Fundamenta Informaticae (to appear) 1001–1031
IOS Press

Controllable Delay-Insensitive Processes

Mark B. Josephs
Centre for Concurrent Systems and VLSI
London South Bank University, UK
`josephmb@lsbu.ac.uk`

Hemangee K. Kapoor
Dhirubhai Ambani Institute of Information and
Communication Technology, Gandhinagar, India
`hemangee_kapoor@daiict.ac.in`

Abstract. Josephs and Udding's DI-Algebra offers a convenient way of specifying and verifying designs that must rely upon delay-insensitive signalling between modules (asynchronous logic blocks). It is based on Hoare's theory of CSP, including the notion of refinement between processes, and is similarly underpinned by a denotational semantics. Verhoeff developed an alternative theory of delay-insensitive design based on a testing paradigm and the concept of reflection. The first contribution of this paper is to define a relation between processes in DI-Algebra that captures Verhoeff's notion of a closed system passing a test (by being free of interference and deadlock). The second contribution is to introduce a new notion of controllability, that is, to define what it means for a process to be controllable in DI-Algebra. The third contribution is to extend DI-Algebra with a reflection operator and to show how testing relates to controllability, reflection and refinement. It is also shown that double reflection yields fully-abstract processes in the sense that it removes irrelevant distinctions between controllable processes. The final contribution is a modified version of Verhoeff's factorisation theorem that could potentially be of major importance for constructive design and the development of design tools. Several elementary examples are worked out in detail to illustrate the concepts. The claims made in this paper are accompanied by formal proofs, mostly in an annotated calculational style.

Keywords: Process algebra, refinement, testing, controllability, reflection, factorisation

Address for correspondence: Prof. M.B. Josephs, Faculty of BCIM, London South Bank University, 103 Borough Road, London SE1 0AA, UK.

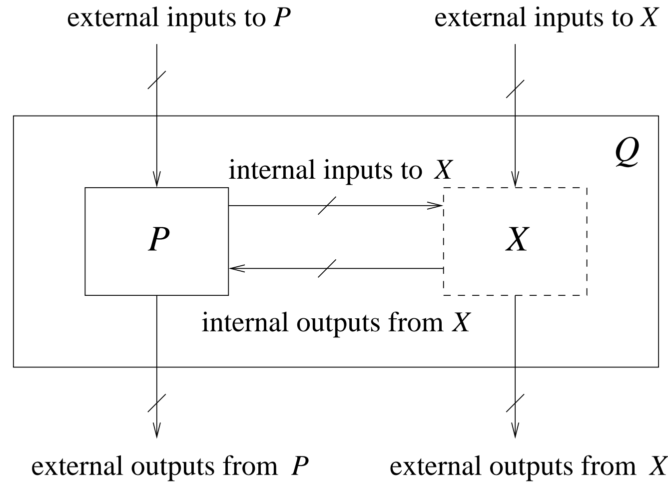

Figure 1. Design by factorisation.

1. Introduction

In top-down design, a module is decomposed into submodules that will interact so as to achieve the required top-level functionality. This paper is concerned with the formal description of modules and submodules using process algebra. Given the description P of a submodule (e.g. a library component) intended to be part of a module described by Q, the problem of design by factorisation, also known as submodule construction, is to find the most general description X of what remains. The submodule described by X may then itself be decomposed. Here, as illustrated in Figure 1, we shall assume that Q describes how the module should interact through external inputs and outputs with its environment, and that Q does not mention internal communications between the submodules.

Bochmann [21] proposed a constructive method to solve the problem of design by factorisation, but ignored deadlock and livelock (loops without progress, or infinite chatter). Besides communication protocols, his group has addressed the problem in such contexts as supervisory control [2] and I/O automata [6]. Parrow [26] proposed a method that solves the problem for descriptions expressed in CCS [22]; an improvement over [21] is that the method takes into account potential deadlocks.

Verhoeff [28] solved this design problem for delay-insensitive processes [27, 14]. Such processes are typically implemented as asynchronous logic blocks; the possibility of transmission interference along the wires that connect them is considered to be a design error. He formulated the problem as follows: find the most nondeterministic process X such that P in parallel with X refines Q. This is written $P \,||\, X \sqsupseteq Q$, where $||$ is a form of parallel composition and $\sqsupseteq$ is a refinement (conformance) relation between processes.

Verhoeff's Factorisation Theorem involves the application of parallel composition and reflection (mirror) operators to a semantic model. It is underpinned by a theoretical framework related to De Nicola and Hennessy's work on process testing [25, 8] and Dill's conformation checking [5]. Verhoeff's semantic model (the XDI model [29]) of processes is based upon finite traces (as in Dill's model) and addresses both safety and progress properties, whereas Dill's only captures safety properties. Verhoeff pointed out that his approach is too optimistic in that it ignores livelock. Subsequently, Mallon [19, 20] found some

technical problems with Verhoeff's approach. He was able to work around these by extending the XDI model to include certain 'unhealthy' processes and redefining the parallel composition operator. Verhoeff's approach has also been incorporated into Negulescu's theory of 'process spaces' [24] and Benko and Ebergen's work on the composition of 'snippets' [1].

Delay-insensitive processes are most conveniently expressed in DI-Algebra [14], which is based on CSP [9]. DI-Algebra treats livelock as a design error, just like transmission interference. When it comes to processes that can refuse to output, Verhoeff's XDI model distinguishes between input-demanding processes and indifferent processes; the denotational semantics of DI-Algebra [17, 19] is usually given in a failures/divergences model [10] that does not make this distinction. Verhoeff also pointed out that it would be interesting to extend DI-Algebra with a reflection operator. This is something that Lucassen [17] tried to do, but was unsuccessful.

This paper takes up the challenge of reconstructing Verhoeff's theory in the framework of DI-Algebra. Testing a process involves connecting it to another process so as to form a closed system: if their interaction guarantees that the system is free from interference and deadlock, the process passes the test. We classify a process as controllable if there is some test that it can pass.[1] In particular, we shall prove that a controllable process successfully interacts with its reflection. A controllable process that can refuse to output must be able to safely receive some input.[2] Our definition of reflection lends itself to symbolic manipulation of terms in the process algebra, whereas Verhoeff and Mallon work within a semantic model. It transpires that in DI-Algebra a controllable process should be twice reflected so as to make it fully abstract under testing. In other words, double reflection removes irrelevant distinctions between controllable processes.

We show that Galois connections from Verhoeff's theory carry over in suitably modified form to DI-Algebra. These relate the refinement ordering of DI-Algebra (that allows one to replace a process with a more deterministic one) to (i) reflection and (ii) factorisation (that involves reflection and parallel composition). Like Mallon [19, 20], we must take care to avoid infinite chatter when performing factorisation. Indeed, Mallon developed a tool, `ludwig`, based on the XDI model that can automatically factorise DI processes and display their components as state graphs.

The remainder of this paper is organised as follows. Section 2 begins with a short introduction to DI-Algebra. It then shows how to analyse the observable behaviour of a process as it evolves, using this framework to formalise the concept of refinement between processes. The section ends by giving some insight into the trace-theoretic semantics of DI-Algebra. The concepts of testing-by-interaction and controllability are introduced in Sections 3 and 4, respectively. The reflection operator on processes is defined in Section 5 and we establish our main result, namely, the reflection of a controllable process is the most nondeterministic process with which it can successfully interact. Full abstraction with respect to testing-by-interaction is examined in Section 6 and, finally, a factorisation theorem is given in Section 7 and applied to some small examples.

2. DI-Algebra

A process in DI-Algebra describes a delay-insensitive module (asynchronous logic block) that reacts to signal transitions (changes in logic level from logic-0 to logic-1 or vice versa). Such a process is

[1]Unlike Verhoeff, we do not allow interaction with miracles, a test that every process can pass.
[2]Note that Verhoeff imposes such a restriction on input-demanding processes, whereas Mallon drops it.

associated with an alphabet C (a finite set of control signals) that is partitioned into an input alphabet A and an output alphabet B.[3]

The process $a? \,;\, P$ describes a module that inputs a transition on $a \in A$ (once one becomes available) and then behaves like the process P. This is called input-prefixing. Output-prefixing is similar: $b! \,;\, P$ describes a module that outputs a transition on $b \in B$ and then behaves like P. (The ? and ! are just syntactic sugar and can be omitted if the direction of communication is clear.) As modules are loosely coupled, the order in which we record the input of multiple transitions or the output of multiple transitions by a module is unimportant: $a? \,;\, a'? \,;\, P \;=\; a'? \,;\, a? \,;\, P$ and $b! \,;\, b'! \,;\, P \;=\; b'! \,;\, b! \,;\, P$. An unexpected or unsafe input leads to undefined behaviour denoted by the 'divergent' process $\bot$. In particular, pulses cannot be reliably transmitted: $a? \,;\, a? \,;\, P \;=\; a? \,;\, a? \,;\, \bot$ and $b! \,;\, b! \,;\, P \;=\; \bot$.

A guarded choice describes a module that must first select an action to perform. For example, the guarded choice $[b! \rightarrow P \,\Box\, a? \rightarrow Q]$ describes a module that must output a transition on b unless it can input one on a. The module then behaves like P or like Q, as appropriate. Any $skip$ guard or output guard can be selected unconditionally, whereas an input guard can only be selected if that input is available. A module that does not perform any action can be modelled by the process $stop$ which is equivalent to a choice with no alternatives, i.e., $[\,]$. One choice is no choice, as follows: $[\, a? \rightarrow P \,] = a? \,;\, P$, $[\, b! \rightarrow P \,] = b! \,;\, P$ and $[\, skip \rightarrow P \,] = P$. The nondeterministic choice between two processes P and Q is denoted by $P \sqcap Q$, and can also be expressed as $[\, skip \rightarrow P \,\Box\, skip \rightarrow Q \,]$.

Cyclic behaviour is expressed using recursion. For example, a Wire module that alternates between inputting a transition on a and outputting a transition on b is described by $W_{a,b} \;=\; a? \,;\, b! \,;\, W_{a,b}$. Formally, the meaning of a recursion $P = f(P)$ is the least fixed point $\mu X.f(X)$ of f. Its successive approximations are $\bot$, $f(\bot)$, $f(f(\bot))$, etc.

Parallel composition is denoted by the infix operator $||$. In the parallel composition $P \,||\, Q$, the input (output) alphabet of P should be disjoint from that of Q. The input (output) alphabet of $P \,||\, Q$ then consists of those input (output) signals of P and of Q that are not output (input) signals of the other. The processes composed in parallel describe submodules connected by a wire for every signal that the processes have in common. The propagation of transitions along these wires is not observable in the composite.

$P[c'/c]$, where $c \in C$ and $c' \notin C$, describes a module that behaves like P except that signal c is renamed to c'. As a consequence of delay-insensitivity, this renaming operator is a special case of parallel composition:

$$P[c'/c] = \begin{cases} P \,||\, W_{c',c} & \text{, if } c \in A \\ P \,||\, W_{c,c'} & \text{, if } c \in B\,. \end{cases}$$

2.1. Observable behaviour

The initial behaviour of a process P, as observed by its environment, is characterised by

[3] Readers who are more familiar with Petri nets than with asynchronous logic should think of a module as being part of a Petri net; this fragment is connected to the remainder of the net by a set C of places through which tokens flow. Thus a module corresponds to a Petri net fragment, a signal to a place, and a (signal) transition to a token [30, 11]. The removal of a token from place $a \in A$ mimics the input of a transition on signal a by the module; the addition of a token to place $b \in B$ mimics the output of a transition on signal b by the module. It is considered unsafe ('transmission interference') for such places to be occupied by more than one token at any time.

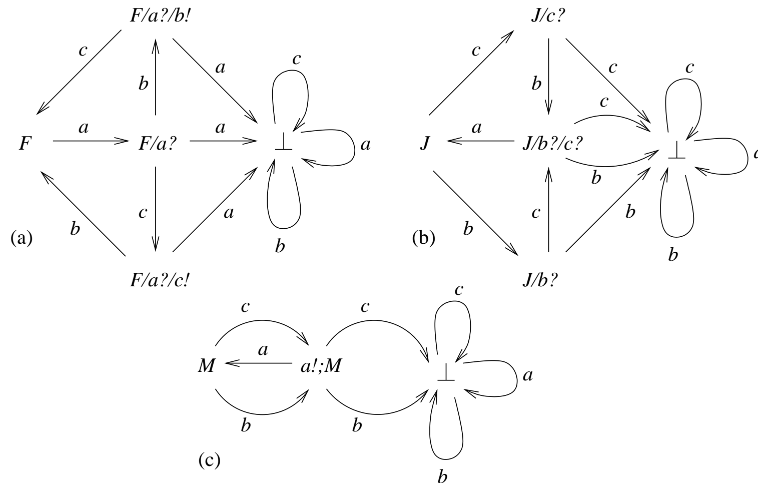

Figure 2. Graphical representation of the evolution of processes F, J and M.

div(P)	whether or not it is divergent
out(P)	the set of initially-possible outputs
refuses(P)	whether or not it can refuse to output

where *out*$(P) \subseteq B$, *out*$(P) = \emptyset \Rightarrow$ *refuses*(P) and *div*$(P) \Rightarrow$ *out*$(P) = B \wedge$ *refuses*(P). Note that there are non-divergent processes with a non-empty set of initially-possible outputs that can refuse to output, e.g., $stop \sqcap (b! \; ; \; stop)$. It is nondeterministic whether such a process will output or not. Indeed an output that is initially possible need not remain so after some other output has been observed, e.g., the process $(b! \; ; \; stop) \sqcap (b'! \; ; \; stop)$ chooses nondeterministically between outputting on b and outputting on b'.

