Two Implementation Relations and the Correctness of Communicating Replicated Processes

Maciej Koutny* Luigi V. Mancini† Giuseppe Pappalardo‡

Abstract

This paper studies the correctness of distributed systems made up of replicated processes that communicate by message passing. Processes are described within the divergence model of CSP. The notion of correctness introduced is based on a relation that formally expresses the conformance of an implementation process with the target process it is intended to implement. A weak and a strong such relations are introduced, aimed at treating acyclic and cyclic process networks respectively. Both allow the study of partial as well as total correctness of implementations, and may cope with non-deterministic targets and implementations.

We then show how a target process may be implemented (in the formal sense introduced) by replicating it in a set of copies, a majority of which is non-faulty.

Keywords: Fault-tolerance, distributed replicated systems, communicating sequential processes, nondeterminism, partial correctness, total correctness, divergence.

Contents

*Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU, U.K. Email: Maciej.Koutny@newcastle.ac.uk

†DISI, Università di Genova, 16132 Genova, Italy. Email: lvm@di.unipi.it

‡Università di Reggio Calabria, Italy. Email: pappalardo@iit.unict.it

1 Introduction

1.1 N-Modular Redundancy

One of the most commonly known fault-tolerance techniques is N-Modular Redundancy (NMR), a generalization of TMR (Triple MR) originally suggested by von Neumann for masking hardware faults. In NMR a computation is replicated in N instances (normally executed in parallel, by different processors). A reliable computation result is determined by applying a decision algorithm, e.g. majority voting, to the multiple results available. One of the advantages of NMR is that no explicit action needs to be performed for error detection and recovery.

Traditionally, NMR has been employed to replicate centralized computations. More recently, however, there has been considerable interest for distributed systems composed of Communicating Replicated Processes (CRP). CRP are advantageously employed in critical applications in which processing should not stop.

Distribution across remote sites introduces some entirely new problems into NMR. Replicating processes is not enough—communication channels have to be replicated too, for a single channel would be a bottleneck for performance and a weak link for reliability. But replicated communication, in the presence of arbitrary faults and nondeterminism due to distribution, may cause different replicas of a process to receive input messages that are different or in a different order. This is in contrast with the requirement that non-faulty replicas should process the same input sequences, in order to remain in a consistent state.

Many techniques for the construction of CRP are proposed in the literature. For example, a scheme for supporting CRP in a CSP framework, with synchronous communication and non-determinism, is described in [Man86]. Cooper [Coo85] discusses a solution in which remote procedure calls are used for replicated interprocess communication. A general ("state machine") approach, in which a CRP system is viewed as consisting of servers and clients, is proposed in [Sch90]. The implementation of CRP in the context of object-based system is tackled in [Bir85, LS90]. In particular, [LS90] discusses how CRP can be implemented in transaction systems; to ensure that replicas remain consistent, the scheme proposed employs reliable message delivery at low level and transaction ordering at high level.

1.2 Implementations, Specifications and Abstraction

The starting point of this paper is the definition of a rigorous correctness criterion for CRP—a question to which insufficient attention has been paid so far. This is of course an instance of a wider issue: defining the notion that the behaviour of a system satisfies a specification. In traditional approaches, the behaviour that must comply with the specification is exactly that observed at the system's interface. However, there are systems whose actual behaviour is better perceived and interpreted through some form of abstraction.

The simplest such abstraction is renaming, separately applied to each action (i.e. interaction instance) occurring at the system's interface; e.g., a CPU is more easily described by viewing data exchanged by registers as integers rather than bit sequences. More generally, in this work abstraction will aim to interpret the whole sequence of actions occurred; e.g. a serial line can be thought of as a channel conveying characters, the last of which is encoded by the last eight valid bits transmitted over the line. Another example is a replicated clock, which may be abstractly viewed as a single clock whose value is the median among the last available ones from each clock replica.

In such cases, specifications of actual system behaviour are apt to be less natural and simple than those of the abstracted behaviour. This is mainly due to the lower abstraction level involved, and may become worse if abstract behaviour has multiple actual realizations. Compare, e.g., the specification of a clock with that of a replicated clock whose replicas have varying relative speeds. We feel that a better approach is to retain specifications based on the abstract view of a system, and devise a notion of satisfaction of these specifications by the actual system behaviour, relative to the desired form of abstraction.

1.3 Fault Tolerant Systems

A fault tolerant system is regarded as acceptable even though it can be affected by faults, provided these stay within constraints known as *fault assumptions*. The choice of a fault assumption is of course motivated by the wish, and actual ability, to mask a certain class of faults. Following the ideas of Section 1.2, fault masking may be viewed as a form of abstraction according to which actual, possibly faulty system behaviour satisfies a specification for an abstract, ideal behaviour. Fault assumptions directly or indirectly[1] define the domain to which fault-maskable system behaviour belongs; it is only to this behaviour that it makes sense to apply the fault masking abstraction. Since fault assumptions and fault masking are closely tied, in our formalization they will be represented together, by a device called *extraction pattern*.

A most debated issue is whether the specification of a system should be given through a set of properties or through another system, whose behaviour serves as a standard to be reproduced or at least approximated. Here, since we are mainly interested in replicated systems, which in practice are obtained by assembling N replicas or copies of a *target* or *base* system, it seems quite natural to assume (the behaviour of) the target itself as the specification. As customary when the specification is itself

[1] E.g. the assumption that a minority of replicas is faulty constrains system behaviour indirectly.

a system, replicated systems that conform to the target will be said to implement it, and referred to as implementations.

1.4 The Implementation Relation

This work tackles the general problem of defining the *implementation relation* holding between a target system and systems that implement it according to some form of abstraction. Thus, owing to the role abstraction has in it, the implementation relation cannot be simply formalized with one of the observation equivalences or preorders in the literature (see [DN87] for a survey); on the contrary it should reflect the intended abstraction, through a dependence on extraction patterns. Within this approach, implementation by replication is catered for by devising an appropriate extraction pattern, aimed at formalizing the majority voting abstraction together with the fault assumption that a majority of replicas is non faulty. Other kinds of abstraction and fault assumptions may be dealt with just as easily: e.g., an appropriate extraction pattern will be introduced to model the *fail-stop* assumption, that any system is either non-faulty or inactive.

It is not easy, in our view, to indicate a unique implementation relation, which proves immediately acceptable to intuition. In fact, different relations may turn out to be useful for different applications. On the other hand, it is possible to state two natural, intuitively appealing properties that an implementation relation should possess. The first property, *accessibility*, ensures that the abstraction built into the implementation relation may be put to good use; in practice, this means that it should be possible to build a target system by plugging one of its implementations into an appropriate environment. In our treatment, this environment is made up of *disturbers*, feeding faulty but sufficiently redundant input, and *extractors*, which interpret the implementation's output. *Distributivity*, the second property required of the implementation relation, requires it to distribute over system composition; thus, a target composed of two connected systems may be implemented by connecting two of their respective implementations.

1.5 CSP and Base Systems

To complete the groundwork of our study, we must decide how system behaviour is to be formally rendered. We adopt the CSP failure-divergence model [BR85, Hoa85]. In this model, a system, or *process*, is described not only by the traces (sequences of actions) in which it may engage, but also by its failures, describing which actions it can refuse after a trace, and divergences, i.e. traces after which it may diverge (which essentially amounts to performing an unbounded sequence of internal actions). Traces alone provide a tool to study the partial (functional) correctness of processes. Failures and divergences add the ability to describe deadlock (internal nondeterminism) and divergence properties, which affords a means to reason about total correctness.

There is another aspect of CSP that makes it particularly suitable for our purposes. Defining abstraction and fault assumptions for system behaviour is much facilitated if its formal representation is firmly based on intuition. This is certainly true for the explicit failure-divergence model chosen; less so for models which view behaviour as an equivalence class (see [DN87] for examples), and may even depend on the set of process operators and axioms adopted [BW90].

CSP is also well-suited to describing the class of base systems considered in our treatment. This class comprises those systems whose interface may be conceptually divided into an input and an output part, and whose behaviour may be characterized by a relation between their input and output communications.

1.6 Outline of the Paper

The CSP failure-divergence model is presented in Section 2. Section 3 introduces extraction patterns as a tool for capturing abstraction and fault assumptions. Section 4 defines an implementation relation for functionally characterized base processes. This relation is shown to enjoy distributivity in Section 5, and accessibility in Section 7 (Section 6 deals with disturbers and extractors). Section 8 extends the notion of distributivity to process networks, including cyclic ones. Dealing with the latter requires widening the class of base systems considered (Section 9) and strengthening the implementation relation (Section 10). Section 11 studies NMR implementations. The last section presents some concluding remarks and discusses some related work.

2 A Brief Overview of CSP

2.1 Preliminaries

Communicating Sequential Processes (CSP) [BHR84, BR85, Hoa85] is a formal model for the description of concurrent computing systems. Informally, a CSP process can be regarded as a black box which may engage in interaction with its environment. Atomic instances of this interaction are called *actions* and must belong to the *alphabet* of the process. A *trace* of the process is a finite sequence of actions that it may engage in.

Actions and traces

Actions may be elements of any set, but, since we are concerned with networks of communicating processes, it is convenient to introduce structured actions of the form $c!v$, where v is a message and c is a communication channel. The action $c!v$ is said to occur at c and to cause v to be exchanged between processes communicating over c. For every channel c, let μc be the non-empty message set of c—the set of all v such that $c!v$ is a valid action. We define $\alpha c = \{c!v \mid v \in \mu c\}$ to be the alphabet of channel c, and $\alpha C = \bigcup_{c \in C} \alpha c$ to be the alphabet of a set of channels C.

The following notation is similar to that of [Hoa85]:

1. A^* is the set of all finite sequences of elements of the set A; if A contains actions, a sequence of A^* is said to be a trace over A (below t and u range over sequences).
2. $\langle\rangle$ is the empty sequence, $\langle a_1, \ldots, a_n\rangle$ is the sequence whose ith element is a_i.
3. $t \cdot u$ is obtained by appending u to t.
4. The length of t is denoted $|t|$.
5. $t \leq u$ holds true if t is a prefix of u.

Let now t be a trace, c be a channel and C be sets of channels. Then:

1. If b is a channel with $\mu b = \mu c$, then $(c!v)[b/c] = b!v$ and $(d!v)[b/c] = d!v$ for all $d \neq c$. $t[b/c]$ denotes the result of applying $[b/c]$ to every action in the trace t; for a set of actions A, $A[b/c]$ is defined likewise.
2. $t \lceil C$ is the trace obtained by projecting t on C, i.e. by deleting from t all the actions that do not occur on channels in C.
3. $t \backslash C$ is t with actions at channels in C deleted.
4. $t \downarrow c$ is obtained from $t \lceil \{c\}$ by deleting the $c!$ from each action.

For example, if $t = \langle c!1, d!2, c!3, e!6\rangle$ then $t \lceil \{c, e\} = \langle c!1, c!3, e!6\rangle$, $t \backslash \{c\} = \langle d!2, e!6\rangle$ and $t \downarrow c = \langle 1, 3\rangle$.

The four operators above can be extended to trace sets in the obvious way.

Proposition 2.1 Let $C \subseteq D$ be two sets of channels with αD finite. If $T \subseteq (\alpha D)^*$ is an infinite prefix-closed set of traces such that $T \backslash C$ is a finite set, then there exist $t \in T$ and $a_1, a_2, \ldots \in \alpha C$ satisfying $t \cdot \langle a_1, \ldots, a_k\rangle \in T$ for all $k \geq 1$.

Proof. Let $\mathcal{T}$ be a tree generating the traces in T and having no node with two outgoing edges labelled by the same action. Then $\mathcal{T}$ is infinite (like T) and finitely-branching (since αD is finite). Hence, by König's Lemma, $\mathcal{T}$ contains a path from the root generating an infinite sequence $\langle b_1, b_2, \ldots\rangle$. Since $T \backslash C$ is finite, there is $k \geq 0$ such that $b_{k+i} \in \alpha C$ for all $i \geq 1$. Hence the thesis holds for $t = \langle b_1, b_2, \ldots, b_k\rangle$ and $a_i = b_{k+i}$ for all $i \geq 1$. $\square$

Tuples of channels

A finite tuple of distinct channels like $(c_1, \ldots, c_n)$ may also be referred to with the notation $\vec{c}$. For brevity, the tuple (c) is also written c; and the tuple $\vec{c}$ is used where the set of its elements would be expected,[2] e.g. we write $x \in \vec{c}$ if $x \in \{c_1, \ldots, c_n\}$. The following notational devices will prove useful:

1. The set $dom_{\vec{c}}$ is defined as $(\mu c_1)^* \times \ldots \times (\mu c_n)^*$. The prefix relation is extended componentwise to sequence tuples of $dom_{\vec{c}}$.

2. If $\vec{c}$ and $\vec{e}$ are tuples of channels, a function $f\colon dom_{\vec{c}} \to dom_{\vec{e}}$ is monotonic if, for all $y \in dom_{\vec{c}}$ and $z \leq y$, $f(z) \leq f(y)$. Such functions will be qualified as "over $\vec{c}$ and $\vec{e}$", and their set will be denoted by $\mathcal{F}(\vec{c}, \vec{e})$.

3. Suppose $\vec{b} = (c_{n_1}, \ldots, c_{n_k})$ is a tuple of distinct channels from $\vec{c}$. Then, for $z = (z_1, \ldots, z_n) \in dom_{\vec{c}}$, we define $z \Downarrow \vec{b} = (z_{n_1}, \ldots, z_{n_k})$.

4. If t is a trace, then $t \downarrow \vec{c}$ stands for $(t \downarrow c_1, \ldots, t \downarrow c_n)$; note that $t \downarrow \vec{c} \in dom_{\vec{c}}$.

2.2 Processes

We use the failure-divergence model of CSP [Hoa85, BR85] in which a process P is identified with a triple $(\alpha P, \phi P, \delta P)$ where αP — *alphabet* — is a non-empty finite[3] set of actions, ϕP — *failures* — is a subset of $\alpha P^* \times 2^{\alpha P}$, and δP — *divergences* — is a subset of αP^*. The three components of P must satisfy the conditions given below, where $\tau P = \{t \mid (t, R) \in \phi P\}$ denotes the *traces* of P.

F1 $\langle\rangle \in \tau P$.

F2 $t \cdot s \in \tau P \;\Rightarrow\; t \in \tau P$.

F3 $(t, R) \in \phi P \wedge S \subseteq R \;\Rightarrow\; (t, S) \in \phi P$.

F4 $(t, R) \in \phi P \wedge a \in \alpha P \wedge t \cdot \langle a \rangle \notin \tau P \;\Rightarrow\; (t, R \cup \{a\}) \in \phi P$.

D1 $t \in D \wedge a \in \alpha P \;\Rightarrow\; t \cdot \langle a \rangle \in D$.

D2 $t \in D \wedge R \subseteq A \;\Rightarrow\; (t, R) \in \phi P$.

If $(t, R) \in \phi P$ then P is said to *refuse* R after t.[4] Intuitively, this means that P can deadlock if the environment offers it R as a set of possible events to be executed

[2] It will be clear from the context whether $\vec{c}$ is regarded as a set or a tuple.

[3] This is as in [Hoa85], and implies that the sets of channels, messages and actions considered in this paper are finite.

[4] Refusals are employed to model internal non-determinism in process behaviour.

after t. If $t \in \delta P$ then P is said to *diverge* after t. In the CSP approach this means the process behaves in a totally uncontrollable way.[5]

We will associate with P a set of channels, χP, and stipulate that the alphabet of P is that of χP. Thus, we shall be able to identify P with the triple $(\chi P, \phi P, \delta P)$ in lieu of $(\alpha P, \phi P, \delta P)$.

Operations on processes

For our purposes neither the syntax nor the semantics of the whole standard CSP is needed. Essential to our treatment are only the parallel composition of processes, hiding of communication over a set of channels, and channel renaming.

Parallel composition $P \parallel Q$ models synchronous communication between processes in such a way that each of them is free to engage independently in any action that is not in the other's alphabet, but they have to engage simultaneously in all actions that are in the intersection of their alphabets. Parallel composition is commutative and associative; we will use $\parallel(P_1, \ldots, P_n)$ to denote the parallel composition of processes $P_1, \ldots, P_n$.

Let P be a process and C be a set of channels of P; then $P \backslash C$ is a process that behaves like P with the actions occurring at the channels in C made invisible. Hiding is associative in that $(P \backslash C) \backslash D = P \backslash (C \cup D)$.

Renaming a channel c to b in a process P results in a new process $P[b/c]$ that behaves like P, with each action $c!v$ replaced by $b!v$.

The three operations on processes are formally defined in Table 1.

The next two results follow directly from the definitions of parallel composition and hiding.

