/techreports/TR142.ps.gz`.

[20] J. Plosila and K. Sere. Action Systems in Pipelined Processor Design. In *Proc. of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eindhoven, The Netherlands, April 1997*. IEEE Computer Society Press, 1997.

[21] R. Rukšėnas and K. Sere. Handshaking expansion as action system refinement. TUCS Technical Report 55, Turku Centre for Computer Science, October 1996. Available from `http://www.tucs.abo.fi/publications/techreports/TR55.ps.gz`.

[22] F. Schalij. Tangram manual. Technical Report UR-008/93, Philips Research Laboratories, Eindhoven, 1993.

[23] K. Sere. Design of DI circuits within the action systems framework. Computing Science Notes CSN 9602, University of Groningen, 1996.

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

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

[26] A. V. Yakolev, A. M. Koelmans, and L. Lavagno. High level modelling and design of asynchronous interface logic. Computing Science Technical Report No. 460, University of Newcastle upon Tyne, November 1993.

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**