If P is constructed without using $||$, then its initial behaviour can be established by inspection, as follows:

- *div*$(\perp)$, *out*$(\perp) = B$ and *refuses*$(\perp)$.
- A guarded choice is divergent if there is a $skip$-guarded process P and *div*(P), or there is a $b!$-guarded process P with $b \in$ *out*(P) (because of the possibility of transmission interference).
- The set of initially-possible outputs of a guarded choice is B if it can diverge. Otherwise, it is the set of all output guards and initially-possible outputs of each $skip$-guarded or output-guarded process.
- A guarded choice can refuse to output if it is divergent or if there are only input guards. Otherwise, it can only refuse to output if there is a $skip$-guarded process that can refuse to do so.
- *div*$(\mu X.f(X)) \equiv$ *div*$(f(\perp))$, *out*$(\mu X.f(X)) =$ *out*$(f(\perp))$ and *refuses*$(\mu X.f(X)) \equiv$ *refuses*$(f(\perp))$.

As in CSP, P/c describes a module (that behaves like P) *after* an event (transition on c) has been observed. That is, a process P evolves to $P/a?$ after a transition on $a \in A$ has been sent to the module

by its environment, and to $P/b!$ after a transition on $b \in out(P)$ from the module has been received by its environment. Note that $P/a?/a? = \bot$ for any P because of the possibility of transmission interference. Also, the / operator generalises to sequences of events: $P/\varepsilon = P$ and $P/tc = (P/t)/c$ if $c \in A \cup out(P/t)$.

Example 2.1. The process F with input alphabet $\{a\}$ and output alphabet $\{b, c\}$ is defined by $F = a?\ ;\ b!\ ;\ c!\ ;\ F$. It describes a Fork module. Fig. 2(a) shows how F evolves as events are observed one at a time. The following table provides a complete analysis of its observable behaviour.

Process			*div*	*out*	*refuses*
F	=	$a?\ ;\ b!\ ;\ c!\ ;\ F$	false	$\emptyset$	true
$F/a?$	=	$b!\ ;\ c!\ ;\ F$	false	$\{b, c\}$	false
$F/a?/a?$	=	$\bot$	true	$\{b, c\}$	true
$F/a?/b!$	=	$c!\ ;\ F$	false	$\{c\}$	false
$F/a?/c!$	=	$b!\ ;\ F$	false	$\{b\}$	false
$F/a?/b!/a?$	=	$\bot$		(see above)	
$F/a?/b!/c!$	=	F		(see above)	
$F/a?/c!/a?$	=	$\bot$		(see above)	
$F/a?/c!/b!$	=	F		(see above)	

Example 2.2. As a second example, consider the process J with input alphabet $\{b, c\}$ and output alphabet $\{a\}$, defined by $J = b?\ ;\ c?\ ;\ a!\ ;\ J$, that describes a Join module, Fig. 2(b).

Process			*div*	*out*	*refuses*
J	=	$b?\ ;\ c?\ ;\ a!\ ;\ J$	false	$\emptyset$	true
$J/b?$	=	$[\ b? \rightarrow \bot \,\Box\, c? \rightarrow a!\ ;\ J\ ]$	false	$\emptyset$	true
$J/c?$	=	$[\ c? \rightarrow \bot \,\Box\, b? \rightarrow a!\ ;\ J\ ]$	false	$\emptyset$	true
$J/b?/b?$	=	$\bot$	true	$\{a\}$	true
$J/b?/c?$	=	$[\ b? \rightarrow \bot \,\Box\, c? \rightarrow \bot \,\Box\, a! \rightarrow J\ ]$	false	$\{a\}$	false
$J/c?/b?$	=	$J/b?/c?$		(see above)	
$J/c?/c?$	=	$\bot$		(see above)	
$J/b?/c?/a!$	=	J		(see above)	
$J/b?/c?/b?$	=	$\bot$		(see above)	
$J/b?/c?/c?$	=	$\bot$		(see above)	

Example 2.3. As a final example, consider the process M also with input alphabet $\{b, c\}$ and output alphabet $\{a\}$, defined by $M = [\ b? \rightarrow a!\ ;\ M \,\Box\, c? \rightarrow a!\ ;\ M\ ]$, that describes a Merge module, Fig. 2(c).

Process			*div*	*out*	*refuses*
M	=	$[\,b? \to a!\,;\, M \,\Box\, c? \to a!\,;\, M\,]$	false	$\emptyset$	true
$M/b?$	=	$a!\,;\, M$	false	$\{a\}$	false
$M/c?$	=	$a!\,;\, M$		(see above)	
$M/b?/a!$	=	M		(see above)	
$M/b?/b?$	=	$\bot$	true	$\{a\}$	true
$M/b?/c?$	=	$\bot$		(see above)	

A deeper treatment of this subject [7, 18] explains how processes with the same observable behaviour can be transformed algebraically so that they have the same syntactic shape.

2.2. Refinement

Refinement allows one process to be substituted for another if any observation of the former is a possible observation of the latter. Here we formally define P refines Q, written as $P \sqsupseteq Q$ or as $Q \sqsubseteq P$, in terms of the limit as i tends to ∞ of finite approximations[4] $\sqsupseteq_i$:

$$P \sqsupseteq Q \;\equiv\; (\forall i \,:\, 0 \le i \,:\, P \sqsupseteq_i Q)\,,$$

where P refines Q for 0 events if any initial observation of P is a possible observation of Q,

$$P \sqsupseteq_0 Q \;\equiv\; (\mathit{div}(P) \Rightarrow \mathit{div}(Q)) \wedge (\mathit{out}(P) \subseteq \mathit{out}(Q)) \wedge (\mathit{refuses}(P) \Rightarrow \mathit{refuses}(Q))\,,$$

and P refines Q for $i+1$ events if it does so for i events, and every successor process of P (that arises from an input or output event) refines the successor of Q (that arises from that event) for i events,

$$P \sqsupseteq_{i+1} Q \;\equiv\; P \sqsupseteq_i Q \wedge (\forall c \,:\, c \in A \cup \mathit{out}(P) \,:\, P/c \sqsupseteq_i Q/c)\,.$$

Refinement satisfies the following 'circular definition'.

Proposition 2.1. $P \sqsupseteq Q \;\equiv\; (\mathit{div}(P) \Rightarrow \mathit{div}(Q)) \wedge (\mathit{out}(P) \subseteq \mathit{out}(Q)) \wedge (\mathit{refuses}(P) \Rightarrow \mathit{refuses}(Q)) \wedge$
$(\forall c \,:\, c \in A \cup \mathit{out}(P) \,:\, P/c \sqsupseteq Q/c)\,.$

Proof:

$P \sqsupseteq Q$

$\equiv$ { Definition of $\sqsupseteq$ }

$(\forall i \,:\, 0 \le i \,:\, P \sqsupseteq_i Q)$

$\equiv$ { Splitting the range of i }

$P \sqsupseteq_0 Q \wedge (\forall i \,:\, 0 \le i \,:\, P \sqsupseteq_{i+1} Q)$

$\equiv$ { Definition of $\sqsupseteq_{i+1}$ and predicate calculus }

[4] This is similar to the way Milner originally defined observation equivalence in CCS [22].

$$
\begin{array}{ll}
& P \sqsupseteq_0 Q \wedge (\forall c \;:\; c \in A \cup \mathit{out}(P) \;:\; (\forall i \;:\; 0 \leq i \;:\; P/c \sqsupseteq_i Q/c)) \\
\equiv & \quad \{ \text{ Definition of } \sqsupseteq_0 \text{ and } \sqsupseteq \} \\
& (\mathit{div}(P) \Rightarrow \mathit{div}(Q)) \wedge (\mathit{out}(P) \subseteq \mathit{out}(Q)) \wedge (\mathit{refuses}(P) \Rightarrow \mathit{refuses}(Q)) \;\wedge \\
& (\forall c \;:\; c \in A \cup \mathit{out}(P) \;:\; P/c \sqsupseteq Q/c)\,.
\end{array}
$$

□

Note that refinement induces an equivalence on processes, i.e., $P = Q \;\equiv\; P \sqsupseteq Q \wedge Q \sqsupseteq P$.

2.3. Trace-theoretic semantics

Suppose a process P has engaged in a finite sequence (trace) t of events. As a result it may have become divergent, $\mathit{div}(P/t)$, and t is called a divergence of P. Another possibility is that the process may have become quiescent, refusing to output at least until further input is supplied. In either case $\mathit{refuses}(P/t)$, and t is called a failure of P. So, by definition, every divergence is also a failure and every failure is also a trace [10].

A denotational semantics is given in [17, 19] to DI-Algebra. This associates a set $F[\![P]\!]$ of failures with each process P. For example, $F[\![\bot]\!] = (A \cup B)^*$ and $F[\![P/c]\!] = \{t \;:\; ct \in F[\![P]\!] \;:\; t\}$ for $c \in A \cup \mathit{out}(P)$. The set $T[\![P]\!]$ of traces can be calculated from $F[\![P]\!]$; so can the set $D[\![P]\!]$ of divergences provided that $B \neq \emptyset$ (which we shall assume here). Moreover, $\mathit{div}(P) \equiv \varepsilon \in D[\![P]\!]$, $\mathit{out}(P) = \{b \;:\; b \in B \wedge b \in T[\![P]\!] \;:\; b\}$ and $\mathit{refuses}(P) \equiv \varepsilon \in F[\![P]\!]$.

In this paper, as in [19, 20], *we restrict ourselves to parallel compositions $P_0 \,||\, P_1$ that cannot give rise to infinite chatter*. Let P_i $(i = 0, 1)$ have alphabet C_i partitioned into A_i and B_i, and let $C_2 = (C_0 \setminus C_1) \cup (C_1 \setminus C_0)$. Note that $C_0 \cup C_1 = C_0 \cup C_2$. We shall assume not only that $A_0 \cap A_1 = \emptyset$ and $B_0 \cap B_1 = \emptyset$, but also that P_0 and P_1 cannot communicate with each other ad infinitum without one of them becoming divergent. Using $t \upharpoonright C_j$ $(j = 0, 1, 2)$ to denote the restriction of t to C_j, the semantics of parallel composition[5] is then given by

$$
\begin{array}{rcll}
D[\![P_0 \,||\, P_1]\!] & = & \{t, t_0, t_1, t_2, u_2 & : \; t \in (C_0 \cup C_1)^* \wedge t_0 = t \upharpoonright C_0 \wedge t_1 = t \upharpoonright C_1 \wedge t_2 = t \upharpoonright C_2 \;\wedge \\
& & & \quad u_2 \in C_2^* \;\wedge ((t_0 \in T[\![P_0]\!] \wedge t_1 \in D[\![P_1]\!]) \;\vee \\
& & & \qquad (t_1 \in T[\![P_1]\!] \wedge t_0 \in D[\![P_0]\!])) \;:\; t_2 u_2\} \\
T[\![P_0 \,||\, P_1]\!] & = & \{t, t_0, t_1, t_2 & : \; t \in (C_0 \cup C_1)^* \wedge t_0 = t \upharpoonright C_0 \wedge t_1 = t \upharpoonright C_1 \wedge t_2 = t \upharpoonright C_2 \;\wedge \\
& & & \quad t_0 \in T[\![P_0]\!] \wedge t_1 \in T[\![P_1]\!] \;:\; t_2\} \\
& & \cup\, D[\![P_0 \,||\, P_1]\!] & \\
F[\![P_0 \,||\, P_1]\!] & = & \{t, t_0, t_1, t_2 & : \; t \in (C_0 \cup C_1)^* \wedge t_0 = t \upharpoonright C_0 \wedge t_1 = t \upharpoonright C_1 \wedge t_2 = t \upharpoonright C_2 \;\wedge \\
& & & \quad t_0 \in F[\![P_0]\!] \wedge t_1 \in F[\![P_1]\!] \;:\; t_2\} \\
& & \cup\, D[\![P_0 \,||\, P_1]\!]\,. &
\end{array}
$$

The following lemmas enable us to establish the standard trace-theoretic definition of refinement.