Proposition 2.2 If $P = \parallel(P_1, \ldots, P_n)$ then:

(a) $\chi P = \chi P_1 \cup \ldots \cup \chi P_n$

(b) $\delta P = \{t \cdot s \in \alpha P^* \mid (\forall i : t \upharpoonright \chi P_i \in \tau P_i) \wedge (\exists j : t \upharpoonright \chi P_j \in \delta P_j)\}$

(c) $\phi P = \{(t, R_1 \cup \ldots \cup R_n) \mid t \in \alpha P^* \wedge (\forall i : (t \upharpoonright \chi P_i, R_i) \in \phi P_i)\} \cup \delta P \times 2^{\alpha P}$. □

Proposition 2.3 Let $(t, R) \in \phi(P \backslash C)$ and $t \notin \delta(P \backslash C)$. Then there is $(s, R \cup \alpha C) \in \phi P$ such that $s \notin \delta P$ and $s \backslash C = t$. Moreover, $(\tau P) \backslash C \subseteq \tau(P \backslash C)$. □

---

[5] The semantical treatment is based on what is often referred to as 'catastrophic' divergence; the process in a diverging state is modelled as being able to accept any trace and generate any refusal. This facilitates a smooth definition of fixpoints.

$$
\begin{array}{lcl}
\chi(P \,\|\, Q) & = & \chi P {\cup} \chi Q \\
\delta(P \,\|\, Q) & = & \{t{\cdot}s{\in}\alpha(P\|Q)^* \mid (t{\restriction}\chi P, t{\restriction}\chi Q){\in}(\tau P{\times}\delta Q){\cup}(\delta P{\times}\tau Q)\} \\
\phi(P \,\|\, Q) & = & \{(t, R{\cup}S) \mid t{\in}\alpha(P\|Q)^* \wedge (t{\restriction}\chi P, R) \in \phi P \wedge (t{\restriction}\chi Q, S){\in}\phi Q\} \cup \\
 & & \delta(P \,\|\, Q){\times}2^{\alpha(P \,\|\, Q)} \\
\hline
\chi(P \backslash C) & = & \chi P - C \\
\delta(P \backslash C) & = & \{(t \backslash C){\cdot}s \mid s \in \alpha(P \backslash C)^* \wedge (t{\in}\delta P \\
 & & \qquad \vee\ \exists a_1, a_2, \ldots {\in} \alpha C\ \forall n{\geq}1 : t{\cdot}\langle a_1, \ldots, a_n\rangle{\in}\tau P)\} \\
\phi(P \backslash C) & = & \{(t \backslash C, R - \alpha C) \mid (t, R{\cup}\alpha C){\in}\phi P\} \cup \delta(P \backslash C){\times}2^{\alpha(P \backslash C)} \\
\hline
\chi P[b/c] & = & (\chi P - \{c\}){\cup}\{b\} \\
\delta P[b/c] & = & \{t[b/c] \mid t{\in}\delta P\} \\
\phi P[b/c] & = & \{(t[b/c], R[b/c]) \mid (t, R){\in}\phi P\}
\end{array}
$$

Table 1: Parallel composition, hiding and renaming operators (C is a subset of χP; b and c are channels with the same message sets, $c{\in}\chi P$ and $b{\notin}\chi P$).

Process networks

Processes $P_1, \ldots, P_n$ ($n \geq 1$) form a network if no channel is shared by more than two of them. By $\#(P_1, \ldots, P_n)$ we denote the parallel composition of these processes with communication within the network hidden, i.e.:

$$\#(P_1, \ldots, P_n) \quad = \quad \|(P_1, \ldots, P_n) \backslash \bigcup_{i \neq j} (\chi P_i \cap \chi P_j).$$

Clearly, the order in the tuple $(P_1, \ldots, P_n)$ is immaterial for the definition of $\#$. We also let $\#(P)=P$ and $\#(P_1, P_2)=P_1 \# P_2$.

The next theorem follows easily from [BR85], Lemma 4. It shows that processes in a network may be composed (by $\#$) stepwise, and in arbitrary order.

Theorem 2.4 Let $P_1, \ldots, P_{n+m}$ ($m, n \geq 1$) be a network of processes. Then

$$\#(P_1, \ldots, P_n, P_{n+1}, \ldots, P_{n+m}) \; = \; \#(\#(P_1, \ldots, P_n), P_{n+1}, \ldots, P_{n+m}). \qquad \Box$$

Even though this property may be applied tacitly in the sequel, it is essential for a modular and hierarchical modelling and analysis of distributed systems. Note that it holds in the failure-divergence model of CSP, but not in the failure model [BHR84].

We will usually partition the channel set χP of a process P into the input channels $\imath P$ and output channels ωP. It will be implicitly assumed that no two processes in a network have a common input channel or a common output channel. The input (output) channels of the network will be those input (output) channels that belong only to one process in the network (shared channels are concealed by internal communication hiding).

The communication graph of the network $P_1, \ldots, P_n$ is defined to have n vertices and an edge from vertex i to vertex j if and only if $\omega P_i \cap \imath P_j \neq \emptyset$. A cyclic (acyclic) network is one with a cyclic (acyclic) communication graph.

3 Modelling Replicated Distributed Systems

In this section we aim at modelling the notion that a system is a correct implementation of another (base) system. Although this will be accomplished in a general fashion, abstracting from the implementation technique, we are mainly interested in replicated implementations, obtained by assembling multiple copies or *replicas* (some possibly faulty) of the base system (see Section 11). The base systems which will be considered in detail are functional processes generalizing the notion of pipe [BHR84, Hoa85].

Figure 1: Base processes and their implementations.

3.1 Extraction Patterns

The notion of *extraction pattern* is central to our approach, for it serves to characterise the communication of an implementation in a twofold sense: (1) by relating it to that of the corresponding base system, and (2) by describing the fault assumptions under which faults affecting it can be masked.

To explain the basic idea, we consider two process P and Q, which are intended to implement the base processes K and L respectively. K sends messages to L through a single channel c. In contrast, P and Q are not connected by a single channel, but rather by a set of channels *chan*, which they use to implement the communication carried out by K and L over c (see Figure 1).

With a view to establishing whether P and Q are correct implementations of K and L, we will relate communication over *chan* to that over c using an *extraction pattern* $ep{=}(chan, msg, wf, extr, sub, rub)$, whose components following *chan* have the intended meaning explained below.

- *msg* is the message set of c. We need neither assume that all the channels in *chan* have the same message sets, nor that any of them coincides with or is distinct from *msg*.

- *wf* is a set of *well-formed* traces over *chan*. A well-formed trace, though potentially faulty, is assumed to be sufficiently redundant to be interpreted as a trace over c. By choosing a suitable *wf* we can specify fault assumptions like the kind and number of faulty messages that can be tolerated by the fault-masking mechanism.

- *extr* is a mapping that provides an interpretation of the traces in *wf* as traces over the channel c. Intuitively, it *extracts* the information embedded in the traces of *wf*.

- *sub* and *rub* play the role of *upper bounds* for the refusals at *chan* of respectively the sender P and the receiver Q. If their respective refusals S and R are limited thus, P and Q are guaranteed not to deadlock ($S \cup R \neq \alpha chan$) by condition

EP4 below. This will be required whenever communication between P and Q is "incomplete" in a sense to be made precise later.

Definition 3.1 An extraction pattern is a tuple $(chan, msg, wf, extr, sub, rub)$ such that:

EP0 $chan$ is a finite, non-empty channel set.

EP1 msg is a message set.

EP2 $wf \subseteq (\alpha chan)^*$ is a non-empty, prefix-closed trace set.

EP3 $extr$ is a mapping with domain wf and codomain msg^*.

EP4 sub and rub are two mappings with domain wf such that, for every $t \in wf$, $sub(t)$ and $rub(t)$ are non-empty sets of subsets of $\alpha chan$,[6] and:

$$\text{if } S \in sub(t) \text{ and } R \in rub(t) \text{ then } S \cup R \neq \alpha chan.$$

If c is a channel with the message set msg then we will often refer to ep as an extraction pattern over c. □

Remark 3.2 The above definition of extraction pattern will be understood and used in a more general sense. With a slight abuse of notation, we introduce a superset of wf and still denote it by wf; i.e. for *any* trace t we write $t \in wf$ if $t \lceil chan \in wf$. Moreover, for any trace t such that $t \lceil chan \in wf$ (i.e. $t \in wf$ after extending the meaning of wf) and any set of actions R we denote:

$$\begin{array}{rcl} extr(t) & = & extr(t \lceil chan) \\ R \in rub(t) & \text{iff} & R \cap \alpha chan \in rub(t \lceil chan) \\ R \in sub(t) & \text{iff} & R \cap \alpha chan \in sub(t \lceil chan). \end{array}$$

In the sequel we often employ an extraction pattern family $ep_{\vec{c}}$ indexed over a channel tuple like $\vec{c} = (c_1, \ldots, c_n)$. In doing so we will assume that ep_x is an extraction pattern $(chan_x, msg_x, wf_x, extr_x, sub_x, rub_x)$ over each $x \in \vec{c}$. The notation introduced for single extraction patterns is generalized to $ep_{\vec{c}}$ in the following way:

$$\begin{array}{lcl} chan_{\vec{c}} & = & chan_{c_1} \cup \ldots \cup chan_{c_n} \\ wf_{\vec{c}} & = & wf_{c_1} \cap \ldots \cap wf_{c_n} \\ rub_{\vec{c}}(t) & = & rub_{c_1}(t) \cap \ldots \cap rub_{c_n}(t) \\ sub_{\vec{c}}(t) & = & sub_{c_1}(t) \cup \ldots \cup sub_{c_n}(t) \\ extr_{\vec{c}}(t) & = & (extr_{c_1}(t), \ldots, extr_{c_n}(t)). \end{array}$$

Thus, for example, $rub_{\vec{c}}(t) = \{R \mid \forall x \in \vec{c} : R \cap \alpha chan_x \in rub(t \lceil chan_x)\}$. Note that $rub_{\vec{c}}(t)$, $sub_{\vec{c}}(t)$ and $extr_{\vec{c}}(t)$ are only defined if $t \in wf_{\vec{c}}$.
Given the intended meaning of the wf_x's, $wf_{\vec{c}}$ comprises the traces whose projections on $\vec{c}$ are well formed. Note that $wf_{\vec{c}}$ is prefix-closed because each wf_x is. □

[6] That is, $sub(t)$ and $rub(t)$ are elements of $2^{2^{\alpha chan}} - \{\emptyset\}$.

Extraction pattern examples

To appreciate the following examples, it is necessary to point out that the base systems considered in this paper communicate asynchronously and therefore their channels never refuse any input. This property tends to be reflected by the *rub* set.

Our first example is a canonical extraction pattern which can be used, e.g., to show that a system is a correct implementation of itself. According to this pattern, message sequences exchanged over a singleton channel set *chan* are interpreted as themselves and *chan* refuses no input, like a base process channel. In Figure 1 this corresponds to the situation where $chan=\{c\}$.

The *canonical* extraction pattern for a channel c is defined to be the tuple

$$can_c = (\{c\}, \mu c, \alpha c^*, extr, sub, rub)$$

where for all $t\in\alpha c^*$:

CP1 $extr(t)=t\downarrow c$.

CP2 $sub(t)=\{S\subseteq\alpha c \mid S\neq\alpha c\}$.

CP3 $rub(t)=\{\emptyset\}$.

To understand the *sub–rub* pair, suppose P and Q of Figure 1 refuse S and R respectively at $chan=\{c\}$. Since no input should be refused at *chan*, R must be in $\{\emptyset\}=rub(t)$, and $S\in\{S\subseteq\alpha c \mid S\neq\alpha c\}=sub(t)$ is the weakest condition to avoid deadlock at c. Note that *sub* is maximal with respect to *rub* and condition EP4.

Another kind of extraction pattern may be used to model replicated systems employing fail-stop components, which either function correctly or do nothing [Sch84]. Let *chan* be a finite set of channels with a common message set *msg* and let $\emptyset\subset NF\subseteq chan$; channels in NF are supposed to belong to non-faulty system components and therefore to refuse no input, like base process channels. The *fail-stop* extraction pattern for *chan* and NF is $fs(chan, NF)=(chan, msg, wf, extr, sub, rub)$, where:

FS1 $wf=\{t\in(\alpha chan)^* \mid \forall b,d\in chan : (t\downarrow b\leq t\downarrow d) \vee (t\downarrow d\leq t\downarrow b)\}$.

FS2 $extr(t)$ is the longest string in $\{t\downarrow b \mid b\in chan\}$.

FS3 $sub(t)=\{S\subseteq\alpha chan \mid S\cap\alpha NF\neq\alpha NF\}$.

FS4 $rub(t)=\{R\subseteq\alpha chan \mid R\cap\alpha NF=\emptyset\}$.

The fail-stop assumption, that any communication actually taking place with channels in *chan* (including $chan - NF$) is correct, is naturally reflected by FS1 and FS2. To understand the *sub–rub* pair, suppose P and Q of Figure 1 refuse S and R respectively at *chan*. Since no input should be refused at the non-faulty channels NF,

$R\cap\alpha NF$ must be empty—i.e. $R\in rub(t)$—and $S\in sub(t)$ is the weakest condition to avoid deadlock at *chan* (again *sub* is chosen maximal).

In Section 11 we will introduce extraction patterns aimed at modelling N-modular redundancy with majority voting.

3.2 Deterministic Input-Output Base Processes

In the CSP approach, a process is regarded as deterministic if its behaviour is totally determined by the environment (more commonly, such a process would be said to be free from "internal non-determinism"). As explained in [Hoa85], a deterministic process K may be characterized simply by its alphabet and traces.

Definition 3.3 Let C and D be two disjoint finite sets of channels, and T be a non-empty prefix-closed set of traces over C and D. Define K to be a process satisfying $\iota K{=}C$, $\omega K{=}D$, $\delta K{=}\emptyset$ and:

$$\phi K = \{(t,R)\in T\times 2^{\alpha K} \ \mid \ \forall a\in R : t\cdot\langle a\rangle\notin T\}.$$

Such a K will be called a *deterministic input-output* process implementing T and denoted by $\det(T,C,D)$.[7] $\square$

A simple and useful sufficient condition for a process to be equal to a given deterministic process is formulated next.

Proposition 3.4 Let K be a deterministic process and K' be a process such that $\iota K'{=}\iota K$, $\omega K'{=}\omega K$ and the following hold:

(a) $\tau K'\subseteq\tau K$.

(b) $(t,\{a\})\in\phi K'$ implies $t\cdot\langle a\rangle\notin\tau K$.

Then $K'{=}K$.

Proof. First observe that $\delta K'{=}\emptyset{=}\delta K$. Indeed, if $t\in\delta K'$ then, by D1 and D2 in the definition of process, $(t,\{a\})\in\phi K'$ and $t\cdot\langle a\rangle\in\tau K'$, for every $a\in\alpha K'$. This, however, contradicts (a) and (b).

We next observe that $\tau K\subseteq\tau K'$. Indeed, if this does not hold, by contradiction assume $t\cdot\langle a\rangle\in\tau K-\tau K'$. Then, $t\notin\tau K'$ or, by F4, $(t,\{a\})\in\phi K'$. But if $(t,\{a\})\in\phi K'$ then, by (b), $t\cdot\langle a\rangle\notin\tau K$, contradicting the latter assumption. Thus $t\in\tau K-\tau K'$, and the same argument can be applied a finite number of times until $t{=}\langle\rangle$: the contradiction sought.

Thus, by (a), $\tau K'{=}\tau K$. As a result, by (b) and the definition of det(.) and F4, we obtain $\phi K'{=}\phi K$. $\square$

[7] It is straightforward to show that K is indeed a process. Note that $\tau K{=}T$.

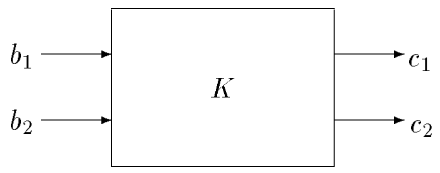

Figure 2: Deterministic input/output process K.

The first class of base processes we consider is that of functional deterministic input-output processes. For a process like K in Figure 2 to belong to this class, at any time the two message sequences output at c_1 and c_2 should either take values completely determined by the two message sequences input at b_1 and b_2, or (arbitrarily) lag behind these values. It is therefore possible to define a function

$$f_i : (\mu b_1)^* \times (\mu b_2)^* \rightarrow (\mu c_i)^* \qquad \text{(for } i{=}1,2\text{)}$$

such that, if t is a trace of K, then $t{\downarrow}c_i {\leq} f_i(t{\downarrow}b_1, t{\downarrow}b_2)$. Moreover, K should not refuse to extend a trace with either any input action or an output action allowed by the current input. It is not difficult to see that the latter conditions amount to f_1, f_2 being monotonic and K deterministic.

These notions are the motivation for the following definition (where the tuple notation of Section 2.1 is employed).

Definition 3.5 Let $\vec{b}$ and $\vec{c}$ be non-empty tuples of distinct channels, $f{\in}\mathcal{F}(\vec{b}, \vec{c})$ be a monotonic function and:

$$T = \{t{\in}(\alpha\vec{b} {\cup} \alpha\vec{c})^* \mid \forall u{\leq}t : \; u{\downarrow}\vec{c} \leq f(u{\downarrow}\vec{b})\}.$$

Then $\det(T, \vec{b}, \vec{c})$ is said to be a *deterministic input-output* (DIO) process implementing f and denoted by $\mathrm{DIO}(f, \vec{b}, \vec{c})$. □

In particular, for a function $f : \mu b^* \rightarrow \mu c^*$, $\mathrm{DIO}(f, b, c)$ is a pipe [Hoa85]. If $\mu b{=}\mu c$ and $f : \mu b^* \rightarrow \mu c^*$ is the identity function, then $\mathrm{DIO}(f, b, c)$ is easily recognized to be an unbounded deterministic buffer with input channel b and output channel c. In the sequel, it will be denoted by *Buff*(b, c).

A DIO process K is *buffered*, in the sense that it is the same as the process obtained by connecting the output of a buffer to an input channel of K, or an output channel of K to the input of a buffer. A proof of this fact is postponed to Section 7.1. Here, it can be exploited to observe that, in addition to being functionally characterized, DIO processes communicate asynchronously, like those introduced in [KM77].

