Asynchronous Communication in Process Algebra (Extended Abstract)

Frank S. de Boer[*], Jan Willem Klop[†‡], Catuscia Palamidessi[†§]

Abstract

We study the paradigm of asynchronous process communication, as contrasted with the synchronous communication mechanism which is present in process algebra frameworks such as CCS, CSP and ACP. We investigate semantics and axiomatizations with respect to various observability criteria: bisimulation, traces and abstract traces. Our aim is to develop a process theory which can be regarded as a kernel for languages based on asynchronous communication, like data flow, concurrent logic languages and concurrent constraint programming.

1 Introduction

In order to introduce the framework of asynchronous communication that will be adopted and investigated in this paper, we will first give an informal comparison with synchronous communication as in ACP [BK86]. Synchronous communication is modeled in ACP by a binary function $|$ on actions. In the case of value transmission the typical equation is $c{\uparrow}d \mid c{\downarrow}d = c{\Updownarrow}d$, where the actions $c{\uparrow}d$ and $c{\downarrow}d$ are interpreted as "send datum d along channel c" and "receive datum d at channel c" respectively, and $c{\Updownarrow}d$ represents the completion of the communication action. As an example, consider the parallel composition

$$\partial_{\{c\downarrow d,\, c\uparrow d\}}(a \cdot c{\downarrow}d \cdot b \parallel a' \cdot c{\uparrow}d \cdot b').$$

Here ∂ is the encapsulation operator, enforcing the intended communication by acting as a garbage collector for attempted but failed communications. According to the axioms of ACP this expression corresponds to the process graph represented in Figure 1(a). In this figure, the dashed parts express the unintended traces that are pruned by $\partial_{\{c\downarrow d,\, c\uparrow d\}}$.

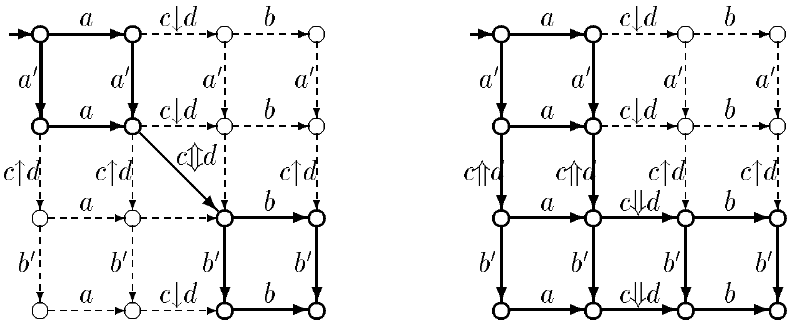
Note that there is no difference in directionality between the communication partners $c{\downarrow}d, c{\uparrow}d$. In the asynchronous setting, on the other hand, there is an asymmetry between $c{\downarrow}d$ and $c{\uparrow}d$: one of them, $c{\uparrow}d$, is now supposed to precede its partner $c{\downarrow}d$, for an indeterminate amount of 'time' (i.e. number of steps). Due to the temporal split between $c{\uparrow}d$, $c{\downarrow}d$, we need now two actions signifying their completion; let them be $c{\Downarrow}d$ and $c{\Uparrow}d$, respectively. Furthermore, some memory mechanism, in the form of a state operator, must now be available to convey the effect

[*]Department of Computer Science, Eindhoven Technical University P.O. Box 513, 5600 MB Eindhoven, The Netherlands email: wsinfdb@tuewsd.win.tue.nl

[†]CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands email: jwk@cwi.nl, katuscia@cwi.nl

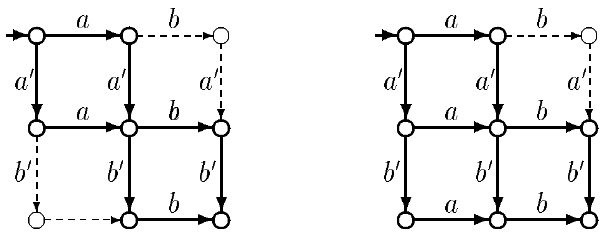
[‡]Department of Computer Science, Free University, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

[§]Department of Computer Science, State University of Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands



(a) (b)

Figure 1: Synchronous versus asynchronous communication.



(a) (b)

Figure 2: Synchronous versus asynchronous communication with abstraction.

of the event $c{\uparrow}d$ during the interval from $c{\uparrow}d$ to $c{\downarrow}d$. Let us now consider the asynchronous counterpart of the example above. The evaluation of

$$\mu_c^{\epsilon}(a \cdot c{\downarrow}d \cdot b \parallel a' \cdot c{\uparrow}d \cdot b'),$$

where μ is the state operator, by using the axioms in $\mathrm{aprPA}_{\delta,\mu}$ (see Table 2) gives now the process graph in Figure 1(b). Again the dashed parts express the unintended traces, originating by free interleaving, that are pruned away by μ.

Just as in the synchronous case, one might wish to abstract from these completed actions. The representation of the processes from the example above 'shrinks' by this abstraction to the process graphs in Figure 2.

Note that both the operators ∂, μ, can be viewed as 'actualization' operators, that make the process actually run. So we refer to $c \Uparrow d, c \Downarrow d$ actions as being *completed*, or *realized*, and to $c{\uparrow}d, c{\downarrow}d$ actions as being *intended*, or *possible*.