Lemma 2.1. $t \in T[\![P]\!] \wedge P \sqsupseteq Q \Rightarrow t \in T[\![Q]\!] \wedge P/t \sqsupseteq Q/t\,.$

[5] In [17] a more complex definition of the set of divergences was given because additional divergences must be included in the set if infinite chatter can arise.

Proof:
Induction on $\#t$, the length of t.
Base case: 0. Then $t = \varepsilon$ and the result follows since ε is a trace of every process and $/\varepsilon$ is an identity function on processes.
Inductive step: $i + 1$. Then $t = ct'$ for some $c \in A \cup \mathit{out}(P)$ and sequence t' of length i.

$$
\begin{array}{ll}
 & c \in A \cup \mathit{out}(P) \wedge ct' \in T[\![P]\!] \wedge P \sqsupseteq Q \\
\Rightarrow & \{ \text{ Semantics of / and Proposition 2.1 } \} \\
 & c \in A \cup \mathit{out}(Q) \wedge t' \in T[\![P/c]\!] \wedge P/c \sqsupseteq Q/c \\
\Rightarrow & \{ \text{ Induction hypothesis } \} \\
 & c \in A \cup \mathit{out}(Q) \wedge t' \in T[\![Q/c]\!] \wedge (P/c)/t' \sqsupseteq (Q/c)/t' \\
\equiv & \{ \text{ Semantics of / } \} \\
 & ct' \in T[\![Q]\!] \wedge P/ct' \sqsupseteq Q/ct' \,.
\end{array}
$$

□

Lemma 2.2. $F[\![P]\!] \subseteq F[\![Q]\!] \;\Rightarrow\; (\forall t \;:\; t \in T[\![P]\!] \;:\; P/t \sqsupseteq Q/t)\,.$

Proof:
Assume $F[\![P]\!] \subseteq F[\![Q]\!]$, which implies $D[\![P]\!] \subseteq D[\![Q]\!]$ and $T[\![P]\!] \subseteq T[\![Q]\!]$. It suffices to prove that $(\forall t \;:\; t \in T[\![P]\!] \;:\; P/t \sqsupseteq_i Q/t)$ for all i, which we do by induction on i.
Base case: 0.

$$
\begin{array}{ll}
 & \mathit{div}(P/t) \\
\equiv & \{ \text{ Property of } \mathit{div} \text{ and semantics of / } \} \\
 & t \in D[\![P]\!] \\
\Rightarrow & \{ \; D[\![P]\!] \subseteq D[\![Q]\!] \; \} \\
 & t \in D[\![Q]\!] \\
\equiv & \{ \text{ Property of } \mathit{div} \text{ and semantics of / } \} \\
 & \mathit{div}(Q/t)\,,
\end{array}
$$

and similarly for $\mathit{out}(P/t) \subseteq \mathit{out}(Q/t)$ and $\mathit{refuses}(P/t) \Rightarrow \mathit{refuses}(P/t)$. Hence, $P/t \sqsupseteq_0 Q/t$.
Inductive step: $i + 1$.

$$
\begin{array}{ll}
 & P/t \sqsupseteq_{i+1} Q/t \\
\equiv & \{ \text{ Definition of } \sqsupseteq_{i+1} \; \} \\
 & P/t \sqsupseteq_i Q/t \wedge (\forall c \;:\; c \in A \cup \mathit{out}(P/t) \;:\; (P/t)/c \sqsupseteq_i (Q/t)/c) \\
\equiv & \{ \text{ Property of } \mathit{out} \text{ and semantics of / } \} \\
 & P/t \sqsupseteq_i Q/t \;\wedge (\forall c \;:\; tc \in T[\![P]\!] \;:\; P/tc \sqsupseteq_i Q/tc) \\
\Leftarrow & \{ \text{ Induction hypothesis } \} \\
 & \text{true}\,.
\end{array}
$$

□

Theorem 2.1. $P \sqsupseteq Q \;\equiv\; F[\![P]\!] \subseteq F[\![Q]\!]$.

Proof:
($\Rightarrow$) Assume $P \sqsupseteq Q$. Then

$$
\begin{array}{ll}
 & t \in F[\![P]\!] \\
\equiv & \{ \text{ Property of } \textit{refuses} \text{ and semantics of } / \; \} \\
 & t \in T[\![P]\!] \wedge \textit{refuses}(P/t) \\
\Rightarrow & \{ \text{ Lemma 2.1 and Proposition 2.1 } \} \\
 & t \in T[\![Q]\!] \wedge \textit{refuses}(Q/t) \\
\equiv & \{ \text{ Property of refuses and semantics of } / \; \} \\
 & t \in F[\![Q]\!] \,.
\end{array}
$$

($\Leftarrow$) Lemma 2.2, putting $t = \varepsilon$. □

Note that $F[\![P \sqcap Q]\!] = F[\![P]\!] \cup F[\![Q]\!]$, so we have that $P \sqsupseteq Q$ if and only if $P \sqcap Q = Q$, as in CSP.

Finally, a process expressed in DI-Algebra is invariant when composed with a Foam Rubber Wrapper [23, 27] modelling wires of unbounded delay. In particular,

$$s \ltimes t \;\wedge\; t \in F[\![P]\!] \Rightarrow s \in F[\![P]\!] \,,$$

where the reordering relation $\ltimes$ between traces [13] allows

- input events to be moved in front of other events
- output events to be moved behind other events.

Formally, $a \in A \vee b \in B \Rightarrow s\,a\,b\,t \ltimes s\,b\,a\,t$. Note that, if $s \ltimes t$ and $t \in T[\![P]\!]$, then $P/s \sqsubseteq P/t$.

3. Testing by interaction

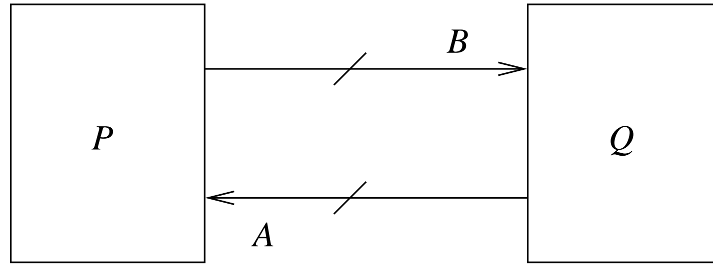

Figure 3. Closed system formed from two processes.

A process P with input alphabet A and output alphabet B can be tested by connecting it to a process Q with input alphabet B and output alphabet A so that they form a closed system, Figure 3. Following Verhoeff [28], passing the test requires that the system is *free of interference and deadlock*; P and Q

are then said to be *friends*. We shall refer to this as successful interaction between P and Q and write $P \longleftrightarrow Q$; unsuccessful interaction will be written $P \not\longleftrightarrow Q$. As we did with $\sqsupseteq$ in Section 2.2, it is convenient to define the (symmetric) relation $\longleftrightarrow$ as the limit of finite approximations $\longleftrightarrow_i$:

$$P \longleftrightarrow Q \;\equiv\; (\forall i : 0 \leq i : P \longleftrightarrow_i Q)\,,$$

where non-divergent processes that cannot both refuse to output successfully interact for 0 events,

$$P \longleftrightarrow_0 Q \;\equiv\; \neg \mathit{div}(P) \wedge \neg \mathit{div}(Q) \wedge \neg(\mathit{refuses}(P) \wedge \mathit{refuses}(Q))\,,$$

and processes successfully interact for $i+1$ events if they do so for i events, and every successor process of one (that arises from an output event) successfully interacts for i events with the successor process of the other (that arises from the corresponding input event),

$$P \longleftrightarrow_{i+1} Q \;\equiv\; P \longleftrightarrow_i Q \wedge (\forall c \;:\; c \in \mathit{out}(P) \cup \mathit{out}(Q) \;:\; P/c \longleftrightarrow_i Q/c)\,.$$

Successful interaction satisfies the following ‘circular definition’.

Proposition 3.1. $P \longleftrightarrow Q \;\equiv\; \neg \mathit{div}(P) \wedge \neg \mathit{div}(Q) \wedge \neg(\mathit{refuses}(P) \wedge \mathit{refuses}(Q)) \wedge$
$(\forall c \;:\; c \in \mathit{out}(P) \cup \mathit{out}(Q) \;:\; P/c \longleftrightarrow Q/c)\,.$

This can be proved in four steps, cf. the proof of Proposition 2.1.

Example 3.1. Consider connecting a Fork module to a Join module so that they form a closed system. The processes F and J defined in Examples 2.1 and 2.2 are suitable for modelling this situation since the input alphabet $\{a\}$ of F is the same as the output alphabet of J and the output alphabet $\{b, c\}$ of F is the same as the input alphabet of J. Since $\mathit{refuses}(F) \wedge \mathit{refuses}(J)$, the system can deadlock immediately and so $F \not\longleftrightarrow J$. Indeed, since $\mathit{out}(F) = \mathit{out}(J) = \emptyset$, the system can do nothing else!

Example 3.2. Successful interaction happens when the Fork module is initialised so that it is ‘hot’, i.e., $F/a? \longleftrightarrow J$.

Example 3.3. Suppose instead that the hot Fork module is connected to a Merge module. Using the process M defined in Example 2.3, we note that $b \in \mathit{out}(F/a?)$, $c \in \mathit{out}(F/a?/b!)$ and $\mathit{div}(M/b?/c?)$. That is, interference is possible and so $F/a? \not\longleftrightarrow M$.

We shall now establish a ‘trace-theoretic definition’ of successful interaction.

Lemma 3.1. $P \longleftrightarrow_i Q \;\equiv\; (\forall t \;:\; \#t \leq i \wedge t \in T[\![P]\!] \cap T[\![Q]\!] \;:\; t \notin D[\![P]\!] \cup D[\![Q]\!] \cup (F[\![P]\!] \cap F[\![Q]\!]))\,.$

Proof:
Induction on i.
Base case: 0. Then ε is the only value for t that satisfies $\#t \leq 0 \wedge t \in T[\![P]\!] \cap T[\![Q]\!]$. The result follows from $\varepsilon \in D[\![X]\!] \equiv \mathit{div}(X)$ and $\varepsilon \in F[\![X]\!] \equiv \mathit{refuses}(X)$ for any process X.
Inductive step: $i+1$.

$P \longleftrightarrow_{i+1} Q$

$\equiv$ { Definition of $\longleftrightarrow_{i+1}$ }
$P \longleftrightarrow_i Q \wedge (\forall c \,:\, c \in out(P) \cup out(Q) \,:\, P/c \longleftrightarrow_i Q/c)$
$\equiv$ { Induction hypothesis }
$(\forall t \,:\, \#t \leq i \wedge t \in T[\![P]\!] \cap T[\![Q]\!] \,:\, t \notin D[\![P]\!] \cup D[\![Q]\!] \cup (F[\![P]\!] \cap F[\![Q]\!])) \;\wedge$
$(\forall c \,:\, c \in out(P) \cup out(Q) \,:$
$(\forall t \,:\, \#t \leq i \wedge t \in T[\![P/c]\!] \cap T[\![Q/c]\!] \,:\, t \notin D[\![P/c]\!] \cup D[\![Q/c]\!] \cup (F[\![P/c]\!] \cap F[\![Q/c]\!])))$
$\equiv$ { Semantics of / }
$(\forall t \,:\, \#t \leq i+1 \wedge t \in T[\![P]\!] \cap T[\![Q]\!] \,:\, t \notin D[\![P]\!] \cup D[\![Q]\!] \cup (F[\![P]\!] \cap F[\![Q]\!]))\,.$

□

Theorem 3.1. $P \longleftrightarrow Q \;\equiv\; (\forall t \,:\, t \in T[\![P]\!] \cap T[\![Q]\!] \,:\, t \notin D[\![P]\!] \cup D[\![Q]\!] \cup (F[\![P]\!] \cap F[\![Q]\!]))\,.$

Proof:

$P \longleftrightarrow Q$
$\equiv$ { Definition of $\longleftrightarrow$ }
$(\forall i \,:\, 0 \leq i \,:\, P \longleftrightarrow_i Q)$
$\equiv$ { Lemma 3.1 }
$(\forall i \,:\, 0 \leq i \,:\, (\forall t : \#t \leq i \wedge t \in T[\![P]\!] \cap T[\![Q]\!] \,:\, t \notin D[\![P]\!] \cup D[\![Q]\!] \cup (F[\![P]\!] \cap F[\![Q]\!])))$
$\equiv$ { Traces are finite }
$(\forall t : t \in T[\![P]\!] \cap T[\![Q]\!] \,:\, t \notin D[\![P]\!] \cup D[\![Q]\!] \cup (F[\![P]\!] \cap F[\![Q]\!]))\,.$

□

Note that successful interaction cannot arise from one process outputting to the other forever because a non-divergent process can only output a finite number of times before requiring input. Thus, even if in general we were to admit processes with empty output alphabets, a necessary condition for successful interaction would be that $A \neq \emptyset$ and $B \neq \emptyset$.