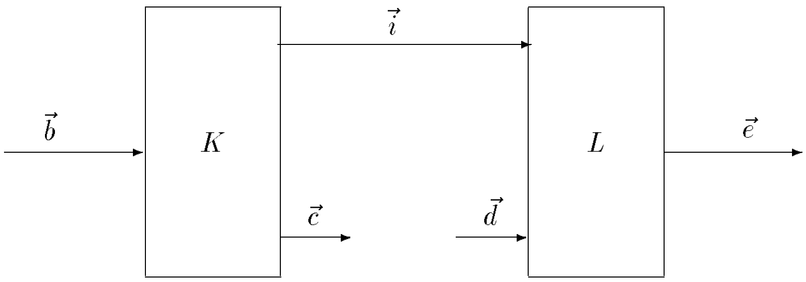

Figure 3: DIO processes K and L.

Function and process composition

The *composition* $f{\cdot}g$ of monotonic functions f and g is defined as follows.
Let $f{\in}\mathcal{F}(\vec{b}, \vec{i}{\cdot}\vec{c})$ and $g{\in}\mathcal{F}(\vec{i}{\cdot}\vec{d}, \vec{e})$, where all channels in the tuples are distinct. Then the function $f{\cdot}g{\in}\mathcal{F}(\vec{b}{\cdot}\vec{d}, \vec{c}{\cdot}\vec{e})$ is defined, for all $z{\in}dom_{\vec{b}\cdot\vec{d}}$ by:[8]

$$f{\cdot}g(z) \;=\; (f(z{\Downarrow}\vec{b}){\Downarrow}\vec{c}) \cdot g(\,(f(z{\Downarrow}\vec{b}){\Downarrow}\vec{i}){\cdot}(z{\Downarrow}\vec{d}))$$

Let $K{=}\mathrm{DIO}(f, \vec{b}, \vec{i}{\cdot}\vec{c})$ and $L{=}\mathrm{DIO}(g, \vec{i}{\cdot}\vec{d}, \vec{e})$ be as in Figure 3. In Section 7.1 we will show that the process $K\#L$ obtained by connecting K and L and hiding internal communication turns out to be $\mathrm{DIO}(f{\cdot}g, \vec{b}{\cdot}\vec{d}, \vec{c}{\cdot}\vec{e})$.

4 Class A Implementation of DIO Processes

Suppose that a system P is intended as an implementation of the base process $K{=}\mathrm{DIO}(f, \vec{b}, \vec{c})$ (see Figure 4). The first requirement this poses on P is that it should be possible to relate its communication to that of K, through an extraction pattern for each channel of K. Moreover, if input to P is well-formed according to the extraction patterns, P has to satisfy the following properties, formalized as AI1-4 below. Firstly, output of P should be well-formed too (AI1). Secondly, if P feeds another implementation process Q, they should not block while output from P is incomplete in the sense that extraction maps it onto incomplete output of K (the output of $K{=}\mathrm{DIO}(f, \vec{b}, \vec{c})$ is complete whenever it equates the f-image of the input). Bounding the refusals S of P and R of Q by AI3 and AI2 respectively prevents blocking (i.e. $\alpha P{\cap}\alpha Q{\not\subseteq}S{\cup}R$) by virtue of EP4. Finally, when connected to Q, P should not introduce divergence; this is ensured by forbidding infinite uninterrupted output from P (AI4).

[8]It is immediate to prove that $f{\cdot}g$ is indeed a monotonic function.

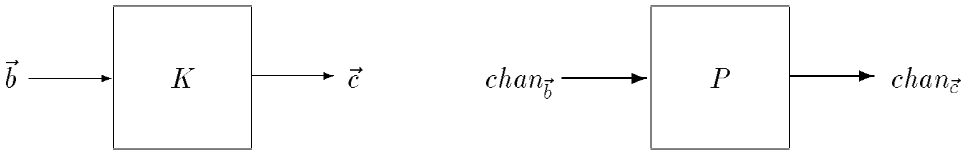

Figure 4: DIO processes K and its implementation P.

Definition 4.1 Let $K=\mathrm{DIO}(f,\vec{b},\vec{c})$ be a deterministic input–output process. Suppose that for every channel x in $\vec{b}$ and $\vec{c}$, ep_x is an extraction pattern over x such that $chan_x \cap chan_{x'} = \emptyset$ for $x \neq x'$.
We say that P is a *class A implementation* of f (or K), and write $P \in \mathrm{AI}(f, ep_{\vec{b}}, ep_{\vec{c}})$ if $\iota P = chan_{\vec{b}}$, $\omega P = chan_{\vec{c}}$, and for every trace $t \in \tau P \cap wf_{\vec{b}}$ the following hold:

AI1 $t \in wf_{\vec{c}}$.

AI2 If $(t,R) \in \phi P$ then $R \in rub_{\vec{b}}(t)$.

AI3 If $(t,R) \in \phi P$ and $extr_{\vec{c}}(t) \neq f(extr_{\vec{b}}(t))$ then $R \in sub_{\vec{c}}(t)$.

AI4 There are only finitely many traces $w \in (\alpha\omega P)^*$ such that $t \cdot w \in \tau P$.

We will also write $P \in \mathrm{AI}i \ldots j(f, ep_{\vec{b}}, ep_{\vec{c}})$ to mean that P satisfies some chosen conditions $\mathrm{AI}i, \ldots, \mathrm{AI}j$ (f is omitted if AI3 is not involved). $\square$

Note that different extraction patterns (hence fault assumptions) are allowed on different channels of P, which is necessary to model heterogeneous networks of implementation processes.

It may not be obvious, at least not until Section 11, how to translate conditions AI1-4 into the properties of actual replicated implementations. This stems from the fact that the notion of implementation is intended to be an abstraction of actual replication techniques.

From AI4 and D1 in the definition of process, it immediately follows that, under the assumptions of Definition 4.1:

AI5 $\delta P \cap wf_{\vec{b}} = \emptyset$.

It is worth observing that a DIO process is its own class A implementation through canonical extraction patterns.

Proposition 4.2 $\mathrm{DIO}(f,\vec{b},\vec{c}) \in \mathrm{AI}(f, can_{\vec{b}}, can_{\vec{c}})$.

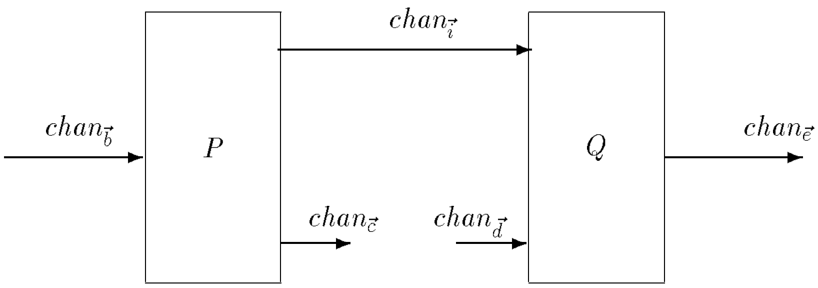

Figure 5: Class-A implementation is preserved through composition.

Proof. Follows immediately from the definition of det(.) and AI1-AI4. □

In order to be motivated operationally, class A implementation should now be shown to enjoy the two properties mentioned in the introduction: distributivity and accessibility.

5 Distributivity

In this section we will show that, under some simple conditions, class A implementability is preserved through process composition. Each of the conditions AI1-AI4 will be dealt with separately to provide more insight into the relationship between the various properties that class A implementation is supposed to capture. Given two base processes $K{=}\mathrm{DIO}(f, \vec{b}, \vec{i}{\cdot}\vec{c})$ and $L{=}\mathrm{DIO}(g, \vec{i}{\cdot}\vec{d}, \vec{e})$ as in Figure 3, and two class A implementations, P of f and Q of g, distributivity requires that $P\#Q$, the interconnection of P and Q (see Figure 5), should be a class A implementation of the composition $f{\cdot}g$ of functions f and g.

Assumption 5.1 For the rest of this section we make the following assumptions:

1. $\vec{b}, \vec{c}, \vec{i}, \vec{d}, \vec{e}$ are disjoint tuples of channels. For each channel x, ep_x is an extraction pattern over x such that $chan_x \cap chan_{x'} = \emptyset$ for $x \neq x'$.
2. $f \in \mathcal{F}(\vec{b}, \vec{i}{\cdot}\vec{c})$ and $g \in \mathcal{F}(\vec{i}{\cdot}\vec{d}, \vec{e})$ are monotonic functions.
3. $P \in \mathrm{AI14}(ep_{\vec{b}}, ep_{\vec{i}\cdot\vec{c}})$ and $Q \in \mathrm{AI14}(ep_{\vec{i}\cdot\vec{d}}, ep_{\vec{e}})$. All the lemmata and propositions below are all formulated and proved under this assumption.

Note that since $\vec{b}, \vec{c}, \vec{i}, \vec{d}, \vec{e}$ are tuples of distinct channels, the channels shared by the base processes K and L of Figure 3 cannot form a cycle. Moreover, domains and ranges of f and g imply that input and output channels of P and Q should be as in Figure 5. In particular, $\chi P \cap \chi Q = \omega P \cap \iota Q = chan_{\vec{i}}$. □

Before showing the desired properties of the composition of P and Q, we prove two auxiliary lemmata. The first one shows that $P\|Q$ satisfies AI5 and, provided $P\|Q$ is thought of as having the same input channels as $P\#Q$, AI1.

Lemma 5.2

(a) $\delta(P\|Q)\cap wf_{\vec{b}.\vec{d}} = \emptyset$.

(b) $\tau(P\|Q)\cap wf_{\vec{b}.\vec{d}} \subseteq wf_{\vec{c}.\vec{i}.\vec{e}}$.

Proof. (a) Suppose by contradiction $t\in\delta(P\|Q)\cap wf_{\vec{b}.\vec{d}}$. From the definition of $\|$ it follows that there is u such that $u\leq t$ and one of the following holds:

(1) $u\lceil\chi P\in\delta P$ and $u\lceil\chi Q\in\tau Q$, or

(2) $u\lceil\chi P\in\tau P$ and $u\lceil\chi Q\in\delta Q$.

Clearly, $u\in wf_{\vec{b}.\vec{d}}$ because $wf_{\vec{b}.\vec{d}}$ is prefix-closed. Since $u\lceil\chi P\in wf_{\vec{b}}$, case (1) contradicts AI5 for P. Case (2), by AI1 for P, implies $u\lceil\chi P\in wf_{\vec{i}.\vec{c}}$, whence $u\lceil\chi Q\in wf_{\vec{i}.\vec{d}}$. This, together with the second conjunct of (2), contradicts AI5 for Q. Hence (a) holds.

(b) Let $t\in\tau(P\|Q)\cap wf_{\vec{b}.\vec{d}}$. Part (a) and the definition of $\|$ imply $t\lceil\chi P\in\tau P$ and $t\lceil\chi Q\in\tau Q$. Hence, using $t\in wf_{\vec{b}.\vec{d}}$ and AI1 for P and Q, $t\in wf_{\vec{c}.\vec{i}.\vec{e}}$ follows. □

The next lemma proves AI5 for $P\#Q$ and shows that the well formed traces of $P\#Q$ (i.e. $(P\|Q)\backslash chan_{\vec{i}}$) are obtained by hiding communication over $chan_{\vec{i}}$ in the well formed traces of $P\|Q$.

Lemma 5.3

(a) $\delta(P\#Q)\cap wf_{\vec{b}.\vec{d}} = \emptyset$.

(b) $\tau(P\#Q)\cap wf_{\vec{b}.\vec{d}} = (\tau(P\|Q)\cap wf_{\vec{b}.\vec{d}})\backslash chan_{\vec{i}}$.

Proof. (a) Suppose $t\in\delta(P\#Q)\cap wf_{\vec{b}.\vec{d}}$. Then, by definition of hiding, there exists u such that $u\backslash chan_{\vec{i}}\leq t$ (whence $u\in wf_{\vec{b}.\vec{d}}$) and either

(1) $u\in\delta(P\|Q)$ or

(2) $u\in\tau(P\|Q)$ and there are $a_1, a_2, \ldots\in\alpha chan_{\vec{i}}$ such that for all $k\geq 1$,

$$u_k = u\cdot\langle a_1, \ldots, a_k\rangle\in\tau(P\|Q).$$

Case (1), together with $u\in wf_{\vec{b}.\vec{d}}$, contradicts Lemma 5.2(a). For case (2), note that $u_k\in wf_{\vec{b}.\vec{d}}$, so by Lemma 5.2(a), $u_k\notin\delta(P\|Q)$. Hence, by definition of $\|$, we obtain $u_k\lceil\chi P=(u\lceil\chi P)\cdot\langle a_1, \ldots, a_k\rangle\in\tau P$ for all $k\geq 1$. This and $u\lceil\chi P\in wf_{\vec{b}}$ contradict AI4 for P, which proves part (a) of the thesis.

Part (b) follows from (a) and Proposition 2.3. □

We may now prove that the AI1 and AI4 properties distribute.

Proposition 5.4 $P\#Q{\in}\mathrm{AI1}(ep_{\vec{b}\cdot\vec{d}}, ep_{\vec{c}\cdot\vec{e}})$.

Proof. Suppose $t{\in}\tau(P\#Q){\cap}wf_{\vec{b}\cdot\vec{d}}$. From Lemma 5.3(b) it follows that there is $s{\in}\tau(P\|Q){\cap}wf_{\vec{b}\cdot\vec{d}}$ such that $s\backslash chan_{\vec{i}}{=}t$. Then, by Lemma 5.2(b), $s{\in}wf_{\vec{c}\cdot\vec{i}\cdot\vec{e}}$, whence $t{=}s\backslash chan_{\vec{i}}{\in}wf_{\vec{c}\cdot\vec{e}}$. □

Proposition 5.5 $P\#Q{\in}\mathrm{AI4}(ep_{\vec{b}\cdot\vec{d}}, ep_{\vec{c}\cdot\vec{e}})$.

Proof. Suppose $t{\in}\tau(P\#Q){\cap}wf_{\vec{b}\cdot\vec{d}}$. Define the following two sets of traces:

$$\begin{aligned} T &= \{t{\cdot}w{\in}\tau(P\#Q) \mid w{\in}\alpha\omega(P\#Q)^*\}{\cup}\{u \mid u{\leq}t\} \\ S &= \{s{\in}\tau(P\|Q) \mid s\backslash chan_{\vec{i}}{\in}T\}. \end{aligned}$$

Clearly, $T{\subseteq}wf_{\vec{b}\cdot\vec{d}}$ and $S{\subseteq}wf_{\vec{b}\cdot\vec{c}\cdot\vec{i}\cdot\vec{d}\cdot\vec{e}}$ (by Lemma 5.2(b)) and S is prefix-closed. Moreover, $S\backslash chan_{\vec{i}}{=}T$ (by Lemma 5.3(b)).
Contrary to AI4, suppose T is infinite. Then S is also infinite (by $S\backslash chan_{\vec{i}}{=}T$). Since $S\backslash(\omega P{\cup}\omega Q){=}\{u \mid u{\leq}t\}\backslash\omega(P\#Q)$ is finite, Proposition 2.1 implies that there are $s{\in}S$ and $a_1, a_2, \ldots{\in}\alpha(\omega P{\cup}\omega Q)$ such that $s_k{=}s{\cdot}\langle a_1, \ldots, a_k\rangle{\in}S$ for all $k{\geq}1$. By Lemma 5.2(a), $s_k{\notin}\delta(P\|Q)$, so (by definition of $\|$) $s_k{\lceil}\chi P{\in}\tau P$ and $s_k{\lceil}\chi Q{\in}\tau Q$ for all $k{\geq}1$. Now, if $a_k{\in}\alpha\omega P$ for infinitely many k, $\{s_k{\lceil}\chi P \mid k{\geq}1\}$ would be infinite, contradicting AI4 for P. Thus, let n be the maximum index for which $a_n{\in}\alpha\omega P$; then:

$$\{s_k{\lceil}\chi Q \mid k{>}n\} = \{(s{\cdot}\langle a_1, \ldots, a_n\rangle{\lceil}\chi Q){\cdot}\langle a_{n+1}, \ldots, a_k\rangle \mid k{>}n\}$$

is infinite, contradicting AI4 for Q. □

The next two propositions provide a means of proving AI2 and AI3 in constructing a class A implementation. In fact, for the purposes of this section, they could have just stated, respectively, that if P and Q are AI2 so is $P\#Q$, and that if P is AI3 and Q is AI23 then $P\#Q$ is AI3. However, the hypotheses about P and Q have been intentionally made slightly weaker.

Proposition 5.6 Let $\vec{j}$ be a channel tuple, $\vec{j}{\subseteq}\vec{i}$, such that:

(a) If $(t, R){\in}\phi P$ and $t{\in}wf_{\vec{b}}$ and $R{\notin}sub_{\vec{j}}(t)$, then $R{\in}rub_{\vec{b}}(t)$.

(b) If $(t, R){\in}\phi Q$ and $t{\in}wf_{\vec{i}\cdot\vec{d}}$ then $R{\in}rub_{\vec{d}\cdot\vec{j}}(t)$.

Then $P\#Q{\in}\mathrm{AI2}(ep_{\vec{b}\cdot\vec{d}}, ep_{\vec{c}\cdot\vec{e}})$. □

Proposition 5.7 Let Ψ be a set of traces, $\Psi{\subseteq}(\alpha chan_{\vec{i}})^*$, such that:

(a) If $(t,R)\in\phi P$ and $t\in wf_{\vec{b}}$ and $R\notin sub_{\vec{i}\cdot\vec{c}}(t)$, then:

(i) $extr_{\vec{i}\cdot\vec{c}}(t)=f(extr_{\vec{b}}(t))$.

(ii) $t\lceil chan_{\vec{i}}\in\Psi$.

(b) If $(t,R)\in\phi Q$ and $t\in wf_{\vec{i}\cdot\vec{d}}$ and $R\notin sub_{\vec{e}}(t)$, then:

(i) $R\in rub_{\vec{i}}(t)$.

(ii) $t\lceil chan_{\vec{i}}\in\Psi$ implies $extr_{\vec{e}}(t)=g(extr_{\vec{i}\cdot\vec{d}}(t))$.

Then $P\#Q\in\mathrm{AI3}(f\cdot g, ep_{\vec{b}\cdot\vec{d}}, ep_{\vec{c}\cdot\vec{e}})$. □

The two propositions are proved together.

Proof. Let $(t,R)\in\phi(P\#Q)$ be such that $t\in wf_{\vec{b}\cdot\vec{d}}$. Then $t\notin\delta(P\#Q)$ by Lemma 5.3(a).

¿From Proposition 2.3, it follows that there is $(w, R\cup\alpha chan_{\vec{i}})\in\phi(P\|Q)$ such that $w\notin\delta(P\|Q)$ and $t=w\backslash chan_{\vec{i}}$. Note that $w\in wf_{\vec{b}\cdot\vec{d}}$ and, by Lemma 5.2(b), $w\in wf_{\vec{c}\cdot\vec{i}\cdot\vec{e}}$. Thus, by the definition of $\|$, there are X and Y such that:

(1) $X\cup Y=R\cup\alpha chan_{\vec{i}}$.

(2) $(w\lceil\chi P, X)\in\phi P$ and $w\lceil\chi P\in wf_{\vec{b}}$.

(3) $(w\lceil\chi Q, Y)\in\phi Q$ and $w\lceil\chi Q\in wf_{\vec{i}\cdot\vec{d}}$.

In the following, R is often interchanged with X or Y, in accordance with:

For all $x\in\vec{b}\cup\vec{c}$, $R\cap\alpha chan_x = X\cap\alpha chan_x$ (by (1) and $Y\subseteq\alpha Q$, by (3)).

For all $y\in\vec{d}\cup\vec{e}$, $R\cap\alpha chan_y = Y\cap\alpha chan_y$ (by (1) and $X\subseteq\alpha P$, by (2)).

For $x\in\vec{i}$, (1) implies $(X\cap\alpha chan_x)\cup(Y\cap\alpha chan_x)=\alpha chan_x$. Hence, EP4 (note that $w\lceil chan_x\in wf_x$) implies that, for all $x\in\vec{i}$, $Y\in rub_x(w)$ implies $X\notin sub_x(w)$. Thus:

(4) For all $\vec{i'}\subseteq\vec{i}$, $Y\in rub_{\vec{i'}}(w)$ implies $X\notin sub_{\vec{i'}}(w)$.

To prove Proposition 5.6, we use (3) and hypothesis 5.6(b) for Q to infer:

(5) $R\in rub_{\vec{d}}(w)=rub_{\vec{d}}(t)$.

(6) $Y\in rub_{\vec{j}}(w)$.

From (4) and (6) we infer:

(7) $X\notin sub_{\vec{j}}(w)$.

Finally this, (2) and hypothesis 5.6(a) for P implies:

(8) $R\in rub_{\vec{b}}(w)=rub_{\vec{b}}(t)$.

Thus, by (5) and (8), Proposition 5.6 holds.

We now tackle the proof of Proposition 5.7. To show that AI3 holds, assume:

(9) $R \notin sub_{\vec{c}\cdot\vec{e}}(t) = sub_{\vec{c}\cdot\vec{e}}(w)$.

From (3,9), by hypothesis 5.7(b)(i) it follows:

(10) $Y \in rub_{\vec{i}}(w)$.

From (9) (for subscripts $\vec{c}$ and $\vec{e}$) and (10,4) (for subscript $\vec{i}$), it follows:

(11) $X \notin sub_{\vec{c}\cdot\vec{i}}(w)$.

(12) $Y \notin sub_{\vec{e}}(w)$.

From (2,11), by hypothesis 5.7(a), it follows:

(13) $extr_{\vec{i}\cdot\vec{c}}(w) = f(extr_{\vec{b}}(w))$.

(14) $w \lceil chan_{\vec{i}} \in \Psi$.

From (3,12,14), by hypothesis 5.7(b)(ii), it follows:

(15) $extr_{\vec{e}}(w) = g(extr_{\vec{i}\cdot\vec{d}}(w))$.

Finally, by (13) and (15), recalling Remark 3.2, $t = w \backslash chan_{\vec{i}}$ and the definition of function composition:

$$\begin{aligned} extr_{\vec{c}\cdot\vec{e}}(t) &= (f(extr_{\vec{b}}(t)) \Downarrow \vec{c}) \cdot g(\,(f(extr_{\vec{b}}(t)) \Downarrow \vec{i}) \cdot extr_{\vec{d}}(t)\,) \\ &= f \cdot g(extr_{\vec{b}\cdot\vec{d}}(t)). \end{aligned}$$

Hence AI3 holds for $P \# Q$. $\square$

We may now state the distributivity theorem for class A implementation.

Theorem 5.8 If $P \in \mathrm{AI}(f, ep_{\vec{b}}, ep_{\vec{i}\cdot\vec{c}})$ and $Q \in \mathrm{AI}(g, ep_{\vec{i}\cdot\vec{d}}, ep_{\vec{e}})$ then

$$P \# Q \in \mathrm{AI}(f \cdot g, ep_{\vec{b}\cdot\vec{d}}, ep_{\vec{c}\cdot\vec{e}}).$$

Proof. If P and Q satisfy AI2 and AI3 then they also satisfy the hypotheses of Propositions 5.6 and 5.7, with respectively $\vec{j} = \vec{i}$ and $\Psi = (\alpha chan_{\vec{i}})^*$.[9] Together with Propositions 5.4 and 5.5, this yields the thesis. $\square$

6 Extractors and Disturbers

Extractors and disturbers are processes that may be used to build a base process from its class A implementation, as required by the accessibility property. Intuitively, a disturber transforms communication along its only input channel into output interpretable through an extraction pattern, while an extractor works the other way round.

[9] An application of Proposition 5.6 with $\vec{j} \neq \vec{i}$ will be made in Theorem 10.4, and of Proposition 5.7 with $\Psi \neq (\alpha chan_{\vec{i}})^*$ in Theorem 11.5.

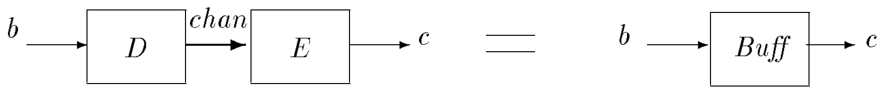

Figure 6: A disturber/extractor series.

Let $ep{=}(chan, msg, wf, extr, sub, rub)$ be an extraction pattern over a channel $c{\notin}chan$ and $id\colon msg^* \rightarrow msg^*$ be the identity function. Both ep and c will be fixed for the rest of this section.

A process D is a *disturber* for ep and c if $D{\in}\mathrm{AI}(id, can_c, ep)$, and a process E is an extractor for ep and c if $E{\in}\mathrm{AI}(id, ep, can_c)$. We denote this respectively by $D{\in}\mathrm{DIST}(ep, c)$ and $E{\in}\mathrm{EXTR}(ep, c)$.

We will show that the composition of a disturber with a suitable extractor always yields a buffer process (see Figure 6 and Proposition 7.2). The following lemma shows that an extractor outputs the $extr$-image of its input and will never refuse to do so.

Lemma 6.1 Let E be an extractor as above. If $(t, R){\in}\phi E$ and $t{\in}wf$ then:

(a) There is $w{\in}\alpha c^*$ such that $t{\cdot}w{\in}\tau E$ and $(t{\cdot}w){\downarrow}c{=}extr(t)$.

(b) $t{\downarrow}c{\leq}extr(t)$.

(c) If $(t{\downarrow}c){\cdot}\langle v\rangle{\leq}extr(t)$ then $c!v{\notin}R$, else $(t, R{\cup}\{c!v\}){\in}\phi E$.

(d) If $t{\downarrow}c{=}extr(t)$ then $(t, R{\cup}\alpha c){\in}\phi E$.

Proof. (a,b) There exists $w{\in}\alpha c^*$ such that $(t{\cdot}w, \alpha c){\in}\phi E$, or AI4 is easily seen to be violated. For such a w, by AI3, $t{\downarrow}c{\leq}(t{\cdot}w){\downarrow}c{=}extr(t{\cdot}w){=}extr(t)$.

(c) (then part) If $(t{\downarrow}c){\cdot}\langle v\rangle{\leq}extr(t)$, (b) implies $t{\cdot}\langle a\rangle{\notin}\tau E$ for all $a{\in}\alpha c{-}\{c!v\}$, whence, by F4 and induction, $(t, R{\cup}(\alpha c - \{c!v\})){\in}\phi E$; thus $c!v{\notin}R$ or, by AI3, $t{\downarrow}c{=}extr(t)$, a contradiction.

(c) (else part) If $(t{\downarrow}c){\cdot}\langle v\rangle{\not\leq}extr(t)$, then $t{\cdot}\langle c!v\rangle{\notin}\tau E$ (or (b) would be contradicted). Thus $(t, R{\cup}\{c!v\}){\in}\phi E$ by F4.

(d) Follows directly from (c). □

A result similar to the (a) and (b) parts of the last lemma can also be shown to hold for disturbers.

Lemma 6.2 Let D be a disturber as above and $t{\in}\tau D$. Then:

(a) There is $w \in (\alpha chan)^*$ such that $t \cdot w \in \tau D$ and $t{\downarrow}c = extr(t \cdot w)$.

(b) If there is $E \in \mathrm{EXTR}(ep, d)$ such that $d \notin chan \cup \{c\}$ and $t{\lceil}chan \in (\tau E){\lceil}chan$, then $extr(t) \leq t{\downarrow}c$.

Proof. (a) There exists $w \in (\alpha chan)^*$ such that $(t \cdot w, \alpha chan) \in \phi E$, or AI4 would not hold. For such a w, by AI3, $t{\downarrow}c = (t \cdot w){\downarrow}c = extr(t \cdot w)$. (Note that we cannot deduce $extr(t) \leq t{\downarrow}c$ since $extr$ need not be monotonic.)
(b) By hypothesis, there exists $u \in \tau E$ such that $t{\lceil}chan = u{\lceil}chan$. By Lemma 6.1(a), we can choose u such that $u{\downarrow}d = extr(u{\lceil}C) = extr(t{\lceil}C)$. It is clearly possible to choose $z \in (\alpha(D \| E))^*$ such that $z{\lceil}\chi D = t$ and $z{\lceil}\chi E = u$, so that, by definition of $\|$, $z \in \tau(D \| E)$. By Proposition 2.3, $s = z{\lceil}\{c, d\} \in \tau(D \# E)$. Moreover, by Proposition 7.2,[10] $D \# E = \mathit{Buff}(c, d)$. Hence $s{\downarrow}d \leq s{\downarrow}c$. Thus we obtain:

$$extr(t{\lceil}C) = u{\downarrow}d = z{\downarrow}d = s{\downarrow}d \leq s{\downarrow}c = z{\downarrow}c = t{\downarrow}c. \qquad \square$$

6.1 Existence of Extractors

To show that for each extraction pattern there is at least one extractor, we supplement the requirements EP0–4 about extraction pattern with a constraint on potential non-monotonicity of $extr$. More specifically, an extractor should be allowed to refuse to extend its input by actions that violate monotonicity of the extraction mapping.

EP5 Let $t \in \mathit{wf} \cap (\alpha chan)^*$ and $R \subseteq \alpha chan$. If $t \cdot \langle a \rangle \in \mathit{wf}$ and $extr(t) \not\leq extr(t \cdot \langle a \rangle)$, for all $a \in R$, then $R \in rub(t)$.

All extraction patterns with $extr$ monotonic, including the canonical and fail-stop ones, clearly satisfy EP5.
EP5 permits the following existence result to be proved.

Proposition 6.3 If ep satisfies EP5 then the process $E = \det(V, \mathit{chan}, \{c\})$ is an extractor in $\mathrm{EXTR}(ep, c)$, where V is defined by:

$$V = \{t \in (\alpha chan \cup \alpha c)^* \mid \forall u \leq t : u \in \mathit{wf} \Rightarrow u{\downarrow}c \leq extr(u)\}.$$

Proof. E is a process since V is non-empty and prefix-closed. Clearly, AI1 holds. To prove AI2-4, assume $(t, R) \in \phi E$ and $t \in \mathit{wf}$, whence

(1) $t{\downarrow}c \leq extr(t)$.

[10] The proof of Proposition 7.2 does not depend on Lemma 6.2.

Let $a \in R \cap \alpha chan$. By the definition of det(.), we have $t \cdot \langle a \rangle \notin V$. Hence $t \cdot \langle a \rangle \in wf$ and $t{\downarrow}c \not\leq extr(t \cdot \langle a \rangle)$, which implies, by (1), $extr(t) \not\leq extr(t \cdot \langle a \rangle)$. Thus, by EP5, $R \in rub(t)$. Hence AI2 holds.

If $\alpha c \subseteq R$, then, by the definitions of det(.) and V, for all $a \in \alpha c$, $(t \cdot \langle a \rangle){\downarrow}c \not\leq extr(t)$. Hence, by (1), $t{\downarrow}c = extr(t)$, as required by AI3. Finally, the set

$$\{t \cdot w \in \tau E \mid w \in (\alpha \omega E)^*\} \;=\; \{t \cdot w \mid w \in (\alpha c)^* \wedge (t{\downarrow}c) \cdot (w{\downarrow}c) \leq extr(t)\}$$

is clearly finite, so AI4 holds. □

Note that the extractor E defined above is a most general one in a twofold sense: for any other extractor E' in EXTR(ep, c), $\tau E' \subseteq \tau E$ holds and, if $\tau E' = \tau E$, E is more deterministic than E'.[11]

EP5 has been shown to be a sufficient condition for the existence of extractors. Moreover, it becomes a necessary one if the extraction pattern does not contain any "useless" traces in the sense of the proviso of the following proposition.

Proposition 6.4 The extraction pattern ep satisfies EP5 provided that, for every $t \in wf \cap (\alpha chan)^*$, there is an extractor $E \in$ EXTR(ep, c) such that $t \in (\tau E) \lceil chan$.

Proof. Let $t \in wf \cap (\alpha chan)^*$. Then there is an extractor $E \in$ EXTR(ep, c) with a trace $z \in \tau E$ such that $z \lceil chan = t$. Moreover, by Lemma 6.1(a), we may assume that:

(1) $z{\downarrow}c = extr(z) = extr(t)$.

Consider any set $R \subseteq \alpha chan$ such that, for all $a \in R$, $t \cdot \langle a \rangle \in wf$ and $extr(t) \not\leq extr(t \cdot \langle a \rangle)$. Then, to prove $R \in rub(t) = rub(z)$, it suffices, by AI2, to show $(z, R) \in \phi E$. The latter fact follows (by F4 and induction) from $z \cdot \langle a \rangle \notin \tau E$ for all $a \in R$. Indeed, suppose the contrary holds, i.e. for some $a \in R$ $z \cdot \langle a \rangle \in \tau E$; then by Lemma 6.1(b), $(z \cdot \langle a \rangle){\downarrow}c \leq extr(z \cdot \langle a \rangle)$, whence $z{\downarrow}c \leq extr(t \cdot \langle a \rangle)$. Thus, by (1), $extr(t) \leq extr(t \cdot \langle a \rangle)$, a contradiction. □

6.2 Existence of Disturbers

Conditions EP0–5 are insufficient to guarantee the existence of disturbers. For example, $wf = \{\langle\rangle\}$, $extr(\langle\rangle) = \langle\rangle$, $rub(\langle\rangle) = sub(\langle\rangle) = \{\emptyset\}$ define an extraction pattern satisfying EP0–5 for which no disturber exists in DIST(ep, c). This is a rather extreme example of an extraction pattern whose $extr$ mapping lacks surjectivity—a property needed to ensure the existence of at least one disturber. For suppose that for some $z \in msg^*$ there is no u such that $extr(u) = z$ ($extr$ is not surjective). Then no D can lie in DIST(ep, c) because by AI2 this D should accept $s \in (\alpha c)^*$ such that $s{\downarrow}c = z$. Yet, Lemma 6.2(a) ensures that, for some $w \in (\alpha chan)^*$, $s{\downarrow}c = z = extr(w)$, which contradicts the assumption about $extr$.

Therefore, we formulate an additional condition for ep:

[11] This follows from the definition of the V set and Lemma 6.1(b).

EP6 There is a mapping $inv : msg^* \rightarrow 2^{wf \cap (\alpha chan)^*}$ such that for all $z, y \in msg^*$ the following are satisfied (below $Inv(z)$ denotes the prefix-closure of $inv(z)$):

- **(a)** $inv(z)$ is a non-empty, finite set such that $extr(inv(z))=\{z\}$.
- **(b)** If $z \leq y$ then $inv(z) \subseteq Inv(y)$.
- **(c)** If $t \in Inv(z) - inv(z)$ and $R \subseteq \alpha chan$ are such that, for all $a \in R$, $t \cdot \langle a \rangle \notin Inv(z)$, then $R \in sub(t)$.