The friends of a process that behaves like P or like P' are the friends of both P and P'.

Theorem 3.2. $P \sqcap P' \longleftrightarrow Q \;\equiv\; P \longleftrightarrow Q \wedge P' \longleftrightarrow Q\,.$

Proof:
This follows from Theorem 3.1, the semantics of $\sqcap$ and set theory. □

Corollary 3.1. $P' \sqsupseteq P \wedge P \longleftrightarrow Q \;\Rightarrow\; P' \longleftrightarrow Q\,.$

Finally, we investigate the relationship between successful interaction and parallel composition. The following 'trading' theorem states that the success of an interaction does not depend upon the side of the interaction to which parallel composition is applied.

Theorem 3.3. $P \,||\, Q \longleftrightarrow R \;\equiv\; Q \longleftrightarrow P \,||\, R\,.$

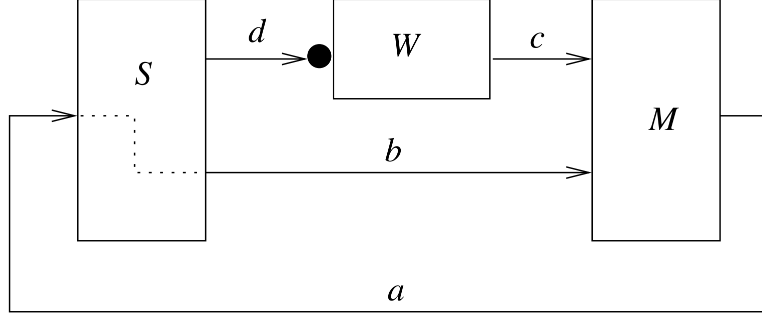

Figure 4. Infinite chatter between the Selector and Merge modules results in starvation of the Wire module.

Proof:
Let P, Q and R have alphabets C_0, C_1 and C_2, respectively. Then

$$\begin{aligned}
& P \,||\, Q \longleftrightarrow R \\
\equiv \quad & \{ \text{ Theorem 3.1 } \} \\
& (\forall t_2 \;:\; t_2 \in T[\![P \,||\, Q]\!] \cap T[\![R]\!] \;:\; t_2 \notin D[\![P \,||\, Q]\!] \cup D[\![R]\!] \cup (F[\![P \,||\, Q]\!] \cap F[\![R]\!])) \\
\equiv \quad & \{ \text{ Semantics of } || \text{ } \} \\
& (\forall t, t_0, t_1, t_2 \;:\; t \in (C_0 \cup C_1 \cup C_2)^* \wedge t_0 = t \restriction C_0 \wedge t_1 = t \restriction C_1 \wedge t_2 = t \restriction C_2 \wedge \\
& \qquad t_0 \in T[\![P]\!] \wedge t_1 \in T[\![Q]\!] \wedge t_2 \in T[\![R]\!] \;: \\
& \qquad\qquad t_0 \notin D[\![P]\!] \wedge t_1 \notin D[\![Q]\!] \wedge t_2 \notin D[\![R]\!] \wedge \\
& \qquad\qquad \neg(t_0 \in F[\![P]\!] \wedge t_1 \in F[\![Q]\!] \wedge t_2 \in F[\![R]\!]))\,.
\end{aligned}$$

The result follows by symmetry. □

Note that, in stating the theorem, we have implicitly assumed that there is no infinite chatter between P and Q and between P and R. The following example shows why this is necessary.

Example 3.4. Consider a closed system consisting of a Merge module (as in Example 2.3), a hot Wire module and a Selector module, Fig. 4. This situation is described by the processes M, $W_{d,c}/d?$, and $S_{a,b,d} \;=\; a? \;;\; b! \;;\; S_{a,b,d}$, i.e., the Selector module always communicates with the Merge module, ignoring the Wire module. As infinite chatter is possible between $S_{a,b,d}$ and M (after a transition on c), the process $S_{a,b,d} \,||\, M$ is not well formed. If we were to ignore the restriction on $||$, we would have *div*$((S_{a,b,d} \,||\, M)/c?)$, which would imply $W_{d,c}/d? \not\longleftrightarrow S_{a,b,d} \,||\, M$ since $c \in$ *out*$(W_{d,c}/d?)$. Yet $S_{a,b,d} \,||\, (W_{d,c}/d?) \;=\; c! \;;\; \mu X.\, a? \;;\; b! \;;\; X$ and so $S_{a,b,d} \,||\, (W_{d,c}/d?) \longleftrightarrow M$.

4. Controllability

In this section we formalise a new notion of controllability and investigate some of its properties. The idea, as we shall show in Section 5, is that controllability is a necessary and sufficient condition for a process to have a friend. We write $\updownarrow P$ to mean that P is controllable and $\not\updownarrow P$ to mean that it is

not. Controllability can be defined in terms of the limit as i tends to ∞ of its finite approximations $\updownarrow_i$ (controllable for at least i events), as follows:

$$\updownarrow P \;\equiv\; (\forall i \;:\; 0 \leq i \;:\; \updownarrow_i P)\,,$$

where a non-divergent process is controllable for 0 events,

$$\updownarrow_0 P \;\equiv\; \neg \mathit{div}(P)\,,$$

and a process is controllable for $i+1$ events if

(i) it is controllable for i events,

(ii) after any output event it is controllable for i events, and

(iii) either it cannot refuse to output or there is some input event after which it is controllable for i events,

$$\updownarrow_{i+1} P \;\equiv\; \updownarrow_i P \wedge (\forall b \;:\; b \in \mathit{out}(P) \;:\; \updownarrow_i(P/b!)) \wedge (\mathit{refuses}(P) \Rightarrow (\exists a \;:\; a \in A \;:\; \updownarrow_i(P/a?)))\,.$$

Example 4.1. A transparent latch is unsuitable as a building block for DI design because a transition that makes the latch opaque is not acknowledged. Nevertheless it can be used as a wire and so is controllable. More formally, a Latch module can be described by a process L with input alphabet $\{a, b\}$ and output alphabet $\{c\}$, defined by $L \;=\; [\, a? \rightarrow c!\; ;\; L \;\Box\; b? \rightarrow a?\; ;\; \bot \,]$. In the following table, we see that $\updownarrow_i$ quickly reaches a fixed point; in particular, $\updownarrow L$, $\updownarrow(L/a?)$ and $\not\updownarrow(L/b?)$. So, to control the module, the environment should alternate between sending a transition on a and receiving a transition on c.

Process			*div*	*out*	*refuses*	$\updownarrow_0$	$\updownarrow_1$	$\updownarrow_2$
L	=	$[\, a? \rightarrow c!\; ;\; L$						
		$\Box\; b? \rightarrow a?\; ;\; \bot \,]$	false	$\emptyset$	true	true	true	true
$L/a?$	=	$[\, c! \rightarrow L \;\Box\; b? \rightarrow \bot \,]$	false	$\{c\}$	false	true	true	true
$L/b?$	=	$[\, a? \rightarrow \bot \;\Box\; b? \rightarrow \bot \,]$	false	$\emptyset$	true	true	false	false
$L/a?/a?$	=	$\bot$	true	$\{c\}$	true	false	false	false
$L/a?/b?$	=	$\bot$			(see above)			
$L/a?/c!$	=	L			(see above)			
$L/b?/a?$	=	$\bot$			(see above)			
$L/b?/b?$	=	$\bot$			(see above)			

As we did for refinement and successful interaction, we also provide a 'circular definition' of controllability.

Proposition 4.1.
$\updownarrow P \equiv \neg \mathit{div}(P) \wedge (\forall b \;:\; b \in \mathit{out}(P) \;:\; \updownarrow(P/b!)) \wedge (\mathit{refuses}(P) \Rightarrow (\exists a \;:\; a \in A \;:\; \updownarrow(P/a?)))\,.$

Proof:

$$
\begin{array}{ll}
& \updownarrow P \\
\equiv & \{ \text{ Definition of } \updownarrow \} \\
& (\forall i \; : \; 0 \leq i \; : \; \updownarrow_i P) \\
\equiv & \{ \text{ Splitting the range of } i \} \\
& \updownarrow_0 P \wedge (\forall i \; : \; 0 \leq i \; : \; \updownarrow_{i+1} P) \\
\equiv & \{ \text{ Definition of } \updownarrow_{i+1} \text{ and predicate calculus } \} \\
& \updownarrow_0 P \wedge (\forall b \; : \; b \in \mathit{out}(P) \; : \; (\forall i \; : \; 0 \leq i \; : \; \updownarrow_i (P/b!))) \; \wedge \\
& (\mathit{refuses}(P) \Rightarrow (\forall i \; : \; 0 \leq i \; : \; (\exists a \; : \; a \in A \; : \; \updownarrow_i (P/a?)))) \\
\equiv & \{ \; A \text{ is finite and } \updownarrow_{i+1} P \; \Rightarrow \; \updownarrow_i P \; \} \\
& \updownarrow_0 P \wedge (\forall b \; : \; b \in \mathit{out}(P) \; : \; (\forall i \; : \; 0 \leq i \; : \; \updownarrow_i (P/b!))) \; \wedge \\
& (\mathit{refuses}(P) \Rightarrow (\exists a \; : \; a \in A \; : \; (\forall i \; : \; 0 \leq i \; : \; \updownarrow_i (P/a?)))) \\
\equiv & \{ \text{ Definition of } \updownarrow_0 \text{ and } \updownarrow \} \\
& \neg \mathit{div}(P) \wedge (\forall b \; : \; b \in \mathit{out}(P) \; : \; \updownarrow (P/b!)) \wedge (\mathit{refuses}(P) \Rightarrow (\exists a \; : \; a \in A \; : \; \updownarrow (P/a?))) \, .
\end{array}
$$

□

As an immediate consequence, we have a 'circular definition' of uncontrollability:

Corollary 4.1.
$\not\updownarrow P \; \equiv \; \mathit{div}(P) \vee (\exists b \; : \; b \in \mathit{out}(P) \; : \; \not\updownarrow (P/b!)) \vee (\mathit{refuses}(P) \wedge (\forall a \; : \; a \in A \; : \; \not\updownarrow (P/a?)))\, .$

Another immediate consequence is that a controllable process remains so after outputting, though it might become uncontrollable after inputting:

Corollary 4.2. If $b \in \mathit{out}(P)$, then $\updownarrow P \; \Rightarrow \; \updownarrow (P/b!)\, .$

Indeed, a process that is controllable must be able to engage in some event and remain controllable:

Corollary 4.3. $\updownarrow P \; \Rightarrow \; (\exists c \; : \; c \in A \cup \mathit{out}(P) \; : \; \updownarrow (P/c))\, .$

Proof:
This follows from the fact that a process that cannot refuse to output must be able to output, $\neg \mathit{refuses}(P) \Rightarrow \mathit{out}(P) \neq \emptyset$. □

More generally, a process that is controllable must be able to engage in sequences of events of unbounded length and remain controllable:

Corollary 4.4. $\updownarrow P \; \Rightarrow \; (\forall i \; : \; 0 \leq i \; : (\exists t \; : \; t \in T[\![P]\!] \wedge \#t = i \; : \; \updownarrow (P/t)))\, .$

Proof:
Induction on i. The base case $i = 0$ is trivial. The inductive step follows from the previous corollary. □

Note that, if $\updownarrow (P/t)$ where $t \in A^*$, then $|A|$ is an upper bound on $\#t$ on account of transmission interference.

Controllability is preserved by refinement.

Theorem 4.1. $P' \sqsupseteq P \wedge \updownarrow P \;\Rightarrow\; \updownarrow P'$.