Informally, traces in $inv(z)$ may be seen as a choice of finitely many possible encodings over *chan* of the sequence z over c; they can be decoded back into z through $extr$. Thus, traces in $Inv(z) - inv(z)$ may be extended to an encoding of z. EP6(b) ensures that, if a message sequence z may be extended to a longer one y, then every encoding of z over *chan* may be extended to an encoding of y. Finally, EP6(c) requires that if a sequence t over *chan* can be extended to an encoding of z, then a process should be allowed to refuse to continue t in a way that cannot be further extended to an encoding of z.

We now can prove an existence result for disturbers (note that EP5 is not used in this section).

Proposition 6.5 If ep satisfies EP6 then the process $D = \det(V, \{c\}, chan)$ is a disturber in $\mathrm{DIST}(ep, c)$, where V is defined by:

$$V = \{t \in (\alpha chan \cup \alpha c)^* \mid \forall u \leq t : u \lceil chan \in Inv(u \downarrow c)\}.$$

Proof. We first observe that V is non-empty and prefix-closed; hence D is well defined. Moreover, AI1 and AI4 are easy to prove for D (the latter using EP6(a)). By EP6(b), if $t \in V$ and $a \in \alpha c$, then $t \cdot \langle a \rangle \in V$. This and the definition of det(.) imply that D cannot refuse any $R \subseteq \alpha c$, hence AI2.

To prove AI3, let $(t, R) \in \phi D$ and $extr(t) \neq t \downarrow c$. Then $t \lceil chan \in Inv(t \downarrow c) - inv(t \downarrow c)$. Moreover, for all $a \in R$ (by the definition of det(.)) $t \cdot \langle a \rangle \notin V$, and (by AI2) $a \in \alpha chan$, whence $(t \cdot \langle a \rangle) \lceil chan \notin Inv((t \cdot \langle a \rangle) \downarrow c) = Inv(t \downarrow c)$. This, by EP6(c), implies $R \in sub(t)$. □

We next show that EP6 is also a necessary condition.

Proposition 6.6 The extraction pattern ep satisfies condition EP6 provided that $\mathrm{DIST}(ep, c)$ is non-empty.

Proof. Let D be a disturber in $\mathrm{DIST}(ep, c)$. Below, we assume $z \in msg^*$. We denote, for $z = \langle z_1, \ldots, z_k \rangle$, $c!z = \langle c!z_1, \ldots, c!z_k \rangle$ and define:

$$inv(z) = \{t \lceil chan \mid t \in \tau D \wedge t \downarrow c = extr(t) = z\}.$$

Note that $t \in \tau D$ implies $t \in wf$ by AI1, which makes $extr(t)$ well-defined. This set is non-empty because $c!z \in \tau D$ (by AI2, D accepts any input) and therefore, by Lemma 6.2(a), for some $w \in (\alpha chan)^*$, $c!z \cdot w \in \tau D$ and

$$extr(c!z \cdot w) = extr(w) = (c!z \cdot w){\downarrow}c = z.$$

$inv(z)$ is also finite or, by Proposition 2.1, AI4 would be violated. Clearly, $inv(z) \subseteq wf$ by AI1, and $extr(inv(z)) = \{z\}$. This proves EP6(a).

A useful property is:

(1) If $s \in \tau D$, then $s \lceil chan \in Inv(s{\downarrow}c)$.

It follows from Lemma 6.2(a), by choosing $w \in (\alpha chan)^*$ such that $s \cdot w \in \tau D$ and $(s \cdot w){\downarrow}c = extr(s \cdot w)$, whereby $(s \cdot w)\lceil chan \in inv((s \cdot w){\downarrow}c) = inv(s{\downarrow}c)$.

To prove EP6(b), let $y = z \cdot z' \in msg^*$ and $w \in inv(z)$. By definition of $inv(z)$, choose $t \in \tau D$ such that $w = t \lceil chan$ and $t{\downarrow}c = z = extr(t)$. By AI2 ($D$ accepts any input), $t \cdot c!z' \in \tau D$, whence, by (1), as desired:

$$(t \cdot c!z')\lceil chan = w \in Inv(z \cdot z') = Inv(y).$$

To prove EP6(c), we assume $t \in Inv(z) - inv(z)$.

We show first that there is $s \in \tau D$ such that:

(2) $s \lceil chan = t$ and $s{\downarrow}c \leq z$ and $extr(s) \neq s{\downarrow}c$.

Indeed, the assumption $t \in Inv(z)$ ensures that, for some $u \in \tau D$, $u \lceil chan = t$ and $u{\downarrow}c \leq z$. If, for this u, $extr(u) \neq u{\downarrow}c$, we are done. Otherwise, $extr(u) = u{\downarrow}c$ and we proceed thus. Since $t \notin inv(z)$, $u{\downarrow}c \neq z$. Hence, by $u{\downarrow}c \leq z$, there is $v \in msg$ such that $(u{\downarrow}c) \cdot \langle v \rangle \leq z$. Define $s = u \cdot \langle c!v \rangle$, so that $s{\downarrow}c \leq z$; moreover, $s \in \tau D$ (by AI2), $s \lceil chan = u \lceil chan = t$ and $extr(s) = extr(u) = u{\downarrow}c \neq s{\downarrow}c$; this completes the proof of (2).

Assume now that, for all $a \in R \subseteq \alpha chan$, $t \cdot \langle a \rangle \notin Inv(z)$. Let s be as in (2). Suppose $a \in R$ and, by contradiction, $s \cdot \langle a \rangle \in \tau D$. Then, by (1),

$$t \cdot \langle a \rangle = (s \cdot \langle a \rangle)\lceil chan \in Inv((s \cdot \langle a \rangle){\downarrow}c) = Inv(s{\downarrow}c).$$

Hence, by EP6(b) and $s{\downarrow}c \leq z$, $t \cdot \langle a \rangle \in Inv(z)$, a contradiction. Thus $s \cdot \langle a \rangle \notin \tau D$, for all $a \in R$. Hence $(s, R) \in \phi D$ (by F4 and induction). Thus from (2) by AI3 we obtain $R \in sub(s) = sub(t)$. □

Both types of extraction patterns defined so far satisfy EP5 and EP6. EP5 is satisfied since the *extr* mappings are monotonic. EP6 can be shown to hold by taking $inv(\langle v_1, \ldots, v_k \rangle) = \{\langle c!v_1, \ldots, c!v_k \rangle\}$ for the canonical extraction pattern, and $inv(\langle v_1, \ldots, v_k \rangle) = \{\langle c_1!v_1, \ldots, c_1!v_k, \ldots, c_m!v_1, \ldots, c_m!v_k \rangle\}$, where $c_1, \ldots, c_m$ are the channels in *NF*, for the fail-stop extraction pattern (the same *inv* works also for the majority voting extraction pattern introduced later in this paper).

The results obtained in this section suggest, especially in view of the role of disturbers and extractors in the accessibility results formulated later on, that extraction patterns should satisfy conditions EP5 and EP6, in addition to EP0–4. Incidentally, neither EP5 nor EP6 is needed in order to establish the distributivity and accessibility of class A implementation.

7 Accessibility

Together with distributivity, accessibility, whereby a base system can be constructed from its implementation, motivates operationally our notion of implementation. For class A implementation, the accessibility property is based on the following result.

Lemma 7.1 Let $P{\in}\mathrm{AI}(f, ep_{\vec{b}}, ep_{\vec{c}})$ be as in Definition 4.1. Suppose that $ep_{\vec{b}}{=}can_{\vec{b}}$ and, for every $x{\in}\vec{c}$, $x{\notin}\chi P$ and $E_x{\in}\mathrm{EXTR}(ep_x, x)$. Then:[12]

$$\#(P, E_{\vec{c}}) = \mathrm{DIO}(f, \vec{b}, \vec{c}).$$

Proof. Let $Q{=}\#(P, E_{\vec{c}}){=}\|(P, E_{\vec{c}})\backslash chan_{\vec{c}}$.
Theorems 2.4, 5.8 (AI distributivity) and $E_x{\in}\mathrm{AI}(id, ep_x, can_x)$ for all $x{\in}\vec{c}$ imply $Q{\in}\mathrm{AI}(f, can_{\vec{b}}, can_{\vec{c}})$. Hence, by AI5, $\delta Q{=}\emptyset$.
Suppose $(t, R){\in}\phi Q$.
From $\delta Q{=}\emptyset$ and Proposition 2.3 it follows that there is $(w, R{\cup}\alpha chan_{\vec{c}}){\in}\phi(\|(P, E_{\vec{c}}))$ such that $w{\notin}\delta(\|(P, E_{\vec{c}}))$ and $t{=}w\backslash chan_{\vec{c}}$. Hence, by Proposition 2.2, there are R_0 and R_x for all $x{\in}\vec{c}$ such that:

(1) $R_0{\cup}\bigcup_{x\in\vec{c}} R_x{=}R{\cup}\alpha chan_{\vec{c}}$.

(2) $(w{\lceil}\chi P, R_0){\in}\phi P$ (whence $R_0{\subseteq}\alpha P$).

(3) $(w{\lceil}\chi E_x, R_x){\in}\phi E_x$ (whence $R_x{\subseteq}\alpha E_x$), for all $x{\in}\vec{c}$.

From (2) and AI1 for P it follows that:

(4) $w{\in}wf_{\vec{c}}$.

Let $x{\in}\vec{c}$. (1,2,3) imply $(R_x{\cap}\alpha chan_x){\cup}(R_0{\cap}\alpha chan_x){=}\alpha chan_x$, while (3,4) and AI2 for E_x imply $R_x{\in}rub_x(w)$. Hence by EP4 $R_0{\notin}sub_x(w)$ for all $x{\in}\vec{c}$, i.e. $R_0{\notin}sub_{\vec{c}}(w)$. Thus, by AI3 for P:

(5) $extr_{\vec{c}}(w) = f(w{\downarrow}\vec{b}){=}f(t{\downarrow}\vec{b})$.

From (3), by Lemma 6.1(b), it follows that $w{\downarrow}\vec{c}{\leq}extr_{\vec{c}}(w)$, whence

(6) $t{\downarrow}\vec{c}{\leq}extr_{\vec{c}}(w)$.

Assume $a{\in}R$. We first observe that, by AI2 for Q, $a{\notin}\alpha\vec{b}$. Thus:

[12] If Q_x is a process for every $x{\in}\vec{c}{=}(c_1, \ldots, c_n)$, $Q_{\vec{c}}$ may abbreviate $Q_{c_1}, \ldots, Q_{c_n}$. E.g., $\#(P, E_{\vec{c}})$ stands for $\#(P, E_{c_1}, \ldots, E_{c_n})$.

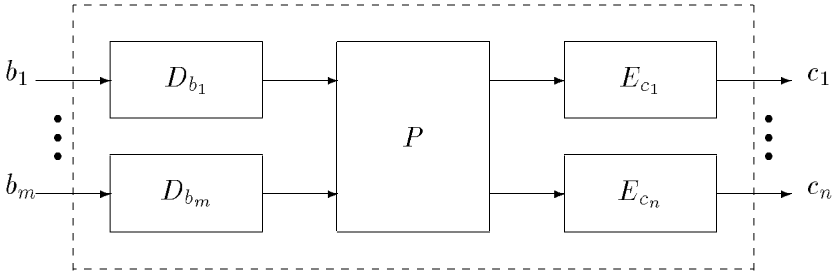

Figure 7: A base DIO process constructed from its implementation P.

(7) $a{\in}\alpha\omega Q$.

Suppose $a{=}x!v$ for some $x{\in}\vec{c}$. This and (1,2,3) implies $a{\in}R_x$. Hence, by (3,4) and Lemma 6.1(c), we obtain $(w{\downarrow}x){\cdot}\langle v\rangle{\not\leq}extr_x(w)$ which implies:

(8) $(t{\cdot}\langle a\rangle){\downarrow}\vec{c}{\not\leq}extr_{\vec{c}}(w)$.

Consequently, by (5), we have $(t{\cdot}\langle a\rangle){\downarrow}\vec{c}{\not\leq}f(t{\downarrow}\vec{b}\,)$, i.e.:

(9) $(t{\cdot}\langle a\rangle){\downarrow}\vec{c}{\not\leq}f((t{\cdot}\langle a\rangle){\downarrow}\vec{b}\,)$.

We are now in a position to prove the thesis. Let $K{=}\mathrm{DIO}(f,\vec{b}\,,\vec{c}\,)$. From (5) and (6) it follows that $\tau Q{\subseteq}\tau K$. Moreover, from (9) it follows that $(t,R){\in}\phi Q$ implies $t{\cdot}\langle a\rangle{\notin}\tau K$, for all $a{\in}R$. Thus, by Proposition 3.4, $Q{=}K$. □

A corollary of the above lemma is that a series of a disturber with an extractor (with the same extraction pattern) is a buffer.

Proposition 7.2 Let b and c be two distinct channels. If $D{\in}\mathrm{DIST}(ep,b)$ and $E{\in}\mathrm{EXTR}(ep,c)$ then $D\#E{=}\mathit{Buff}(b,c)$.

Proof. By definition of disturbers and extractors and Lemma 7.1. □

To show the accessibility result for class A implementation, we surround a class A implementation by suitable disturbers (supplying inputs) and extractors (accepting outputs), and then hide internal communication. The resulting system is the base DIO process that the class A process is supposed to implement (see Figure 7, where $\vec{b}{=}(b_1,\ldots,b_m)$ and $\vec{c}{=}(c_1,\ldots,c_n)$).

Theorem 7.3 Let $P{\in}\mathrm{AI}(f,ep_{\vec{b}},ep_{\vec{c}})$, where $\vec{b}$ and $\vec{c}$ are tuples of distinct channels not in χP. Let $D_x{\in}\mathrm{DIST}(ep_x,x)$ for all $x{\in}\vec{b}$ and $E_x{\in}\mathrm{EXTR}(ep_x,x)$ for all $x{\in}\vec{c}$. Then

$$\#(D_{\vec{b}},P,E_{\vec{c}}) = \mathrm{DIO}(f,\vec{b}\,,\vec{c}\,).$$

Proof. By repeatedly applying Theorems 2.4 and 5.8 (AI distributivity), we obtain that the following holds: $\#(D_{\vec{b}}, P) \in \mathrm{AI}(f, can_{\vec{b}}, ep_{\vec{c}})$. Hence, by Lemma 7.1 and Theorem 2.4, the thesis follows. $\square$

7.1 Composition of DIO Processes

In Section 3.2, the result proved below as Theorem 7.5 was stated without proof because it can be derived from the results of Sections 4 and 7. The theorem is better introduced using a notation to describe the connection of buffers to process channels.

Definition 7.4 Let P be a process. Assume an injective function over channels mapping every $x \in \chi P$ into $\widehat{x} \notin \chi P$, and let $B_x = \mathit{Buff}(\widehat{x}, x)$. Let $\vec{c}$ be a tuple of channels from ωP and $\widehat{P}$ be P with every channel $x \in \vec{c}$ renamed to $\widehat{x}$. Then define:[13]

$$P[\vec{c}\rangle = \#(\widehat{P}, B_{\vec{c}}).$$

Likewise, we define buffering on input channels $\vec{b}$ as $\langle\vec{b}]P$. $\square$

Theorem 7.5 Let $K = \mathrm{DIO}(f, \vec{b}, \vec{i} \cdot \vec{c})$ and $L = \mathrm{DIO}(g, \vec{i} \cdot \vec{d}, \vec{e})$ with all the channels in the tuples being distinct. Then $K \# L = \mathrm{DIO}(f \cdot g, \vec{b} \cdot \vec{d}, \vec{c} \cdot \vec{e})$ (see Figure 3).

Proof. The proof is based on the fact that a DIO process implementing a function is also a class A implementation of that function for the canonical extraction patterns (Proposition 4.2), and that a buffer is an extractor based on a canonical extraction pattern. Lemma 7.1 may then be employed to derive $K = K[\vec{i} \cdot \vec{c}\rangle$ and $L = L[\vec{e}\rangle$, whence:

(1) $K \# L = K[\vec{i} \cdot \vec{c}\rangle \# L[\vec{e}\rangle$.

Theorem 2.4 is employed throughout the proof to decompose process networks as necessary. In particular, from (1) we get:

(2) $K \# L = (K[\vec{i}\rangle \# L)[\vec{c} \cdot \vec{e}\rangle$.

Now infer, by repeated application of Proposition 4.2 and Theorem 5.8 (AI distributivity), $K[\vec{i}\rangle \in \mathrm{AI}(f, can_{\vec{b}}, can_{\vec{i} \cdot \vec{c}})$, and, by one more application, $K[\vec{i}\rangle \# L \in \mathrm{AI}(f \cdot g, can_{\vec{b} \cdot \vec{d}}, can_{\vec{c} \cdot \vec{e}})$. Hence, by Lemma 7.1: $(K[\vec{i}\rangle \# L)[\vec{c} \cdot \vec{e}\rangle = \mathrm{DIO}(f \cdot g, \vec{b} \cdot \vec{d}, \vec{c} \cdot \vec{e})$. From this and (2) the thesis follows. $\square$

A corollary is that a DIO process $K = \mathrm{DIO}(f, \vec{b}, \vec{c})$ is *buffered*, in the sense that $\langle\vec{z}]K = K = K[\vec{w}\rangle$ for all $\vec{z} \subseteq \vec{b}$ and $\vec{w} \subseteq \vec{c}$.

A similar reasoning shows that the DIO process implementing f may be built by buffering a functional unit FU chosen among the class A implementations of f.

[13] The definition is correct since the actual channel represented by each $\widehat{c}$ is irrelevant. For we have $Q \backslash C = Q[b/c] \backslash ((C - \{c\}) \cup \{b\})$.

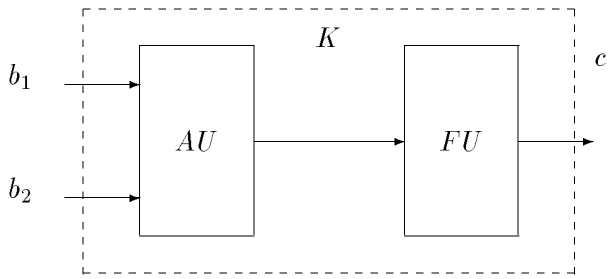

Figure 8: Internal structure of an IO process.

8 Networks of Processes

Distributivity of class A implementation requires that if each P_i in the network $P_1, \ldots, P_N$ implements the base process K_i, then $\#(P_1, \ldots, P_N)$ should implement $\#(K_1, \ldots, K_N)$. This result has already been derived for $N{=}2$ as Theorem 5.8, and may be generalized for $N{>}2$ by repeated application of Theorem 5.8. However, this is only possible if, at each application of the theorem, its proviso about channels is satisfied, which can be proved to be the case exactly when the network $K_1, \ldots, K_N$ is acyclic (as defined in Section 2.2).

The question then arises whether the notion of class A implementation could be used to deal also with cyclic networks. The answer is unfortunately negative. The reason lies in the possibility that a cycle of faulty channels may cause in the implementation network a divergence which is not present in the base network. Before strengthening the AI notion of implementation to cope with cyclic networks, we will extend it to a class of base processes wider than DIO.

9 General Input-Output Processes

The DIO process class may be generalized by considering processes for which the input message sequences determine a set of different output sequences. As an example, consider Figure 8, where K might be a process allocating resources (e.g. flight reservations) in response to requests arriving along channels b_1 and b_2. Assume that within K requests are merged at random by the arbiter AU before being forwarded to the functional unit FU. Then a fixed input at b_1 and a fixed input at b_2 may produce a finite set of different results at c.

Formally, a general input-output (IO) process K with input channels $\vec{b}$ and output channels $\vec{c}$ will be described with a *transformation*, i.e. a mapping from input sequences to (sets of) output sequences defined as follows.

Definition 9.1 Let $\vec{b}$ and $\vec{c}$ be tuples of distinct channels.
A mapping $f: dom_{\vec{b}} \rightarrow 2^{dom_{\vec{c}}}$ is a transformation (over $\vec{b}$ and $\vec{c}$) if, for all $x \in dom_{\vec{b}}$, $f(x)$ is a finite, non-empty subset of $dom_{\vec{c}}$.
The set of transformations over $\vec{b}$ and $\vec{c}$ is denoted $\mathcal{T}(\vec{b}, \vec{c})$. □

The output of an IO process K implementing a transformation f must be consistent with at least one of the results produced by f applied to the input of K. This means that K must output some initial part of this result and cannot refuse to continue with the remaining part. These requirements are formalized in the following definition as IO1 and IO2 respectively. Note that IO2 entails that an IO process should never refuse any input.

Definition 9.2 A process K with input channels $\vec{b}$ and output channels $\vec{c}$ is an IO process implementing a transformation $f \in \mathcal{T}(\vec{b}, \vec{c})$ if, for every $(t, R) \in \phi K$, there exists $y \in f(t \downarrow \vec{b})$ such that:

IO1 $t \downarrow \vec{c} \leq y$.

IO2 For every $a \in R$, $a \in \alpha\omega K$ and $(t \cdot \langle a \rangle) \downarrow \vec{c} \not\leq y$.

We denote this by $K \in \mathrm{IO}(f, \vec{b}, \vec{c})$. □

IO1, D1 in the definition of process and the image–finiteness of f entail that IO processes are divergence-free. Note that IO1-2 do not require that an IO process should generate every possible output for a given input, but only at least one (an underlying assumption is that the environment will be satisfied with any admissible output). If f is really a monotonic function, then it is easy to see that, with a slight abuse of notation, $\mathrm{IO}(f, \vec{b}, \vec{c}) = \{\mathrm{DIO}(f, \vec{b}, \vec{c})\}$. In this sense IO processes may be thought of as generalisation of DIO processes.
The following result follows immediately from the definition of IO.

Proposition 9.3 If $K \in \mathrm{IO}(f, \vec{b}, \vec{c})$ and $(s, \alpha\vec{c}) \in \phi K$ then $s \downarrow \vec{c} \in f(s \downarrow \vec{b})$. □

Let $K \in \mathrm{IO}(f, \vec{b}, \vec{i} \cdot \vec{c})$ and $L \in \mathrm{IO}(g, \vec{i} \cdot \vec{d}, \vec{e})$ (their interface is as in Figure 3). The composition of the transformations f and g is defined as follows.

Definition 9.4 Let $f \in \mathcal{T}(\vec{b}, \vec{i} \cdot \vec{c})$ and $g \in \mathcal{T}(\vec{i} \cdot \vec{d}, \vec{e})$, where all channels in the tuples are distinct. Then the transformation $f \cdot g \in \mathcal{T}(\vec{b} \cdot \vec{d}, \vec{c} \cdot \vec{e})$ is defined, for all $x \in dom_{\vec{b} \cdot \vec{d}}$, in the following way:

$$f \cdot g(x) \;=\; \{(y \Downarrow \vec{c}) \cdot z \mid y \in f(x \Downarrow \vec{b}) \wedge z \in g((y \Downarrow \vec{i}) \cdot (x \Downarrow \vec{d}))\}. \quad \square$$

It is immediate to prove that $f \cdot g$ is indeed a transformation. A proof that $K \# L$ is an IO process implementing $f \cdot g$ can be given as for the deterministic case in Section 7.1, and is therefore omitted.

9.1 Class A Implementation and Distributivity

We introduce class A implementations for transformations as we did for functions in Definition 4.1, in terms of the AI conditions, with one exception: the "$\neq$" relation of AI3 is replaced by "$\notin$". As a result, the distributivity proof differs from that of Theorem 5.8 only in those parts that refer to AI3. Therefore, only Proposition 5.7 needs a slight adaptation.

Proposition 9.5 As Proposition 5.7, with f and g being transformations, and with "$\in$" instead of "$=$".

Proof. Suppose $(t, R) \in \phi(P \# Q)$ and $t \in wf_{\vec{b}\cdot\vec{d}}$. Exactly as in the proof of Proposition 5.7, we derive:

(13) $y \in f(extr_{\vec{b}}(w))$, where $y = extr_{\vec{i}\cdot\vec{c}}(w)$.

(15) $z \in g(extr_{\vec{i}\cdot\vec{d}}(w))$, where $z = extr_{\vec{e}}(w)$.

Hence, recalling Remark 3.2, we may derive:

(16) $z \in g(\ (y \Downarrow \vec{i}) \cdot extr_{\vec{d}}(t)\ )$, by (13,15).

(17) $(y \Downarrow \vec{c}) \cdot z \in f \cdot g(extr_{\vec{b}\cdot\vec{d}}(t))$, by (13,16) and definition of $f \cdot g$.

(18) $extr_{\vec{c}\cdot\vec{e}}(t) \in f \cdot g(extr_{\vec{b}\cdot\vec{d}}(t))$, by (13,15,17).

Hence AI3 holds for $P \# Q$. □

As in the case of DIO processes, the distributivity result is easily extended to acyclic networks.

9.2 Accessibility

The accessibility property is based on the following result, which is the counterpart of Lemma 7.1.

Lemma 9.6 Let $f \in \mathcal{T}(\vec{b}, \vec{c})$ and $P \in \mathrm{AI}(f, can_{\vec{b}}, ep_{\vec{c}})$, where $ep_{\vec{c}}$ is as in Lemma 7.1. Suppose $x \notin \chi P$ and $E_x \in \mathrm{EXTR}(ep_x, x)$ for all $x \in \vec{c}$. Then

$$\#(P, E_{\vec{c}}) \in \mathrm{IO}(f, \vec{b}, \vec{c}).$$

Proof. The same reasoning as for Lemma 7.1, with the first "$=$" of (5) replaced by "$\in$", leads to deriving statements (6,7,8), i.e. the thesis.[14] □

As for DIO processes, this lemma easily enables the following accessibility result to be proved.

[14] Everything below (8) is ignored.

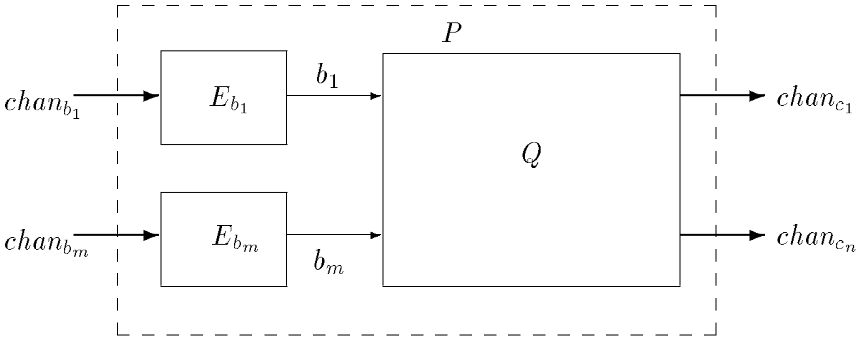

Figure 9: Internal structure of a class B implementation P.

Theorem 9.7 Let $f{\in}\mathcal{T}(\vec{b},\vec{c})$ and $P{\in}\mathrm{AI}(f, ep_{\vec{b}}, ep_{\vec{c}})$, where $\vec{b}$ and $\vec{c}$ are tuples of distinct channels not in χP. Suppose that $D_x{\in}\mathrm{DIST}(ep_x, x)$ for all $x{\in}\vec{b}$ and $E_x{\in}\mathrm{EXTR}(ep_x, x)$ for all $x{\in}\vec{c}$. Then

$$\#(D_{\vec{b}}, P, E_{\vec{c}}) \ \in \ \mathrm{IO}(f, \vec{b}, \vec{c}). \qquad \square$$

10 Class B Implementation

To deal with cyclic process networks we introduce the notion of class B implementation, which relates the implementation process to a target represented by a *superprocess*. A superprocess $\mathbf{K}$ is a non-empty set of processes having all the same input channels and the same output channels. We are therefore able to lift to superprocesses most of the notation based solely on channels, e.g. $\iota\mathbf{K}, \omega\mathbf{K}$ and $\chi\mathbf{K}$.

The purpose of a superprocess is to collect processes that are equivalent from the point of view of the system's designer. Thus, a superprocess may attain greater generality than a process in the specification of a system.

As a matter of fact, this idea is implicit in our definition of an IO process, which is regarded as admissible if it belongs to the superprocess $\mathrm{IO}(f, \vec{b}, \vec{c})$.

Note that nothing prevents the designer from giving an exact specification of his target by making it a singleton superprocess.

The other features of class B are better discussed after its formal definition.

Definition 10.1 Let $\mathbf{K}$ be a superprocess with input channels $\vec{b}$ and output channels $\vec{c}$. Suppose that for every channel x in $\vec{b}$ and $\vec{c}$, ep_x is an extraction pattern over x, and that $chan_x{\cap}chan_{x'}{=}\emptyset$ for $x{\neq}x'$.

We say a process P is a *class B implementation* of $\mathbf{K}$, and write $P{\in}\mathrm{BI}(\mathbf{K}, ep_{\vec{b}}, ep_{\vec{c}})$, if $\iota P{=}chan_{\vec{b}}$, $\omega P{=}chan_{\vec{c}}$, and there are extractors $E_x{\in}\mathrm{EXTR}(ep_x, x)$, for all $x{\in}\vec{b}$, and a process Q such that:

BI1 $\chi P{\cap}\chi\mathbf{K}{=}\emptyset$.

BI2 $\iota Q{=}\iota\mathbf{K}$ and $\omega Q{=}\omega P$.

BI3 $P{=}\#(E_{\vec{b}}, Q)$.

BI4 If $E_x{\in}\mathrm{EXTR}(ep_x, x)$ for all $x{\in}\vec{c}$, then $\#(Q, E_{\vec{c}}){\in}\mathbf{K}$. □

By BI3, it is possible to conceptually identify inside the implementation P above a set of extractors $E_{\vec{b}}$, one for each input channel $x{\in}\vec{b}$ of $\mathbf{K}$ (see Figure 9, where $\vec{b}{=}(b_1, \ldots, b_m)$ and $\vec{c}{=}(c_1, \ldots, c_n)$). What remains of P after removing these extractors is a process Q which is almost the same as a member of $\mathbf{K}$. To retrieve it, according to BI4, one only needs to apply, for each output channel $x{\in}\vec{c}$ of $\mathbf{K}$, the appropriate extractor E_x to the output channel set $chan_x$ of Q.

The notion of a network of superprocesses $\mathbf{K}_1, \ldots, \mathbf{K}_n$ is meaningful[15] and we define:

$$\#(\mathbf{K}_1, \ldots, \mathbf{K}_n) = \{\#(K_1, \ldots, K_n) \mid K_i{\in}\mathbf{K}_i \text{ for all } i\}.$$

It is immediate to see that Theorem 2.4 is easily adapted to networks of superprocesses.

Class B implementation turns out to be distributive for an arbitrary (both cyclic and acyclic) network of superprocesses, and to enjoy accessibility in the sense that:

$$\#(D_{\vec{b}}, P, E_{\vec{c}}) \in \langle\iota\mathbf{K}]\mathbf{K} = \{\langle\iota K]K \mid K{\in}\mathbf{K}\}$$

where $P, \mathbf{K}$ are as in the last definition, and $D_{\vec{b}}, E_{\vec{c}}$ are suitable disturbers and extractors, as in Theorem 7.3. As a result, if $\langle\iota\mathbf{K}]\mathbf{K}{\subseteq}\mathbf{K}$ (which is always true if $\mathbf{K}{=}\mathrm{IO}(f, \vec{b}, \vec{c})$), then:

$$\#(D_{\vec{b}}, P, E_{\vec{c}}) \in \mathbf{K}.$$

Both results, i.e. distributivity and accessibility, are an immediate consequence of Definition 10.1, Proposition 7.2 and Theorem 2.4. It thus follows that class B implementation can be used to model and analyse arbitrary process networks.

10.1 Class B and Class A

Class B implementation applies to a wider class of target systems than the IO superprocesses of class A implementation. However, if comparison is limited to IO

[15] In the sense that it is a network if any $K_1, \ldots, K_n$, where $K_i{\in}\mathbf{K}_i$ for all i, is a network.

targets, class B may well turn out to be a stronger implementation relation than class A (this would be consistent with their different distributivity properties).

This conjecture will be proved by applying class A distributivity results to the extractors $E_{\vec{b}}$ and the subprocess Q that make up, according to Definition 10.1, a class B implementation. This is possible because Q will turn out to be (nearly) a class A implementation, under some assumptions which are reasonable and always satisfied in practice, i.e. that extraction functions are monotonic, rub mappings are subset-closed and sub mappings are maximal[16]. Formally, for an extraction pattern $ep{=}(chan, msg, wf, extr, sub, rub)$ we formulate the following condition:

EP7 $extr$ is monotonic, and for all $t{\in}wf{\cap}(\alpha chan)^*$ and $X{\subseteq}\alpha chan$ the following hold:

- **(a)** If $X{\in}rub(t)$ and $X'{\subset}X$ then $X'{\in}rub(t)$.
- **(b)** If $\alpha chan{\not\subseteq}X{\cup}Y$ for all $Y{\in}rub(t)$ then $X{\in}sub(t)$.

Note that EP7 implies EP5, and both canonical and fail-stop extraction patterns satisfy EP7.

EP7 (even without part (b)) allows a useful kind of extractors to be introduced. Their trace sets and input refusals are the largest compatible with AI1-4, while their output refusals are the smallest permitted by F4 (definition of process).

Lemma 10.2 Let $ep{=}(chan, msg, wf, extr, sub, rub)$ be an extraction pattern over a channel c satisfying EP7. Define: $\iota E{=}chan$, $\omega E{=}\{c\}$, $\delta E{=}\emptyset$ and

$$\begin{aligned}\tau E &= \{t{\in}\alpha E^* \mid \forall u{\leq}t : u{\in}wf \Rightarrow u{\downarrow}c{\leq}extr(u)\}\\ \phi E &= \{(t,R){\in}\tau E{\times}2^{\alpha E} \mid (t{\in}wf \Rightarrow R{\in}rub(t)) \wedge \forall a{\in}R{\cap}\alpha c : t{\cdot}\langle a\rangle{\notin}\tau E\}.\end{aligned}$$

Then:

(a) If $t{\in}\tau E$ then $t{\cdot}\langle a\rangle{\in}\tau E$ for all $a{\in}\alpha chan$.

(b) If $t{\in}\tau E - wf$, then $t{\cdot}\langle a\rangle{\in}\tau E$ for all $a{\in}\alpha c$.

(c) $E{\in}\mathrm{EXTR}(ep, c)$.

Proof. The definition of τE immediately implies parts (b) and (together with the monotonicity of $extr$) (a) of the thesis.

To show that E is a process, note that τE is non-empty and prefix-closed, and F3 clearly holds (also due to EP7(a)). To prove F4, let $(t,R){\in}E$ and $t{\cdot}\langle a\rangle{\notin}\tau E$. By (a), $a{\in}\alpha c$, whence, easily, $(t, R{\cup}\{a\}){\in}\phi E$.