Proof:
By the definition of $\updownarrow$, it suffices to prove that $P' \sqsupseteq P \wedge \updownarrow P \;\Rightarrow\; \updownarrow_i P'$ for all i, $0 \leq i$. This can be done by induction on i, as follows.
Base case: 0.

$$
\begin{array}{ll}
 & P' \sqsupseteq P \wedge \updownarrow P \\
\Rightarrow & \quad \{ \text{ Proposition 4.1 } \} \\
 & P' \sqsupseteq P \wedge \neg \mathit{div}(P) \\
\Rightarrow & \quad \{ \text{ Proposition 2.1 } \} \\
 & \neg \mathit{div}(P') \\
\equiv & \quad \{ \text{ Definition of } \updownarrow_0 \} \\
 & \updownarrow_0 P' .
\end{array}
$$

Inductive step: $i + 1$.

$$
\begin{array}{ll}
 & P' \sqsupseteq P \wedge \updownarrow P \\
\equiv & \quad \{ \text{ Proposition 4.1 } \} \\
 & P' \sqsupseteq P \wedge \updownarrow P \wedge (\forall b \;:\; b \in \mathit{out}(P) \;:\; \updownarrow (P/b!)) \wedge (\mathit{refuses}(P) \Rightarrow (\exists a \;:\; a \in A \;:\; \updownarrow (P/a?))) \\
\Rightarrow & \quad \{ \text{ Proposition 2.1 and induction hypothesis } \} \\
 & \updownarrow_i P' \wedge (\forall b \;:\; b \in \mathit{out}(P') \;:\; \updownarrow_i (P'/b!)) \wedge (\mathit{refuses}(P') \Rightarrow (\exists a \;:\; a \in A \;:\; \updownarrow_i (P'/a?))) \\
\equiv & \quad \{ \text{ Definition of } \updownarrow_{i+1} \} \\
 & \updownarrow_{i+1} P' .
\end{array}
$$

□

An uncontrollable process remains so after inputting, though it might become controllable after outputting:

Corollary 4.5. If $a \in A$, then $\not\updownarrow P \;\Rightarrow\; \not\updownarrow (P/a?)$.

Proof:
If P is divergent, then so is $P/a?$ and so $\not\updownarrow (P/a?)$. Otherwise, let $t \in B^* \cap T[\![P]\!]$ be a maximal sequence for which $\not\updownarrow (P/t)$ — there must be at least one such trace since $\not\updownarrow P$ and $\neg \mathit{div}(P)$. It must then be the case that $\not\updownarrow (P/t/a?)$. Since $P/a?/t \sqsubseteq P/t/a?$, we deduce that $\not\updownarrow (P/a?/t)$; $\not\updownarrow (P/a?)$ follows because a controllable process remains so after outputting. □

5. Reflection

The reflection $\sim P$ of a controllable process P is intended to be the most nondeterministic process that can successfully interact with it. Thus the input alphabet of $\sim P$ is the output alphabet B of P, and the

output alphabet of $\sim P$ is the input alphabet A of P. We propose the following recursive definition of the reflection of a process (whether controllable or not):

$$
\begin{aligned}
\sim P \;=\;& [\ (b\ :\ b \in B \setminus \mathit{out}(P)\ :\ b? \to \bot) && (1)\\
& \Box\,(b\ :\ b \in \mathit{out}(P) \wedge \mathit{refuses}(P)\ :\ b? \to \sim(P/b!)) && (2)\\
& \Box\,(\neg \mathit{refuses}(P)\ :\ \mathit{skip} \to [\ (b\ :\ b \in \mathit{out}(P)\ :\ b? \to \sim(P/b!))\ ]) && (3)\\
& \Box\,(a\ :\ a \in A \wedge \updownarrow(P/a?)\ :\ a! \to \sim(P/a?)) && (4)\\
&]
\end{aligned}
$$

Any input to $\sim P$ that P cannot initially output leads to undefined behaviour; hence, Clause (1) above. All other inputs are handled according to Clauses (2) and (3), as appropriate. The significance of the $skip$-guarded quiescent process is that we want to allow $\sim P$ to refuse to output, since P cannot, in order to obtain the most general (i.e., nondeterministic) process possible. Finally, $\sim P$ is at liberty to output anything that leaves P controllable, Clause (4).

To show how this works in practice, we reflect the descriptions of Wire, Fork, Join, Merge, Undetermined Selector [28] and Latch modules.

Example 5.1. Recall that $W_{a,b} \;=\; a?\ ;\ b!\ ;\ W_{a,b}$. Then

$$
\begin{aligned}
& \sim W_{a,b}\\
=\;& \quad \{\ \mathit{out}(W_{a,b}) = \emptyset,\ \mathit{refuses}(W_{a,b}) \text{ and } \updownarrow(W_{a,b}/a?)\ \}\\
& [\ b? \to \bot \ \Box\ a! \to \sim(W_{a,b}/a?)\ ]\\
=\;& \quad \{\ \mathit{out}(W_{a,b}/a?) = \{b\},\ \neg\mathit{refuses}(W_{a,b}/a?),\ \not\updownarrow(W_{a,b}/a?/a?) \text{ and } W_{a,b}/a?/b! = W_{a,b}\ \}\\
& [\ b? \to \bot \ \Box\ a! \to [\ \mathit{skip} \to [\ b? \to \sim W_{a,b}\ ]\ ]\ ]\\
=\;& \quad \{\ \text{Simplification using laws of DI-Algebra}\ \}\\
& a!\ ;\ b?\ ;\ \sim W_{a,b}\,.
\end{aligned}
$$

This is the description of a hot Wire module, i.e., $\sim W_{a,b} \;=\; W_{b,a}/b?$.

Example 5.2. Recall the process $F \;=\; a?\ ;\ b!\ ;\ c!\ ;\ F$, analysed in Example 2.1. Then

$$
\begin{aligned}
& \sim F\\
=\;& \quad \{\ \mathit{out}(F) = \emptyset,\ \mathit{refuses}(F) \text{ and } \updownarrow(F/a?)\ \}\\
& [\ b? \to \bot \ \Box\ c? \to \bot \ \Box\ a! \to \sim(F/a?)\ ]\\
=\;& \quad \{\ \mathit{out}(F/a?) = \{b, c\},\ \neg\mathit{refuses}(F/a?) \text{ and } \not\updownarrow(F/a?/a?)\ \}\\
& [\ b? \to \bot \ \Box\ c? \to \bot \ \Box\ a! \to [\ \mathit{skip} \to [\ b? \to \sim(F/a?/b!) \ \Box\ c? \to \sim(F/a?/c!)\ ]\ ]\ ]\\
=\;& \quad \{\ \mathit{out}(F/a?/b!) = \{c\},\ \neg\mathit{refuses}(F/a?/b!),\ \not\updownarrow(F/a?/b!/a?) \text{ and } F/a?/b!/c! = F\text{; likewise}\\
& \quad\ \ \text{for } F/a?/c!\ \}\\
& [\ b? \to \bot \ \Box\ c? \to \bot \ \Box\ a! \to [\ \mathit{skip} \to \ [\ b? \to [\ b? \to \bot \ \Box\ \mathit{skip} \to [\ c? \to \sim F\ ]\ ]\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Box\ c? \to [\ c? \to \bot \ \Box\ \mathit{skip} \to [\ b? \to \sim F\ ]\ ]\ ]\ ]\ ]\ ]\\
=\;& \quad \{\ \text{Simplification using laws of DI-Algebra}\ \}
\end{aligned}
$$

$$[\ b? \to \bot \ \square\ c? \to \bot \ \square\ a! \to b?\ ;\ c?\ ;\ {\sim}F\ ]\ .$$

This is the description of a two-hot Join module, i.e., ${\sim}F \ = \ J/b?/c?$, as in Example 2.2.

Example 5.3. Reflecting the process $J \ = \ b?\ ;\ c?\ ;\ a!\ ;\ J$, we derive

$$
\begin{array}{ll}
 & {\sim}J \\
= & \quad \{\ \mathit{out}(J) = \emptyset,\ \mathit{refuses}(J),\ \updownarrow(J/b?) \text{ and } \updownarrow(J/c?)\ \ \} \\
 & [\ a? \to \bot \ \square\ b! \to {\sim}(J/b?) \ \square\ c! \to {\sim}(J/c?)\ ] \\
= & \quad \{\ \mathit{out}(J/b?) = \emptyset,\ \mathit{refuses}(J/b?),\ \not\updownarrow(J/b?/b?) \text{ and } \updownarrow(J/b?/c?)\ ;\text{ likewise for } J/c?\ \} \\
 & [\ a? \to \bot \\
 & \square\ b! \to [\ a? \to \bot \ \square\ c! \to {\sim}(J/b?/c?)\ ] \\
 & \square\ c! \to [\ a? \to \bot \ \square\ b! \to {\sim}(J/c?/b?)\ ]\ ] \\
= & \quad \{\ \mathit{out}(J/b?/c?) = \{a\},\ \neg\mathit{refuses}(J/b?/c?),\ J/b?/c?/a! = J, \text{ and } P/c?/d? = P/d?/c? \text{ and} \\
 & \quad\quad \not\updownarrow(P/c?/c?) \text{ for any process } P \text{ and input signals } c \text{ and } d\ \ \} \\
 & [\ a? \to \bot \\
 & \square\ b! \to [\ a? \to \bot \ \square\ c! \to [\ skip \to [\ a? \to {\sim}J\ ]\ ]\ ] \\
 & \square\ c! \to [\ a? \to \bot \ \square\ b! \to [\ skip \to [\ a? \to {\sim}J\ ]\ ]\ ]\ ] \\
= & \quad \{\ \text{Simplification using laws of DI-Algebra}\ \ \} \\
 & b!\ ;\ c!\ ;\ a?\ ;{\sim}J\ .
\end{array}
$$

This is the description of a hot Fork module, i.e., ${\sim}J \ = \ F/a?$.

Example 5.4. Recall the process $M \ = \ [\ b? \to a!\ ;\ M \ \square\ c? \to a!\ ;\ M\ ]$, analysed in Example 2.3. Then

$$
\begin{array}{ll}
 & {\sim}M \\
= & \quad \{\ \mathit{out}(M) = \emptyset,\ \mathit{refuses}(M),\ \updownarrow(M/b?) \text{ and } \updownarrow(M/c?)\ \ \} \\
 & [\ a? \to \bot \ \square\ b! \to {\sim}(M/b?) \ \square\ c! \to {\sim}(M/c?)\ ] \\
= & \quad \{\ \mathit{out}(M/b?) = \{a\},\ \neg\mathit{refuses}(M/b?),\ \not\updownarrow(M/b?/b?),\ \not\updownarrow(M/b?/c?) \text{ and } M/b?/a! = M;\text{ like-} \\
 & \quad\quad \text{wise for } M/c?\ \} \\
 & [\ a? \to \bot \\
 & \square\ b! \to [\ skip \to [\ a? \to {\sim}M\ ]\ ] \\
 & \square\ c! \to [\ skip \to [\ a? \to {\sim}M\ ]\ ]\ ] \\
= & \quad \{\ \text{Simplification using laws of DI-Algebra}\ \ \} \\
 & (b!\ ;\ a?\ ;{\sim}M) \sqcap (c!\ ;\ a?\ ;{\sim}M)\ .
\end{array}
$$

This is the description of a hot Undetermined Selector module, i.e., ${\sim}M \ = \ U/a?$ where $U \ = \ a?\ ;\ ((b!\ ;\ U) \sqcap (c!\ ;\ U))$.

Example 5.5. Reflecting the process U above, we derive

$${\sim}U$$

$= \quad \{\ \mathit{out}(U) = \emptyset,\ \mathit{refuses}(U) \text{ and } \updownarrow(U/a?)\ \}$

$[\ b? \to \bot \ \Box\ c? \to \bot \ \Box\ a! \to \sim(U/a?)\ ]$

$= \quad \{\ \mathit{out}(U/a?) = \{b, c\},\ \neg\mathit{refuses}(U/a?),\ \not\updownarrow(U/a?/a?) \text{ and } U/a?/b!\ =\ U/a?/c!\ =\ U\ \}$

$[\ b? \to \bot \ \Box\ c? \to \bot \ \Box\ a! \to [\ \mathit{skip} \to [\ b? \to \sim U \ \Box\ c? \to \sim U\ ]\ ]\ ]$

$= \quad \{$ Simplification using laws of DI-Algebra $\}$

$a!\ ;\ [\ b? \to \sim U \ \Box\ c? \to \sim U\ ]\ .$

This is the description of a one-hot Merge module, i.e., $\sim U\ =\ M/b?\ =\ M/c?$.

Example 5.6. Recall the process $L\ =\ [\ a? \to c!\ ;\ L \ \Box\ b? \to a?\ ;\ \bot\ ]$, analysed in Example 4.1. Then

$\sim L$

$= \quad \{\ \mathit{out}(L) = \emptyset,\ \mathit{refuses}(L),\ \updownarrow(L/a?) \text{ and } \not\updownarrow(L/b?)\ \}$

$[\ c? \to \bot \ \Box\ a! \to \sim(L/a?)\ ]$

$= \quad \{\ \mathit{out}(L/a?) = \{c\},\ \neg\mathit{refuses}(L/a?),\ \not\updownarrow(L/a?/a?),\ \not\updownarrow(L/a?/b?) \text{ and } L/a?/c! = L\ \}$

$[\ c? \to \bot \ \Box\ a! \to [\ \mathit{skip} \to [\ c? \to \sim L\ ]\ ]\ ]$

$= \quad \{$ Simplification using laws of DI-Algebra $\}$

$a!\ ;\ c?\ ;\ \sim L\ .$

This is the description of a hot Selector module, i.e., $\sim L\ =\ S_{c,a,b}/c?$, as in Example 3.4.

We shall now characterise the observable behaviour of $\sim P$.

Lemma 5.1. For any $a \in A$, $a \notin \mathit{out}(\sim(P/a?))$.

Proof:
First note that $t \in A^*$ and $\#t > |A|$ implies $\mathit{div}(P/t)$ because of transmission interference. Then derive

$a \in \mathit{out}(\sim(P/t/a?))$

$\equiv \quad \{$ Definitions of $\sim$ and *out* $\}$

$\updownarrow(P/t/a?/a?) \vee (\exists a'\ :\ a' \in A \wedge \updownarrow(P/t/a?/a'?)\ :\ a \in \mathit{out}(\sim(P/t/a?/a'?)))$

$\Rightarrow \quad \{\ \mathit{div}(Q/a?/a?),\ \mathit{div}(Q) \Rightarrow \not\updownarrow Q \text{ and } Q/a?/a'? = Q/a'?/a? \text{ for any } Q\ \}$

$(\exists a'\ :\ a' \in A \wedge \neg\mathit{div}((P/ta')/a?)\ :\ a \in \mathit{out}(\sim((P/ta')/a?)))\ .$

Thus, starting with t being the empty sequence, we would need to be able to extend it indefinitely with input signals such that $\neg\mathit{div}((P/t)/a?)$, but this is impossible as we have noted. Thus, we have proved by contradiction that $a \notin \mathit{out}(\sim(P/a?))$. □

First we consider *div* and *refuses*.

Proposition 5.1. $\neg\mathit{div}(\sim P)$.

Lemma 5.2. $\mathit{refuses}(\sim P)\ \equiv\ \neg\mathit{refuses}(P) \vee (\forall a\ :\ a \in A\ :\ \not\updownarrow(P/a?))$.

This simplifies when P is controllable.

Proposition 5.2. If $\updownarrow P$, then *refuses*$(\sim P) \;\equiv\; \neg$*refuses*(P) .

Next we consider *out* .

Proposition 5.3. *out*$(\sim P) \;=\; \{\, a \;:\; a \in A \wedge \updownarrow(P/a?) \;:\; a \,\}$.

Proof:
It suffices to prove that $a \in$ *out*$(\sim(P/a'?)) \Rightarrow \updownarrow(P/a?)$.

$$
\begin{array}{ll}
 & a \in \mathit{out}(\sim(P/a'?)) \\
\equiv & \quad \{ \text{ Definitions of } \sim \text{ and } \mathit{out} \ \} \\
 & \updownarrow(P/a'?/a?) \vee (\exists a'' \;:\; a'' \in A \wedge \updownarrow(P/a'?/a''?) \;:\; a \in \mathit{out}(\sim(P/a'?/a''?))) \\
\Rightarrow & \quad \{ \text{ A process cannot input indefinitely and remain controllable } \} \\
 & (\exists t \;:\; t \in A^* \wedge a \in \mathit{out}(\sim(P/t)) \;:\; \updownarrow(P/t/a?)) \\
\Rightarrow & \quad \{ \ t \in A^* \text{ implies } ta \ltimes at \text{ and so } P/t/a? \sqsubseteq P/a?/t, \text{ Theorem 4.1 and Corollary 4.5 } \} \\
 & \updownarrow(P/a?) \,.
\end{array}
$$

□

Finally, we consider / .

Proposition 5.4. If $a \in$ *out*$(\sim P)$, then $\sim P/a! \;=\; \sim(P/a?)$.

Proposition 5.5. $\sim P/b? = \begin{cases} \sim(P/b!) & \text{, if } b \in \mathit{out}(P) \\ \bot & \text{, if } b \in B \setminus \mathit{out}(P)\,. \end{cases}$

We now consider a special case that gives the reflection of a process-after-input for free.

Corollary 5.1. If $\sim P \;=\; a!\,;\; Q$, then $\sim(P/a?) \;=\; Q$.

Proof:
$a \in$ *out*$(\sim P)$ and $\sim P/a! \;=\; Q$. The result then follows from Proposition 5.4. □

Example 5.7. We previously calculated that

$$
\begin{array}{rcl}
\sim W_{a,b} & = & a!\,;\; b?\,;\; \sim W_{a,b} \\
\sim J & = & b!\,;\; c!\,;\; a?\,;\; \sim J \\
\sim U & = & a!\,;\; [\, b? \rightarrow \sim U \;\Box\; c? \rightarrow \sim U \,] \\
\sim L & = & a!\,;\; c?\,;\; \sim L\,.
\end{array}
$$

To reflect the descriptions of hot Wire, one-hot Join, two-hot Join, hot Undetermined Selector and hot Latch modules, we simply apply Corollary 5.1. This reveals that

$$\begin{array}{rcl} \sim(W_{a,b}/a?) & = & b?\ ;\ \sim W_{a,b} \\ \sim(J/b?) & = & c!\ ;\ a?\ ;\ \sim J \\ \sim(J/b?/c?) & = & a?\ ;\ \sim J \\ \sim(U/a?) & = & [\ b? \rightarrow \sim U \ \Box\ c? \rightarrow \sim U\ ] \\ \sim L/a? & = & c?\ ;\ \sim L\,. \end{array}$$

These can be re-expressed as $W_{b,a}$, $F/a?/b!$, F, M and $S_{c,a,b}$, respectively.

Before we can establish that a necessary and sufficient condition for a process P to have friends is that it is controllable, and that $\sim P$ is the most nondeterministic process with which it can successfully interact, we need several lemmas. The first lemma states that controllability is a necessary condition for successful interaction.

Lemma 5.3. $P \longleftrightarrow Q \Rightarrow \updownarrow P$.

Proof:
It suffices to prove that $P \longleftrightarrow Q \Rightarrow \updownarrow_i P$ for all i, $0 \leq i$. This follows immediately from Proposition 3.1 by induction on i. □

The second lemma states that the friends of a process are the refinements of its reflection.

Lemma 5.4. $P \longleftrightarrow Q \ \Rightarrow\ Q \sqsupseteq \sim P$.

Proof:
It suffices to prove that $P \longleftrightarrow Q \Rightarrow Q \sqsupseteq_i \sim P$ for all i, $0 \leq i$. This can be done by induction on i, as follows.
Base case: 0.

$Q \sqsupseteq_0 \sim P$

$\equiv$ { Definition of $\sqsupseteq_0$ }

$(\mathit{div}(Q) \Rightarrow \mathit{div}(\sim P)) \wedge (\mathit{out}(Q) \subseteq \mathit{out}(\sim P)) \wedge (\mathit{refuses}(Q) \Rightarrow \mathit{refuses}(\sim P))$

$\Leftarrow$ { Propositions 5.1, 5.2 and 5.3 }

$\updownarrow P \wedge (\mathit{div}(Q) \Rightarrow \text{false}) \wedge (\mathit{out}(Q) \subseteq \{a \ :\ a \in A \wedge \updownarrow(P/a?) \ :\ a\}) \wedge (\mathit{refuses}(Q) \Rightarrow \neg \mathit{refuses}(P))$

$\Leftarrow$ { Lemma 5.3 and Proposition 3.1, in particular $a \in \mathit{out}(Q) \wedge P \longleftrightarrow Q \Rightarrow P/a? \longleftrightarrow Q/a!$ }

$P \longleftrightarrow Q$.

Inductive step: $i+1$.

$Q \sqsupseteq_{i+1} \sim P$

$\equiv$ { Definition of $\sqsupseteq_{i+1}$ }

$Q \sqsupseteq_i {\sim}P \wedge (\forall c \,:\, c \in B \cup \mathit{out}(Q) \,:\, Q/c \sqsupseteq_i {\sim}P/c)$

$\equiv$ { Propositions 5.4 and 5.5 }

$Q \sqsupseteq_i {\sim}P \wedge (\forall c \,:\, c \in \mathit{out}(P) \cup \mathit{out}(Q) \,:\, Q/c \sqsupseteq_i {\sim}(P/c)) \wedge (\forall b \,:\, b \in B \backslash \mathit{out}(P) \,:\, Q/b? \sqsupseteq_i \bot)$

$\Leftarrow$ { Proposition 3.1, induction hypothesis and all processes refine $\bot$ }

$P \longleftrightarrow Q$.

□

The third lemma states that a controllable process successfully interacts with its reflection.

Lemma 5.5. $\updownarrow P \;\Rightarrow\; P \longleftrightarrow {\sim}P$.

Proof:
It suffices to prove that $\updownarrow P \;\Rightarrow\; P \longleftrightarrow_i {\sim}P$ for all i, $0 \leq i$. This can be done by induction on i, as follows.
Base case: 0.

$P \longleftrightarrow_0 {\sim}P$

$\equiv$ { Definition of $\longleftrightarrow_0$ }

$\neg \mathit{div}(P) \wedge \neg \mathit{div}({\sim}P) \wedge \neg(\mathit{refuses}(P) \wedge \mathit{refuses}({\sim}P))$

$\Leftarrow$ { Propositions 4.1, 5.1 and 5.2 }

$\updownarrow P$.

Inductive step: $i+1$.

$P \longleftrightarrow_{i+1} {\sim}P$

$\equiv$ { Definition of $\longleftrightarrow_{i+1}$ }

$P \longleftrightarrow_i {\sim}P \wedge (\forall c \,:\, c \in \mathit{out}(P) \cup \mathit{out}({\sim}P) \,:\, P/c \longleftrightarrow_i {\sim}P/c)$

$\equiv$ { Propositions 5.3, 5.4 and 5.5 }

$P \longleftrightarrow_i {\sim}P \wedge (\forall c \,:\, c \in \mathit{out}(P) \cup \{a \,:\, a \in A \wedge \updownarrow(P/a?) \,:\, a\} \,:\, P/c \longleftrightarrow_i {\sim}(P/c))$

$\Leftarrow$ { Corollary 4.2 and induction hypothesis }

$\updownarrow P$.

□

Theorem 5.1. $P \longleftrightarrow Q \;\equiv\; \updownarrow P \wedge Q \sqsupseteq {\sim}P$.

Proof:
($\Rightarrow$) Lemmas 5.3 and 5.4.
($\Leftarrow$) Lemma 5.5 and Corollary 3.1. □

Since $\longleftrightarrow$ is symmetric, we have the following corollary.

Corollary 5.2. $\updownarrow P \wedge Q \sqsupseteq {\sim}P \;\equiv\; \updownarrow Q \wedge P \sqsupseteq {\sim}Q$.

Controllable processes can be ordered by $\sqsupseteq$ or by the converse ordering $\sqsubseteq$. The corollary tells us that reflection $\sim$ serves both as the lower adjoint F and the upper adjoint G in a Galois connection (F, G) between the two partially-ordered sets, i.e., $F(P) \sqsubseteq Q \;\equiv\; P \sqsupseteq G(Q)$.

6. Full abstraction

Reflection in DI-Algebra ignores uncontrollable behaviour after input events. Consequently, double reflection may weaken a controllable process, whereas in Verhoeff's XDI model it is the identity function.

Example 6.1. From Examples 5.1, 5.2, 5.4 and 5.7, we see that $W_{a,b}$, F and M are unchanged by double reflection, i.e., $\sim\sim W_{a,b} = W_{a,b}$, $\sim\sim F = F$ and $\sim\sim M = M$. On the other hand, $L = [\ a? \to c!\ ;\ L \,\Box\, b? \to a?\ ;\ \bot\ ]$ would be transformed into a less deterministic process — the $a?\ ;\ \bot$ changes to $\bot$ after double reflection.

Indeed, a standard property of Galois connections is a 'cancellation law', which for controllable processes takes the following form:

Theorem 6.1. $\updownarrow P \ \equiv\ \updownarrow\sim P \wedge P \sqsupseteq \sim\sim P$.

Proof:
Substitute $\sim P$ for Q in Corollary 5.2. $\Box$

Another standard property of Galois connections is that an adjoint is monotone:

Corollary 6.1. If $\updownarrow P$ and $P' \sqsupseteq P$, then $\sim P' \sqsubseteq \sim P$.

Proof:
$\updownarrow\sim P \wedge P \sqsupseteq \sim\sim P$ by Theorem 6.1, so $P' \sqsupseteq \sim\sim P$ by transitivity of $\sqsupseteq$. $\sim P' \sqsubseteq \sim P$ then follows by Corollary 5.2 (substituting $\sim P$ for P and P' for Q). $\Box$

Together these properties yield another standard 'cancellation law':

Corollary 6.2. If $\updownarrow P$, then $\sim\sim\sim P = \sim P$.