Finally, in the proof of $E{\in}\mathrm{EXTR}(ep,c)$, AI1 and AI2 are easy, while AI3 and AI4 are derived as in Proposition 6.3. □

[16] I.e. $sub(t)$ contains as many action sets as allowed by EP4.

We now show that, if P is a class B implementation of $\mathrm{IO}(f,\vec{b},\vec{c})$, then Q depicted within P in Figure 9 is nearly a class A implementation of f.

Theorem 10.3 Let $P\in\mathrm{BI}(\mathrm{IO}(f,\vec{b},\vec{c}),ep_{\vec{b}},ep_{\vec{c}})$, and let Q be as in the Definition 10.1 of class B implementation (and Figure 9). Moreover, assume that all extraction patterns in $ep_{\vec{c}}$ satisfy EP7.

Then $Q\in\mathrm{AI134}(f,can_{\vec{b}},ep_{\vec{c}})$ and satisfies the following weak version of AI2:

wAI2 If $(t,R)\in\phi Q$ and $R\notin sub_{\vec{c}}(t)$ then $R\cap\alpha\iota Q=\emptyset$.

Proof. First, define E_x for every $x\in\vec{c}$ as in Lemma 10.2 (by taking $ep{=}ep_x$) and recall that, by definition of class B implementation:

(1) $\#(Q,E_{\vec{c}})\in\mathrm{IO}(f,\vec{b},\vec{c})$.

Note that, by Lemma 10.2(a),

(2) $u\lceil chan_x\in\tau E_x$, for every trace u and $x\in\vec{c}$.

Assume $(t,R)\in\phi Q$.

To prove AI4, let $W=\{w\in\alpha\omega Q^*\mid t\cdot w\in\tau Q\}$, and assume, by contradiction, that W is infinite. Then, by Proposition 2.1, there exist $a_1,a_2,\ldots\in\alpha\omega Q$ such that, for all $k\geq 1$, $\langle a_1,\ldots,a_k\rangle\in W$, so $t_k{=}t\cdot\langle a_1,\ldots,a_k\rangle\in\tau Q$. By (2), $t_k\lceil chan_x\in\tau E_x$, for all $x\in\vec{c}$ and $k\geq 1$. Then, for all k, $t_k\in\tau(\|(Q,E_{\vec{c}}))$ by Proposition 2.2; so, by definition of hiding, $t\backslash\alpha\omega Q\in\delta(\#(Q,E_{\vec{c}}))$. With (1), this contradicts the fact that IO processes are divergence-free.

To prove AI1, suppose by contradiction that, for some $d\in\vec{c}$, $t\notin wf_d$. Choose $w\in\alpha d^*$ such that $w{\downarrow}d\not\leq y{\Downarrow}d$ for all $y\in f(t{\downarrow}\vec{b})$ (which is possible because $f(t{\downarrow}\vec{b})$ is a finite set). Then $(t\lceil chan_d)\cdot w\in\tau E_d$ by (2) and Lemma 10.2(b), and $(t\lceil chan_x)\in\tau E_x$ for $x\in\vec{c}-\{d\}$ by (2). Applying again the properties of composition and hiding, it follows that $s{=}(t\backslash\alpha\omega Q)\cdot w\in\tau(\#(Q,E_{\vec{c}}))$. Finally, by (1) and Definition 9.2, there exists $y\in f(s{\downarrow}\vec{b}){=}f(t{\downarrow}\vec{b})$ such that $s{\downarrow}\vec{c}\leq y$, i.e. $w{\downarrow}d\leq y{\Downarrow}d$, a contradiction.

We next prove AI3 and wAI2 assuming that $R\notin sub_{\vec{c}}(t)$. Thus, for every $x\in\vec{c}$, $R\notin sub_x(t)$, and by EP7(b), there exists $Y_x\in rub_x(t)$ such that:

(3) $(R\cap\alpha chan_x)\cup Y_x{=}\alpha chan_x$.

By (2), for every $x\in\vec{c}$, $(t\lceil chan_x\in)\tau E_x$, and by the definition of E_x (Lemmata 10.2 and 6.1(a,d)), there exists $w_x\in\alpha x^*$ such that:

$$w_x{\downarrow}x{=}extr_x(t) \text{ and } ((t\lceil chan_x)\cdot w_x, Y_x\cup\alpha x)\in\phi E_x.$$

Then, by the properties of composition and hiding (using (3)), for some $w\in\alpha\vec{c}^*$,

(4) $w{\downarrow}\vec{c}{=}extr_{\vec{c}}(t)$ and $((t\backslash\alpha\omega Q)\cdot w,(R\cap\alpha\iota Q)\cup\alpha\vec{c})\in\phi\#(Q,E_{\vec{c}})$.

By (1), $R\cap\alpha\iota Q{=}\emptyset$, i.e. wAI2 holds. From (1) and the last part of (4) by Proposition 9.3 it follows that $w{\downarrow}\vec{c}\in f(t{\downarrow}\vec{b})$. This, with the first part of (4), implies AI2. $\square$

It is now possible to prove the promised result, that—for IO target superprocesses—class B is a stronger implementation relation than class A.

Theorem 10.4 Let $P \in \mathrm{BI}(\mathrm{IO}(f, \vec{b}, \vec{c}), ep_{\vec{b}}, ep_{\vec{c}})$ and all extraction patterns in $ep_{\vec{c}}$ satisfy EP7. Then:

$$P \in \mathrm{AI}(f, ep_{\vec{b}}, ep_{\vec{c}}).$$

Proof. Let P, $E_{\vec{b}}$ and Q be as in Definition 10.1. ¿From the distributivity result for acyclic networks of IO processes, it follows that $P' = \|(E_{\vec{b}}) \in \mathrm{AI}(id, ep_{\vec{b}}, can_{\vec{b}})$ for the appropriate identity transformation id.[17] Thus, by Theorem 10.3 and Propositions 5.4 and 5.5, $P \in \mathrm{AI14}(ep_{\vec{b}}, ep_{\vec{c}})$.

It then suffices to observe that we can apply Propositions 5.6 and 9.5 to P', Q and $P' \# Q = P$. Note that the hypotheses (a) and (b) of Proposition 9.5 are to be derived from those of Proposition 5.7. Indeed, if in Proposition 5.6 we take $\vec{j}$ empty then hypothesis (a) obviously holds, and hypothesis (b) holds because $\vec{d}$ is also empty. As a result, $P \in \mathrm{AI2}(ep_{\vec{b}}, ep_{\vec{c}})$. For Proposition 9.5 we take $\Psi = (\alpha chan_{\vec{i}})^*$; then hypothesis (a) clearly holds, and hypothesis (b) holds by Theorem 10.3. Moreover, $id \cdot f = f$, which implies $P \in \mathrm{AI3}(f, ep_{\vec{b}}, ep_{\vec{c}})$. $\square$

The same technique may be employed to prove an analogous result when **K** is a singleton DIO process.

11 Implementing Replication

In this section we address the problem of assembling N copies (some of which possibly faulty) of a base process P, in order to build a replicated implementation P^N of P. Here an important issue is the interplay between the fault assumptions and the extraction strategy in terms of which P^N implements P. We restrict our treatment to the extraction which is nearly always used in practice, namely majority voting. The design goal then becomes the classical NMR one: to tolerate a minority of faulty copies. We first introduce an extraction pattern for majority voting.

11.1 Majority Voting

We need some auxiliary notation. Let c be a channel and $c[1], \ldots, c[N]$ be distinct channels which differ from c but share its message set. For $I \subseteq \{1, \ldots, N\}$, define the channel set $c^I = \{c[i] \mid i \in I\}$ and abbreviate $c^{\{1,\ldots,N\}}$ as c^N. If $\vec{c} = (c_1, \ldots, c_k)$ then $\vec{c}\,[i] = (c_1[i], \ldots, c_k[i])$.

Let $nf \subseteq \{1, \ldots, N\}$ be such that $|nf| > \frac{N}{2}$ ($|nf|$ is the cardinality of set nf). The majority extraction pattern $maj(c, nf)$ for c and nf is $(c^N, \mu c, wf, mv, sub, rub)$, where:

[17] In other words, P' is a class A implementation of a set of parallel buffers.

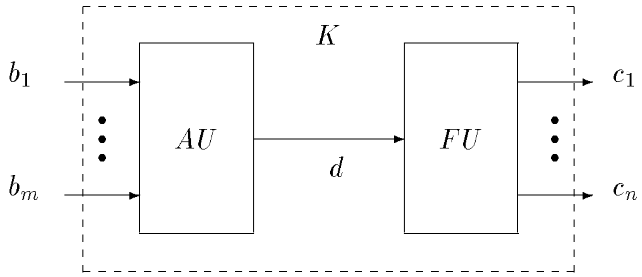

Figure 10: Internal structure of the base process K.

M1 $wf=\{t\in(\alpha c^N)^* \mid \forall i,j\in nf : t{\downarrow}c[i]\leq t{\downarrow}c[j] \vee t{\downarrow}c[j]\leq t{\downarrow}c[i]\}$.

M2 $mv(t)$ is the longest prefix common to a majority of channels in c^N, i.e the longest $u\in\mu c^*$ such that $|\{i \mid u\leq t{\downarrow}c[i]\}| > \frac{N}{2}$.

M3 $sub(t)=\{S\subseteq\alpha c^N \mid S\cap\alpha c^{nf}\neq\alpha c^{nf}\}$.

M4 $rub(t)=\{R\subseteq\alpha c^N \mid R\cap\alpha c^{nf}=\emptyset\}$.

Here, M2 clearly models majority voting. Like NF in the fail-stop case, c^{nf} is intended to contain non-faulty channels; these should carry consistent message sequences (M1) and refuse no input (M4), like the corresponding input channels of IO base processes. Finally, suppose two processes P and Q refuse S and R respectively at their shared channels c^N; then, given M4, M3 is the weakest condition to prevent communication between P and Q from blocking at c^N.

It is worth noting that $maj(c,nf)$ satisfies EP5, EP6 and EP7.

11.2 The Base Processes

Implementation by replication will be applied to base processes made of a series of an arbiter unit AU with a functional unit FU (see Figure 10). AU accepts messages at the input channels in $\vec{b}$, merges them at random and outputs them at d, prefixing with $b!$ the generic message input at $b\in\vec{b}$ (this choice will be motivated later). FU is a DIO process mapping input at d onto output sequences at channels in the tuple $\vec{c}$. Formally:

Assumption 11.1 In the following it is assumed:

1. $K=AU\#FU$.
2. $FU=\mathrm{DIO}(f,d,\vec{c})$, where $f\in\mathcal{F}(b,\vec{c})$.

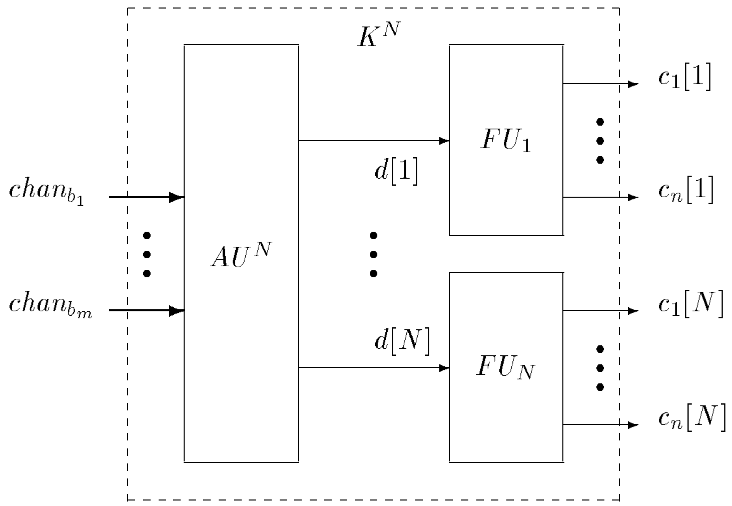

Figure 11: Replicating the base process of Figure 10.

3. $AU \in \mathrm{IO}(merge, \vec{b}, d)$.
4. $\mu d = \alpha \vec{b}$ and $merge \in \mathcal{T}(\vec{b}, d)$ maps $x \in dom_{\vec{b}}$ onto $\{u \in (\alpha \vec{b})^* \mid u \downarrow \vec{b} = x\}$.
   E.g. if $\vec{b} = (b_1, b_2)$, then

$$merge(\langle 2,7\rangle, \langle 5\rangle) = \{\langle b_1!2, b_1!7, b_2!5\rangle, \langle b_1!2, b_2!5, b_1!7\rangle, \langle b_2!5, b_1!2, b_1!7\rangle\}.$$

11.3 Architecture of the Replicated Implementation

To construct an implementation K^N of the base process K in Figure 10, an implementation AU^N of the arbiter unit AU is assembled together with N (possibly faulty) copies of the functional unit FU, as shown in Figure 11.

Input to AU^N is to be interpreted according to an extraction pattern ep_b for each input channel $b \in \vec{b}$ of AU. Output from AU^N is issued by the channel set d^N, which replicates d of AU, and interpreted through a majority extraction pattern (output from $d[i]$ for $i \in nf \subseteq N$ is assumed to be correct).

Each $d[i]$, for $1 \le i \le N$, feeds a process FU_i, which has an output channel $c[i]$ for each output channel $c \in \vec{c}$ of FU. FU_i is assumed to behave like a renamed copy of FU if $i \in nf$, and to be possibly faulty otherwise.

It is intended that, if output from each c^N is interpreted through the appropriate majority extraction pattern, K^N will turn out to be an implementation of K.

The above discussion is formalized by the following assumption.

Assumption 11.2

1. $K^N = AU^N \# FU^N$, where $FU^N = \|(FU_1, \ldots, FU_N)$.
2. For $1 \le i \le N$, $\iota FU_i = \{d[i]\}$ and $\omega FU_i = \vec{c}\,[i]$.
3. $nf \subseteq \{1, \ldots, N\}$, $|nf| > \frac{N}{2}$ and $FU_i = \mathrm{DIO}(f, d[i], \vec{c}\,[i])$ for all $i \in nf$.
4. For $x \in \vec{b}$, ep_x is an extraction pattern.
5. $ep_d = maj(d, nf)$.
6. $AU^N \in \mathrm{AI}(merge, ep_{\vec{b}}, ep_d)$.
7. For $x \in \vec{c}$, $ep_x = maj(x, nf)$. □

Note that the input extraction patterns $ep_{\vec{b}}$ may be arbitrary, but in practice are likely to be of the majority kind, so as to accept the output of other replicated processes.

Moreover, in real replicated architectures, AU^N will contain communication links and N voters, each receiving every message in multiple copies. More specifically, it may be assumed that if a non-faulty voter within AU^N *acquires* a message, i.e. receives it in more than $\frac{N}{2}$ copies, the same must happen for all the other non-faulty voters. In the model of Figure 11, the ith non-faulty module may be imagined to forward the messages acquired to the functional unit FU_i along channel $d[i]$. Note that, due to differences in communication and processing delays, voters may acquire messages in different orders, but must forward them all in the same order to functional units, or these would not behave consistently. The order agreed upon by the voters is in general unpredictable, which explains the introduction of the nondeterministic merger AU in the base process.

Formally, the unique order requirement has been captured rather elegantly by constraining communication along d^N with the majority extraction pattern (cf. M1). This motivates the choice of having a single channel d between AU and FU in the base process.[18]

11.4 Class A Replicated Implementation

Assumption 11.2 is actually not sufficient to prove that K^N is a class A implementation of K, and must be strengthened as follows. Since, ideally, replicated components should all behave in the same way, it will be assumed that, if the non-faulty output channels of AU^N refuse to extend their message sequences, then these must coincide (see the definition of Ψ below). We also assume that the output of a generic replica of FU cannot grow unboundedly without being interrupted by some input. Note that for non-faulty replicas this is ensured by their being DIO. Formally:

[18] Note that the behaviour of a process with channels $d_1, \ldots, d_k$ is easily reproduced by another process with channel d in lieu of $d_1, \ldots, d_k$, simply by encoding an action $d_i!v$ of the former by the action $d!(d_i!v)$ of the latter.

Assumption 11.3

1. $\Psi = \{u \in (\alpha d^N)^* \mid \forall i, j \in nf :\ u{\downarrow}d[i] = u{\downarrow}d[j]\}$.
2. If $t \in \tau FU_i$ and $i \notin nf$, then there are finitely many traces $w \in (\alpha\omega FU_i)^*$ such that $t \cdot w \in \tau FU_i$.
3. If $(t, \alpha d^{nf}) \in AU^N$ and $t \in wf_{\vec{b}}$, then $t \lceil d^N \in \Psi$. □

This allows a lemma on FU^N to be proved.

Lemma 11.4 Under Assumptions 11.1, 11.2 and 11.3, we obtain:

(a) $FU^N \in \mathrm{AI124}(ep_d, ep_{\vec{c}})$.

(b) If $(t, R) \in \phi FU^N$ and $t \lceil d^N \in \Psi$ and $R \notin sub_{\vec{c}}(t)$, then $mv_{\vec{c}}(t) = f(mv_d(t))$, where $mv_{\vec{c}}(t)$ denotes $(mv_{c_1}, \ldots, mv_{c_n})$.

Proof. Assume $(t, R) \in \phi FU^N$ and $t \in wf_d$, i.e.

(1) $t{\downarrow}d[i] \leq t{\downarrow}d[j]$ or $t{\downarrow}d[j] \leq t{\downarrow}d[i]$ for all $i, j \in nf$.