Proof:
$\updownarrow\sim P \wedge P \sqsupseteq \sim\sim P$ by Theorem 6.1. $\updownarrow\sim\sim P \wedge \sim P \sqsupseteq \sim\sim\sim P$ by Theorem 6.1 (substituting $\sim P$ for P). Moreover, $\sim P \sqsubseteq \sim\sim\sim P$ by Corollary 6.1 (substituting P for P' and $\sim\sim P$ for P). $\Box$

We are now ready to see how double reflection affects testing. Firstly, a controllable process has the same friends as its double reflection.

Theorem 6.2. If $\updownarrow P$, then $P \longleftrightarrow Q \ \equiv\ \sim\sim P \longleftrightarrow Q$.

Proof:
By Theorem 6.1, $\updownarrow\sim P$ and so $\updownarrow\sim\sim P$.

$\sim\sim P \longleftrightarrow Q$
$\equiv$ { Theorem 5.1, given $\updownarrow\sim\sim P$ }
$Q \sqsupseteq \sim\sim\sim P$
$\equiv$ { Corollary 6.2 }

$$
\begin{array}{ll}
& Q \sqsupseteq {\sim}P \\
\equiv & \quad \{ \text{ Theorem 5.1, given } \updownarrow P \ \} \\
& P \longleftrightarrow Q \,.
\end{array}
$$

□

Secondly, controllable processes with the same double reflection have the same friends, and vice versa.

Theorem 6.3. If $\updownarrow P$ and $\updownarrow P'$, then ${\sim}{\sim}P = {\sim}{\sim}P'$ if and only if $P \longleftrightarrow Q \ \equiv\ P' \longleftrightarrow Q$ for all Q.

Proof:
(only if) We assume $\updownarrow P$, $\updownarrow P'$ and ${\sim}{\sim}P = {\sim}{\sim}P'$.

$$
\begin{array}{ll}
& P \longleftrightarrow Q \\
\equiv & \quad \{ \text{ Theorem 6.2 } \} \\
& {\sim}{\sim}P \longleftrightarrow Q \\
\equiv & \quad \{ \text{ Assumption } \} \\
& {\sim}{\sim}P' \longleftrightarrow Q \\
\equiv & \quad \{ \text{ Theorem 6.2 } \} \\
& P' \longleftrightarrow Q \,.
\end{array}
$$

(if) We assume $\updownarrow P$, $\updownarrow P'$ and $P \longleftrightarrow Q \ \equiv\ P' \longleftrightarrow Q$ for all Q. Then $P \longleftrightarrow Q \ \equiv\ Q \sqsupseteq {\sim}P'$ for all Q by Theorem 5.1. Put $Q = {\sim}P$ and apply Lemma 5.5 to obtain ${\sim}P \sqsupseteq {\sim}P'$. By symmetry, ${\sim}P' \sqsupseteq {\sim}P$ also. This implies ${\sim}P = {\sim}P'$ and so ${\sim}{\sim}P = {\sim}{\sim}P'$. □

We say that a controllable process P satisfying ${\sim}{\sim}P = P$ is *fully abstract* under testing-by-interaction. This makes sense because, as an immediate consequence of Theorem 6.3, such processes can be distinguished from one another by testing.

Finally, we establish a 'trace-theoretic definition' of fully-abstract controllable processes. These processes remain controllable after any trace that is not a divergence:

Lemma 6.1.
$\updownarrow P \wedge {\sim}{\sim}P = P \ \equiv\ \varepsilon \notin D[\![P]\!] \wedge (\forall t \ :\ t \in T[\![P]\!] \setminus D[\![P]\!] \ :\ \updownarrow(P/t))\,.$

A fully-abstract controllable process is a non-divergent process that must become divergent upon reaching a state in which it can refuse to output after any more inputs:

Theorem 6.4.
$\updownarrow P \wedge {\sim}{\sim}P = P \ \equiv\ \varepsilon \notin D[\![P]\!] \wedge (\forall t \ :\ t \in C^* \wedge (\forall u \ :\ u \in A^* \ :\ tu \in F[\![P]\!]) \ :\ t \in D[\![P]\!])\,.$

Proof:
($\Rightarrow$) $\updownarrow P$ implies $\varepsilon \notin D[\![P]\!]$. Let $t \in C^*$ and assume that $(\forall u \ :\ u \in A^* \ :\ tu \in F[\![P]\!])$. Since u can be ε, $t \in F[\![P]\!]$, i.e., P/t can refuse to output. If P/t is controllable, then it must be able to receive an input and remain so. This can be repeated at most $|A|$ times before the process can no longer refuse to output, but that contradicts the assumption. Thus P/t must be uncontrollable and so divergent by Lemma 6.1

since P is fully abstract.
($\Leftarrow$) By Lemma 6.1, it suffices to prove (by induction on i) that $\varepsilon \notin D[\![P]\!] \wedge (\forall t \ : \ t \in C^* \wedge (\forall u \ : \ u \in A^* \ : \ tu \in F[\![P]\!]) \ : \ t \in D[\![P]\!]) \ \Rightarrow \ (\forall t \ : \ t \in T[\![P]\!] \setminus D[\![P]\!] \ : \ \updownarrow_i(P/t))$ for all i, $0 \leq i$.
Base case: 0. Immediate from definition of $\updownarrow_0$.
Inductive step: $i+1$.

$$
\begin{array}{ll}
& (\forall t \ : \ t \in T[\![P]\!] \setminus D[\![P]\!] \ : \ \updownarrow_{i+1}(P/t)) \\
\equiv & \{ \text{ Definition of } \updownarrow_{i+1} \ \} \\
& (\forall t \ : \ t \in T[\![P]\!] \setminus D[\![P]\!] \ : \\
& \quad \updownarrow_i(P/t) \wedge (\forall b \ : \ b \in \mathit{out}(P/t) \ : \ \updownarrow_i(P/t/b!)) \wedge (\mathit{refuses}(P/t) \Rightarrow (\exists a \ : \ a \in A \ : \ \updownarrow_i(P/t/a?))) \\
\Leftarrow & \{ \text{ Induction hypothesis, using } tb \in T[\![P]\!] \wedge t \notin D[\![P]\!] \Rightarrow tb \notin D[\![P]\!], \text{ and} \\
& \quad \neg(\forall u \ : \ u \in A^* \ : \ tu \in F[\![P]\!]) \text{ if } t \notin D[\![P]\!] \text{ and so } ta \notin D[\![P]\!] \text{ for some } a \in A \ \} \\
& \varepsilon \notin D[\![P]\!] \wedge (\forall t \ : \ t \in C^* \wedge (\forall u \ : \ u \in A^* \ : \ tu \in F[\![P]\!]) \ : \ t \in D[\![P]\!]) .
\end{array}
$$

□

7. Design by factorisation

Verhoeff's Factorisation Theorem, $P \,||\, X \sqsupseteq Q \ \equiv \ X \sqsupseteq {\sim}(P \,||\, {\sim}Q)$, has to be modified to take into account the need for controllability and full abstraction.

Theorem 7.1. $\updownarrow{\sim}Q \wedge P \,||\, X \sqsupseteq {\sim}{\sim}Q \ \equiv \ \updownarrow(P \,||\, {\sim}Q) \wedge X \sqsupseteq {\sim}(P \,||\, {\sim}Q)$.

Proof:

$$
\begin{array}{ll}
& \updownarrow{\sim}Q \wedge P \,||\, X \sqsupseteq {\sim}{\sim}Q \\
\equiv & \{ \text{ Theorem 5.1 } \} \\
& P \,||\, X \longleftrightarrow {\sim}Q \\
\equiv & \{ \text{ Theorem 3.3 } \} \\
& X \longleftrightarrow P \,||\, {\sim}Q \\
\equiv & \{ \text{ Theorem 5.1 } \} \\
& \updownarrow(P \,||\, {\sim}Q) \wedge X \sqsupseteq {\sim}(P \,||\, {\sim}Q) .
\end{array}
$$

□

Given a process Q, one must appreciate that the theorem only helps in the decomposition of ${\sim}{\sim}Q$ which may be weaker than Q itself; any functional requirements of Q that cannot be tested by interaction are ignored. One must also appreciate that this theorem gives a valid decomposition into submodules, described by P and X, only if $P \,||\, {\sim}Q$ is controllable and there is no possibility of infinite chatter between P and X, nor between P and ${\sim}Q$. Note also that $\updownarrow(P \,||\, {\sim}Q)$ implies $\updownarrow{\sim}(P \,||\, {\sim}Q)$ by Theorem 6.1, so $\updownarrow(P \,||\, {\sim}Q) \wedge X \sqsupseteq {\sim}(P \,||\, {\sim}Q)$ implies $\updownarrow X$ by Theorem 4.1.

Anyway, the first step in the decomposition of a module described by Q into submodules, one of which is described by P, is to construct the most general environment for the module; this is described

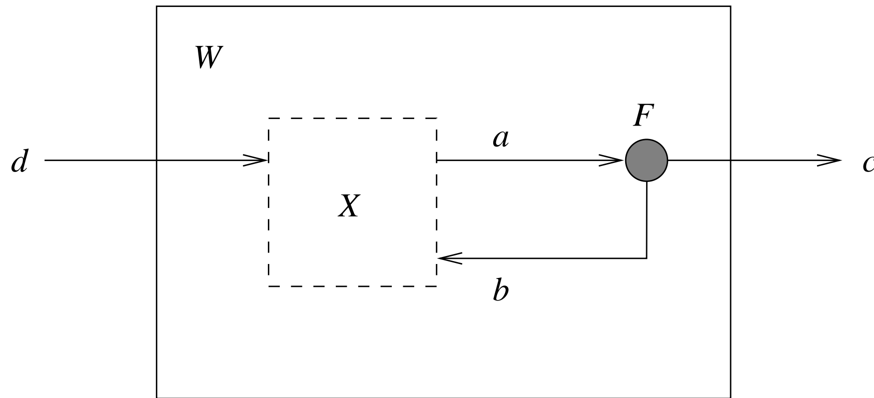

Figure 5. Decomposition of a Wire module using a Fork module.

by $\sim Q$. The second step is to check that there is no possibility of infinite chatter between P and $\sim Q$. The third step is to construct the parallel composition $P \,||\, \sim Q$ and to check that it is controllable. The fourth step is to construct the reflection $\sim(P \,||\, \sim Q)$ and to check that there is no possibility of infinite chatter with P; if infinite chatter is possible, then it may be possible to refine $\sim(P \,||\, \sim Q)$ so as to avoid it.

Note that sometimes we can determine just from their alphabets that there is no possibility of infinite chatter between two processes. This is the case when the output alphabet of one is disjoint from the input alphabet of the other since infinite chatter involves bidirectional communication. As Mallon has noted, it may also be possible to determine that there is no possibility of infinite chatter given just one of the processes since, for example, there can be no infinite chatter with (1) a Fork module if one of its outputs is to the environment, and (2) a Join module if one of its inputs is from the environment.

We now consider a series of examples that illustrate design by factorisation. To keep things simple, in all cases Q describes a Wire module.

Example 7.1. Our first example involves unidirectional communication so there is no danger of infinite chatter. Consider processes $W_{a,b}$ and $W_{a,c}$, representing two Wire modules. Suppose we want to find X such that $W_{a,c} \,||\, X \sqsupseteq W_{a,b}$. Clearly, X must have input alphabet $\{c\}$ and output alphabet $\{b\}$. We have previously determined that $\sim W_{a,b} \;=\; W_{b,a}/b?$, so $W_{a,c} \,||\, \sim W_{a,b} \;=\; (\sim W_{a,b})[c/a] \;=\; W_{b,c}/b?$. X is just the reflection of this process, i.e., $W_{c,b}$, which should not be a surprise. Indeed it is overkill to use the factorisation theorem here as one can immediately deduce that $X[a/c] \sqsupseteq W_{a,b}$.

Example 7.2. Our second example, previously considered by Lucassen [17], involves decomposition using a Fork module that outputs both to the submodule to be constructed and to the environment, Figure 5, so again there is no danger of infinite chatter. We next derive

$$
\begin{array}{ll}
 & F \,||\, \sim W_{d,c} \\
= & \quad \{ \text{ Example 5.1 } \} \\
 & F \,||\, (W_{c,d}/c?) \\
= & \quad \{ \text{ Trading law of DI-Algebra } \}
\end{array}
$$

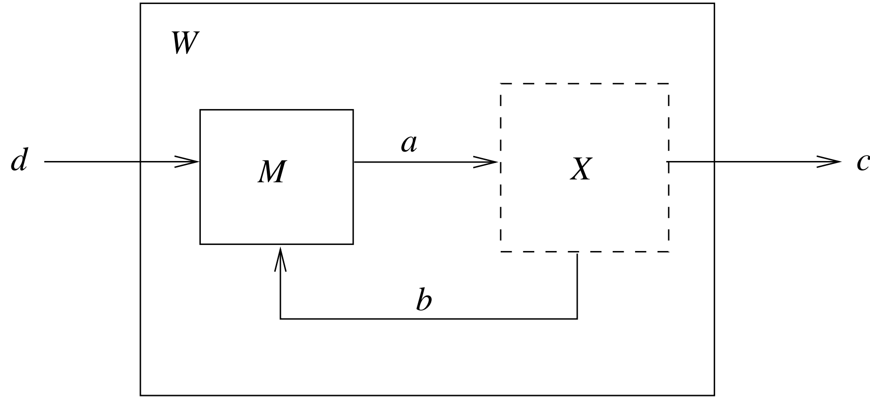

Figure 6. Decomposition of a Wire module using a Merge module.

$(c! \; ; \; F) \; || \; W_{c,d}$
$=$ { Example 2.1 }
$(F/a?/b!) \; || W_{c,d}$
$=$ { Property of renaming }
$(F/a?/b!)[d/c]$,

which is a controllable process. Since $J/b?$ is fully abstract and $\sim(J/b?) \;=\; F/a?/b!$ (as shown in Example 5.7), without any further work we can see that $X \;=\; \sim(F \; || \sim W_{d,c}) \;=\; (J/b?)[d/c]$. In other words, we require a one-hot Join module.

Example 7.3. Our third example involves decomposition using a Merge module that inputs both from the submodule to be constructed and from the environment, but only outputs to the submodule, Figure 6. In this case, we must take care to avoid the possibility of infinite chatter with the submodule.

$M[d/c] \; || \sim W_{d,c}$
$=$ { Example 5.1 }
$M[d/c] \; || \; (d! \; ; \; W_{c,d})$
$=$ { Trading law of DI-Algebra }
$(M[d/c]/d?) \; || \; W_{c,d}$
$=$ { Property of renaming }
$M/c?$,

which is a controllable process. Since U is fully abstract and $\sim U \;=\; M/c?$ (as shown in Example 5.5), without any further work we can see that $\sim(M[d/c] \; || \sim W_{d,c}) \;=\; U$. Unfortunately, infinite chatter is possible between $M[d/c]$ and U after an initial transition on d, so we cannot take X to be U. It is possible, however, to take X to be any refinement of U that places an upper bound on the number of times it can choose to output on b instead of on c. The simplest solution is $X \;=\; S_{a,c,b}$ that always chooses to output on c.

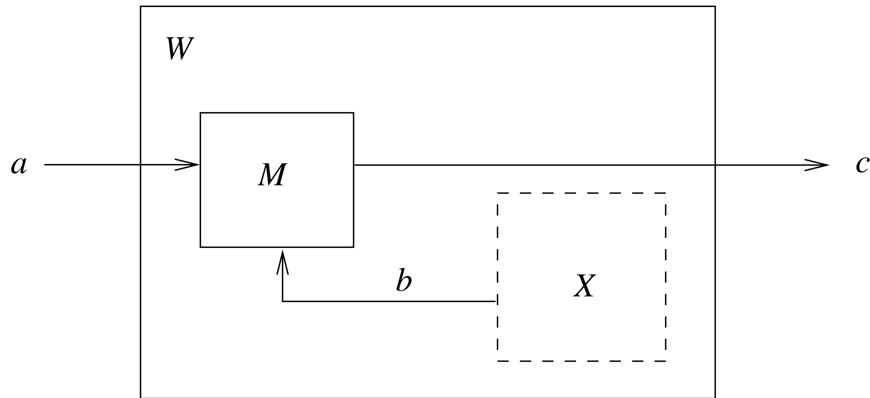

Figure 7. Alternative decomposition of a Wire module using a Merge module, but in this case X cannot be controllable.

Example 7.4. Our final example involves decomposition using a Merge module that inputs both from the submodule to be constructed and from the environment, but only outputs to the environment, Figure 7. In this case, infinite chatter will occur between M and $\sim W_{a,c}$, so the factorisation theorem is not applicable. Indeed it is obvious that no controllable solution exists for X since it has no input signals. Nevertheless, there is a solution to $M \mid\mid X \sqsupseteq W_{a,c}$, namely, $X = stop$.

8. Conclusion

We have formalised a new class of processes in DI-Algebra, the controllable processes. The notion of controllability has enabled us to define a reflection operator on processes. Processes in DI-Algebra are partially ordered by a refinement relation or by its converse; the reflection operator serves both as the lower adjoint and as the upper adjoint in a Galois connection between controllable processes under these orderings. Double reflection of a controllable process transforms it into a fully-abstract one: information is discarded about how the process will behave after an input arrives that does not keep it under control.

To motivate the above concepts and to underpin their formalisation, we have considered closed systems consisting of a pair of processes such that the outputs of one are the inputs of the other. We have defined sucessful interaction to mean that the system is free from interference and deadlock. This testing by interaction follows Verhoeff's approach, though he was more general, allowing systems to be constructed from any number of processes. We have proved that two processes successfully interact precisely when one is controllable and the other refines its reflection.

We have assumed a semantic model in which feasible sequences of events are called 'traces'; a trace is a 'failure' if the process can then refuse to output; a failure is a 'divergence' if the process is then divergent. Reflection transforms a process that can refuse to output into one that cannot, and vice versa. In contrast Verhoeff's semantic model labels sequences as 'bottom', 'demanding', 'indifferent', 'transient' and 'top'. We have not investigated the relationship between the two models in this paper, but we comment here that there was previously a consensus on Mallon's mapping from divergences to bottom, infeasible sequences to top, failures to indifferent, and all other traces to transient. Now, however, having worked with controllable processes, it seems more appropriate to map failures to demanding, so

maintaining consistency with the way Verhoeff defined reflection on labels (demanding and transient are reflections of each other) and with his definition of deadlock (which excludes all processes being in an indifferent state).

Our final contribution is to have modified Verhoeff's Factorisation Theorem so as to take controllability and full abstraction into account. In addition, we have been mindful of the fact, pointed out by Mallon, that the theorem is not valid if there is any possibility of infinite chatter between processes that have to be composed in parallel.

The above theory has been demonstrated on many examples that can be found throughout the paper. These examples are based upon a small set of modules that have been described in DI-Algebra. To apply the theory to more interesting design problems, it would be desirable to have tools, e.g., as previously developed by Mallon. One possibility is to configure a term-rewriting system to check for controllability, to perform reflection and to support design by factorisation. Indeed, Maude [3] has been previously applied to the algebraic manipulation of DI processes [15].

Currently we use the tool `di2pn` [11, 12] to translate DI processes into Petri nets. It acts as a front end to Cortadella's `petrify` tool [4] (for asynchronous logic synthesis) and Furey's `diana` tool [16] (for verification that one process refines another). Thus factorisation need only be attempted if `petrify` fails to synthesise a netlist from the corresponding Petri net. Since `petrify` works with closed systems, we have to supply `di2pn` with a pair of processes, specifying the module and its environment: one process could simply be the reflection of the other. Certainly, if design by factorisation is being attempted by hand, it would be a good idea to use `diana` to check the correctness of any proposed decomposition into submodules.

It would also be interesting to explore how the theory can be adapted to other classes of asynchronous processes, not just delay-insensitive ones, and not necessarily taking divergence into account.

Acknowledgements

The authors are grateful to the anonymous reviewers for their comments.

References

[1] Benko, I., Ebergen, J.: Composing Snippets, in: *Concurrency and Hardware Design Advances in Petri-Nets, LNCS* (J. Cortadella, A. Yakovlev, G. Rozenberg, Eds.), vol. 2549, Springer Verlag, 2002, 1–33.

[2] v. Bochmann, G.: Submodule Construction and Supervisory Control: A Generalization, *Proceedings of the 6th International Conference on Implementation and Application of Automata, LNCS*, **2494**, 2001, 27–39.

[3] Clavel, M., Eker, S., Lincoln, P., Meseguer, J.: Principles of Maude, *Electronic Notes in Theoretical Computer Science* (J. Meseguer, Ed.), 4, Elsevier Science Publishers, 2000.

[4] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers, *IEICE Transactions on Information and Systems*, **3**(E80-D), 1997, 315–325.

[5] Dill, D. L.: *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, ACM Distinguished Dissertations. MIT Press, 1989.

[6] Drissi, J., v. Bochmann, G.: Submodule Construction for Systems of I/O Automata, http://citeseer.ist.psu.edu/229830.html, 1999.

[7] Groenboom, R., Josephs, M. B., Lucassen, P. G., Udding, J. T.: Normal Form in a Delay-Insensitive Algebra, *Proceedings of the IFIP Transactions on Asynchronous Design Methodologies*, April 1993, 57–70.

[8] Hennessy, M.: *Algebraic Theory of Processes*, MIT Press, 1988.

[9] Hoare, C. A. R.: *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.

[10] Josephs, M. B.: Receptive Process Theory, *Acta Informatica*, **29**(1), 1992, 17–31.

[11] Josephs, M. B., Furey, D. P.: Delay-Insensitive Interface Specification and Synthesis, *Proceedings of Design, Automation and Test in Europe (DATE)*, March 2000, 169–173.

[12] Josephs, M. B., Furey, D. P.: A Programming Approach to the Design of Asynchronous Logic Blocks, in: *Concurrency and Hardware Design Advances in Petri-Nets, LNCS* (J. Cortadella, A. Yakovlev, G. Rozenberg, Eds.), vol. 2549, Springer Verlag, 2002, 34–60.

[13] Josephs, M. B., Hoare, C. A. R., He, J.: *A Theory of Asynchronous Processes*, Technical Report PRG-TR-6-89, Oxford University Computing Laboratory, Oxford, England, 1989.

[14] Josephs, M. B., Udding, J. T.: An Overview of D-I Algebra, *System Sciences, 1993, IEEE Proceeding of the Twenty-Sixth Hawaii International Conference*, **1**, January 1993, 329–338.

[15] Kapoor, H. K.: *Delay Insensitive Processes: A Formal Approach to the Design of Asynchronous Circuits*, Ph.D. Thesis, London South Bank University, UK, July 2004.

[16] Kapoor, H. K., Josephs, M. B., Furey, D. P.: Verification and Implementation of Delay-Insensitive Processes in Restrictive Environments, *Fundamenta Informaticae*, **70**(1-2), January 2006, 21–48.

[17] Lucassen, P. G.: *A Denotational Model and Composition Theorems for a Calculus of Delay-Insensitive Specifications*, Ph.D. Thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, May 1994.

[18] Lucassen, P. G., Polak, I., Udding, J. T.: Normal Form in DI-Algebra with Recursion, *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997, 167–174.

[19] Mallon, W. C.: *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*, Ph.D. Thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, January 2000.

[20] Mallon, W. C., Udding, J. T., Verhoeff, T.: Analysis and Applications of the XDI model, *Proceedings of Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'99)*, April 1999.

[21] Merlin, P., v. Bochmann, G.: On the Construction of Submodule Specifications and Communication Protocols, *ACM Transactions on Programming Languages and Systems*, **5**(1), January 1983, 1–25.

[22] Milner, R.: *A Calculus of Communicating Systems, LNCS*, vol. 92, Springer-Verlag, 1980.

[23] Molnar, C. E., Fang, T. P., Rosenberger, F. U.: Synthesis of Delay-Insensitive Modules, in: *Chapel Hill Conference on Very Large Scale Integration* (H. Fuchs, Ed.), Computer Science Press, 1985, 67–86.

[24] Negulescu, R.: General Testers for Asynchronous Circuits, *Proceedings of 10th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'04)*, April 2004.

[25] Nicola, R. D., Hennessy, M.: Testing equivalence for processes, *Theoretical Computer Science*, **34**, 1984, 83–133.

[26] Parrow, J.: Submodule Construction as Equation Solving in CCS, *Theoretical Computer Science*, **68**(2), October 1989, 175–202.

[27] Udding, J. T.: A Formal Model for Defining and Classifying Delay-Insensitive Circuits, *Distributed Computing*, **1**(4), 1986, 197–204.

[28] Verhoeff, T.: *A Theory of Delay-Insensitive Systems*, Ph.D. Thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994.

[29] Verhoeff, T.: Analyzing Specifications for Delay-Insensitive Circuits, *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, 172–183.

[30] Yakovlev, A. V., Koelmans, A. M., Semenov, A., Kinniment, D. J.: Modelling, Analysis and Synthesis of Asynchronous Control Circuits using Petri Nets, *Integration, the VLSI Journal*, **21**(3), December 1996, 143–170.