First, observe that $\delta FU_i = \emptyset$ for all i, otherwise Assumptions 11.2(3) and 11.3(2) would be contradicted. Therefore, by Proposition 2.2, $\delta FU^N = \emptyset$, and:

(2) For all $i \leq N$, $(t \lceil \chi FU_i, R \cap \alpha FU_i) \in \phi FU_i$.

To derive AI1 for FU^N, let $i, j \in nf$ and by (1), without loss of generality, assume $t{\downarrow}d[i] \leq t{\downarrow}d[j]$. Then, by monotonicity of f and by FU_i, FU_j being DIO processes:

$$f(t{\downarrow}d[i]) \leq f(t{\downarrow}d[j]) \ \text{ and } \ t{\downarrow}\vec{c}\,[i] \leq f(t{\downarrow}d[i]) \ \text{ and } \ t{\downarrow}\vec{c}\,[j] \leq f(t{\downarrow}d[j]).$$

From this it follows that for all $x \in \vec{c}$, $t{\downarrow}x[i] \leq t{\downarrow}x[j]$ or $t{\downarrow}x[j] \leq t{\downarrow}x[i]$, i.e. $t \in wf_x$, as required by AI2.

For AI2, use (2) and the DIO properties, to infer $R \cap \alpha d^{nf} = \emptyset$.

For AI4, let $W = \{w \in (\alpha\omega FU^N)^* \mid t \cdot w \in \tau FU^N\}$ and note that by $\delta FU^N = \emptyset$ and Proposition 2.2, for all i,

$$W \lceil \chi FU_i = \{u \in (\alpha\omega FU_i)^* \mid (t \lceil \chi FU_i) \cdot u \in \tau FU_i\}$$

and the latter set is finite by Assumptions 11.2(3) and 11.3(2). Thus W must be also finite.

For part (b) of the thesis, assume $t \lceil d^N \in \Psi$ and $R \notin sub_{\vec{c}}(t)$. From this and $|nf| > \frac{N}{2}$, for $i \in nf$, we derive $t{\downarrow}d[i] = mv_d(t)$. ¿From this and (2), using $FU_i \in \mathrm{DIO}(f, d[i], \vec{c}\,[i])$:

(3) $t{\downarrow}\vec{c}\,[i] = f(mv_d(t))$.

Thus for all $x \in \vec{c}$ and $i, j \in nf$, $t{\downarrow}x[i] = t{\downarrow}x[j]$. Hence, by $|nf| > \frac{N}{2}$, $mv_{\vec{c}}(t) = t{\downarrow}\vec{c}\,[i]$ for any $i \in nf$ which, with (3), yields (b). □

We are now ready to prove the desired result.

Theorem 11.5 Under Assumptions 11.1, 11.2 and 11.3 we obtain:

$$K^N = AU^N \# FU^N \in \mathrm{AI}(merge{\cdot}f, ep_{\vec{b}}, ep_{\vec{c}}).$$

Proof. By Assumption 11.2(6), $AU^N{\in}\mathrm{AI124}(ep_{\vec{b}}, ep_d)$, and, by Lemma 11.4(a), $FU^N{\in}\mathrm{AI124}(ep_d, ep_{\vec{c}})$. Thus, by Propositions 5.4, 5.5 and 5.6 (taking $\vec{j}=\vec{i}$), we obtain $K^N{\in}\mathrm{AI124}(ep_{\vec{b}}, ep_{\vec{c}})$. We then exploit Proposition 9.5 with Ψ defined by Assumption 11.3(1), Lemma 11.4(b) and Assumptions 11.2(6) and 11.3(3), to conclude that $K^N{\in}\mathrm{AI3}(merge{\cdot}f, ep_{\vec{b}}, ep_{\vec{c}})$. □

11.5 Class B Replicated Implementation

We now consider class B replicated implementation. Base and implementation processes have the same structure as in Figures 10 and 11. We retain the previous assumptions, strengthening 11.2(6) by choosing AU^N among class B implementation of *merge* transformation.

Theorem 11.6 Under Assumptions 11.1, 11.2 and 11.3, but replacing 11.2(6) by $AU^N{\in}\mathrm{BI}(\mathrm{IO}(merge, \vec{b}, d), ep_{\vec{b}}, ep_d)$, we obtain that:

$$\begin{array}{rcl} AU \# FU & \in & \mathrm{IO}(merge{\cdot}f, \vec{b}, \vec{c}) \\ AU^N \# FU^N & \in & \mathrm{BI}(\mathrm{IO}(merge{\cdot}f, \vec{b}, \vec{c}), ep_{\vec{b}}, ep_{\vec{c}}). \end{array}$$

Proof. The first part of the thesis is easy.
From the definition of class B implementation and Theorem 10.3, it follows that there are extractors $E_x{\in}\mathrm{EXTR}(ep_x, x)$ for all $x{\in}\vec{b}$, and a process Q, $\iota Q{=}\vec{b}$ and $\omega Q{=}d^N$, such that $AU^N{=}\#(E_{\vec{b}}, Q)$, $Q{\in}\mathrm{AI134}(merge, can_{\vec{b}}, ep_d)$ and:

(1) If $(t, R){\in}\phi Q$ and $R{\cap}\alpha d^{nf}{=}\alpha d^{nf}$ then $R{\cap}\alpha x{=}\emptyset$, for all $x{\in}\vec{b}$.

By proceeding as in the proof of Theorem 11.5, with Q replacing AU^N, we can show that

$$Q \# FU^N \in \mathrm{AI134}(merge{\cdot}f, can_{\vec{b}}, ep_{\vec{c}}).$$

To show that $Q\#FU^N{\in}\mathrm{AI2}(can_{\vec{b}}, ep_{\vec{c}})$ also holds, we use (1), Lemma 11.4(a) and Proposition 5.6 with $\vec{j}{=}\vec{i}{=}(d)$. As a result,

$$Q \# FU^N \in \mathrm{AI}(merge{\cdot}f, can_{\vec{b}}, ep_{\vec{c}}).$$

Let $E_x{\in}\mathrm{EXTR}(ep_x, x)$ for all $x{\in}\vec{c}$. By the accessibility Lemma 9.6 for IO processes we obtain $\#(Q\#FU^N, E_{\vec{c}}) \in \mathrm{IO}(merge{\cdot}f, \vec{b}, \vec{c})$ from which the thesis follows. □

12 Concluding Remarks

12.1 Main Results

We have provided a formal treatment of communicating replicated processes. Central to it is the choice of a criterion whereby a replicated system may be said to implement a target, or base, process. The two criteria proposed—formalized as class A and class B implementation—enjoy accessibility and distributivity (over acyclic and cyclic networks respectively). Distributivity is exploited, both in design and verification, to ensure that a network of communicating replicated processes implements the network of their respective base processes; thus a "global" (network) property may be reduced to a "local" one. Accessibility requires that the behaviour of a base process may be derived from that of its replicated implementation; since the former behaviour also describes (in the CSP model adopted) nondeterminism, deadlock and divergence properties, the replicated implementation may be said to be totally[19] correct with respect to the base. In contrast, the trace semantics of [MP88] only allows the study of partial (functional) correctness.

A remark is in order about the presence of divergences in the CSP model employed: this is necessary for the fundamental Theorem 2.4 on hierarchically combining process networks. Besides, while divergence is generally regarded as a drawback of CSP—because it may arise in situations which are intuitively acceptable—its treatment is instead well-suited to the study of fault tolerance. For it permits us to require that the composition of two fault-prone implementations should not diverge if the composition of their targets does not (cf. Lemma 5.3(a)). This correctly prevents, e.g., the faulty channels shared by two replicated implementations from engaging in infinite chatter.

Concerning the structure of the treatment, it is worth noting the central role played by distributivity properties. Besides having their own significance, they allow a natural and easy proof of two important results:

1. for IO base superprocesses, class B implementation is a stronger relation than class A implementation (Theorem 10.4);

2. the replicated process of Figure 11 is a class A implementation of the intended base process (Theorem 11.5).

These results respectively exploit two other interesting ones:

3. the class B implementation of an IO superprocess contains a class A implementation (Theorem 10.3);

[19] At least in the sense of finite observation semantics.

4. a set of base processes, a majority of which is nonfaulty, is nearly a class A replicated implementation of the base process (Lemma 11.4).

All this seems to justify the conclusion that class A replicated implementation, while not provably the weakest, is certainly a fundamental notion.

Another advantage of our approach is the degree of generality and modularity achieved by defining implementation relations independently of replication or any other particular fault-masking measure; knowledge of the latter is encapsulated within the extraction patterns that appear as parameters in the implementation relation. Different fault masking techniques may be accommodated simply by parametrizing the relation with the appropriate extraction patterns; thus, it is even possible to specify that different techniques are employed for different parts of a system.

As a matter of fact, in our approach fault masking is just a form of abstraction—other forms with different goals are possible and may be treated within our framework. The particular abstraction chosen is applied to the behaviour of a system in order to decide whether it is an implementation of the intended target system. Of course this can also be said of every verification framework in which the specification is represented by a target system. However, we believe that the form of abstraction we can deal with is more general, at least under some respects, than those proposed so far.

12.2 Related Work

The earliest correctness criterion for replicated systems is the replica coordination of [Lam78, Sch90] (all replicas must process the same sequence of requests), which we studied in Section 11. While extremely important in practice, replica coordination cannot be viewed as a general form of abstraction; as observed in [Sch90], more general replicated implementations can be built within the theories of [Aiz89] and [MP88] (a predecessor of this work).

The notion of implementation employed in [Aiz89] is that of [AL91], which evolved from [Lam83]. According to it, system behaviour is described by sequences of state tuples with an internal component; it is required that for every state sequence of a correct implementation there is one of the target such that the two sequences coincide after deleting the internal component from their states. A similar approach is proposed in [Jon94], which uses infinite action sequences (i.e. infinite traces) instead of state sequences; the interface of the target must be a subset of the interface of the implementation, and it is required that every trace of the implementation can be turned into one of the target by deleting from it actions not belonging to the target. In our view, the choice of an action-oriented rather than a state-oriented description may affect ease of specification, but not expressive power (see [DNV] for a formal connection between the two description styles). Thus, we will collectively describe the latter two approaches and other comparable ones (like e.g. [LT87, Sta88]) as based on abstraction *by hiding*.

In contrast, our notion of abstraction is essentially based on the interpretation of traces over a set of channels as traces over another channel. Since abstraction by hiding amounts (in the action-based approach) to interpreting internal action sequences as the empty trace, it could appear to be an instance of abstraction by interpretation; in fact, this can be shown not to be the case, essentially because such an interpretation does not preserve any information (but abstraction by hiding can be easily modelled in CSP by the hiding operator). Conversely, it should be obvious that abstraction by interpretation cannot be reduced to abstraction by hiding.

Therefore, a conclusive comparison of these approaches to abstraction is not easy. Here, we will just try to illustrate how abstraction by hiding has been applied to the analysis of a replicated system in [Jon94] and [Aiz89]. Neither work models faulty replicas, as we instead do, for arbitrary faults, in Section 11.

In [Jon94], the target system has a non-replicated architecture but a replicated interface; this makes its description less concise and intuitively acceptable than that of a fully non replicated target; bridging the gap between these two kinds of targets is exactly the goal of our abstraction by interpretation approach. Moreover, since the interface common to target and implementation is not subject to abstraction, it is hard to see how arbitrary faults made by the implementation at this interface can be modelled without their surfacing in the behaviour of the target.

In [Aiz89], the internal state components of the target and the implementation differ: only the latter is replicated, which makes it possible to specify partly faulty implementations. The external state components, instead, must coincide and are non-replicated. This is alright for the description of the target, but raises a question about that of the implementation: if its external state is literally non-replicated, it would seem that two implementations that interact through a shared subset of their external state cannot do so in a replicated, fault-tolerant way; on the other hand, if the non-replicated external state is just conceived as an abstraction of the actual replicated state, some kind of abstraction by interpretation is needed for a thorough formalization.

The conclusion seems to be that in both examples abstraction by hiding is not fully satisfactory, basically owing to its requirement that the externally visible behaviours of implementation and target should exactly coincide. The solution would seem to lie in abstraction by interpretation.

It is also interesting to compare the implications of the action-based concurrency models underlying our work and [Jon94] (a comparison with state-based models would be too involved).

The model of [Jon94] employs infinite traces; this affords the ability to describe asynchronously communicating I/O systems that are slightly more general than ours, not being tied to a particular (mathematical) relation between input and output. Infinite traces also allow liveness to be expressed and verified.

In our treatment, finite traces and refusals permit asynchronous as well as synchronous communication, and arbitrary I/O systems to be described. However, we have deliberately restricted our attention to asynchronously communicating, relational I/O *target* systems (removing the latter restriction is possible [KMP93, KMP92], but leads to a less rich set of results; removing the former is a topic further work). What is more important, though, we place no constraint on the faulty parts of the *implementation* system: faulty channels need not communicate asynchronously, and faulty modules need not be I/O systems. This is certainly desirable, but is not possible in the model of [Jon94] or in other classical models of asynchronous communication such as [KM77] or [Bro88]. While liveness in the sense of [Jon94] and, in general, temporal logic cannot be expressed with our finite traces, the ability to place constraints on refusals is usually deemed to be a reasonable alternative in CSP circles. Illustrating this controversy is certainly out of the scope of this paper; here we will only say that the virtues of both approaches may perhaps be retained by casting our ideas in a branching time semantic setting, which looks like a rather daunting task.

The approach of [SH93] to the formal analysis of fault-tolerance is quite different from ours, which basically provides a criterion whereby a system, albeit fault-prone, may be abstractly viewed as an implementation of a correct target system. In contrast, in [SH93] the system under study may embed fault-prone components, but its externally observable behaviour must be correct itself, without the filter of abstraction. As in this paper, systems are modelled as CSP processes; the trace model adopted, however, permits only partial correctness to be studied.

The interface displacement implementation proposed in [BJO91] resembles our class B implementation, which accounts for two interesting similarities: (1) their interface transducers play a role comparable to that of our extractors and disturbers; (2) both treaments find a need to constrain the system's and the environment's mutual refusals (this is obtained by the *sub*-*rub* pair in this paper, by introducing refusal pairs in [BJO91]). The essential difference is that their treatment is geared to the refinement of target systems into exact implementations, ours to modelling the unpredictability of a fault-prone environment.

Finally, [Lam86] is a fundamental work that also regards implementation as a form of abstraction more general than simple abstraction by hiding. In it, this idea is developed in a relativistic, partial-order based setting (quite far from our divergence process model), and is then applied to an investigation on the nature of interprocess communication (rather than fault-tolerant systems).

Acknowledgment

The first author was supported by the Esprit WG Caliban, while the other authors were supported by the Italian CNR and MURST.

References

[Aiz89] J. Aizikowitz. *Designing Distributed Services Using Refinement Mappings.* PhD thesis, Computer Science Dept., Cornell University, Ithaca, New York, 1989.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[Bir85] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. of the 10th ACM Symp. on Operating Systems Principles*, pages 79–86, 1985.

[BJO91] E. Brinksma, B. Jonsson, and F Orava. Refining interfaces of communicating systems. In Abramsky and Maibaum, editors, *Proc. Coll. on Combining Paradigms for Software Development*, Lecture Notes in Computer Science, vol. 494, pages 297–312. Springer-Verlag, 1991.

[BR85] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes et al., editors, *Seminar on Concurrency*, Lecture Notes in Computer Science, vol. 197, pages 281–305, Carnegie-Mellon University, 1985. Springer-Verlag.

[Bro88] M. Broy. Nondeterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[Coo85] E. Cooper. Replicated distributed programs. In *Proc. of the 10th ACM Sym. on Operating Systems Principles*, pages 63–78, Washington, 1985.

[DN87] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[DNV] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM.* To appear.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Jon94] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, 1994.

[KM77] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North-Holland, 1977.

[KMP92] M. Koutny, L. V. Mancini, and G. Pappalardo. Replication in acyclic networks of communicating processes. Technical Report 378, Computing Laboratory, The University of Newcastle upon Tyne, 1992.

[KMP93] M. Koutny, L. V. Mancini, and G. Pappalardo. Modelling replicated processing. In *Proceedings of PARLE 93 Parallel Architectures in Europe Conference*, Lecture Notes in Computer Science, vol. 694. Springer-Verlag, 1993.

[Lam78] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2(2):95–114, 1978.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[Lam86] L. Lamport. The mutual exclusion problem: Part I—a theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.

[LS90] M. Little and S. K. Shrivastava. Replicated K-resilient objects in Arjuna. In *Proc. of IEEE Intl. Workshop on the Management of Replicated Data*, pages 53–58, 1990.

[LT87] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of the 6th ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[Man86] L. V. Mancini. Modular redundancy in a message passing system. *IEEE Transactions on Software Engineering*, 12(1):79–86, 1986.

[MP88] L. V. Mancini and G. Pappalardo. Towards a theory of replicated processing. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, vol. 331, pages 175–192, Warwick, 1988. Springer-Verlag.

[Sch84] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[SH93] H. Schepers and J. Hooman. Trace-based compositional reasoning about fault-tolerant systems. In A. Bode, M. Reeve, and G. Wolf, editors, *Proceedings of PARLE 93 Parallel Architectures in Europe Conference*, Lecture Notes in Computer Science, vol. 694. Springer-Verlag, 1993.

[Sta88] E. W. Stark. Proving entailment between conceptual state specifications. *Theoretical Computer Science*, 56:135–154, 1988.