In specifying the send and receive data transmission scheme there is a choice to be made, concerning the nature of the channels. In this paper we will treat the two main cases of 'sequential access' and 'random access'. Of the first we have chosen as a typical representant the queue (but we may also have investigated the case of stack); in the second, the bag is a natural choice.

Our point of departure is the paper [BKT85] where the send/receive mechanism is axiomatized, both for queues and bags as channels. We show that this axiomatization is complete for the 'minimal' setting of bisimulation semantics. Here the specific nature of the communication channels does not yet influence the axiomatization which can be given in a uniform way.

The next notion we investigate is the 'maximal trace-respecting congruence'[1]. We give a characterization of this congruence in terms of a fully abstract model which is a version of the usual failure semantics for synchronous communication ([BHR84]), and we provide a complete axiomatization. Remarkably, for this more abstract notion the above uniformity between queues and bags disappears.

Finally we consider traces with abstraction from the completed communication actions. As usual, the phenomenon of abstraction introduces some quite intricate problems. At this point the difference between bags and queues becomes very prominent: the case of bags requires the introduction of many new axioms, thus identifying more expressions. Therefore we can call the latter theory (for bags) more abstract than the former (for queues). A priori this was to be expected since a bag in itself is already an 'abstraction' of a queue.

1.1 Comparison with related work

A theory for asynchronous communication has also been developed in [JHJ90]. That theory is more abstract than ours; the main reasons are that in [JHJ90] there are, first, some restrictions on processes to be composed in parallel, and, second, communication is modeled in such a way that when a process receives an item from a buffer, that item remains available for the other processes. Such a mechanism can be implemented in our language by means of a copying process; therefore all the contexts which can be specified in [JHJ90] can be specified also in our language. On the other hand, some of our contexts cannot be defined in [JHJ90] (i.e. our congruence is *strictly* less coarse).

The relatively 'low degree of abstraction' of our language has the advantage that we can regard it as a kernel for the axiomatization of other languages based on asynchronous communication. It is possible to show, in fact, that with respect to both observability criteria we have considered in this paper our axioms are correct for data flow [Kah74], concurrent logic programming [Sha89], and concurrent constraint programming [Sar89]. In other words, the communication mechanisms of those languages can be implemented in ours. Completeness can then be obtained by adding some specific axioms.

The fact that we construct a fully abstract model based on failure sets might look surprising, and in contrast with the claim, often made, that asynchronous communication does *not* require refusal information. In fact, recent studies have shown that several languages based on asynchronous communication have linear models: see for instance [Jon85, Jos90] for data flow, [BP90] for concurrent logic languages, [BP91] for concurrent constraint programming and [BKPR91] for a general semantic framework based on reactive sequences. The explanation of this apparent contradiction is that also these last models encode some hidden refusal information. In this paper we have opted for a model in which the refusals are explicitly represented because it facilitates the proof of the completeness of the axiomatization.

2 Bisimulation semantics for asynchronous communication

In this section, that is the basis of our paper, our starting point is [BKT85]. After fixing the notation and the syntax of the language to be considered, we provide a set of transition rules defining its operational semantics.

[1]By 'traces' we mean the 'completed' ones, i.e. the traces associated to the transition sequences which end in a point from which no further transitions are possible.

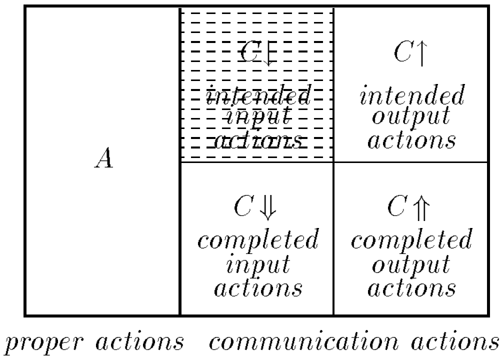


Figure 3: The set of actions. Shaded: dependent actions; unshaded: independent actions.

2.1 Syntax

The set of actions is structured as follows.

1. A is a set of *proper* actions, notation: $a, b, a_1, a_2, \ldots$.

2. Let D be a set of *data*: $d, d_1, d_2 \ldots$, and let C be a set of *channel* names: $c, e, c_1, c_2, \ldots$. For all $c \in C$ and $d \in D$ we have *communication* actions

   - $c{\downarrow}d$ (intended input action)
   - $c{\uparrow}d$ (intended output action)
   - $c{\Downarrow}d$ (completed input action)
   - $c{\Uparrow}d$ (completed output action)

   The set of all intended input actions is $C{\downarrow}$; likewise $C{\uparrow}, C{\Downarrow}, C{\Uparrow}$ contain the intended output actions, the completed input actions and the completed output actions respectively.

The action alphabet *Act* is now defined to be

$$Act = A \cup C{\downarrow} \cup C{\uparrow} \cup C{\Downarrow} \cup C{\Uparrow}.$$

We will write $u, v, \ldots$ for general actions from Act. Intended input actions from $C{\downarrow}$ are also called *dependent* actions (*Dep*); all the other actions are *independent* (*Ind*) (see Figure 3). In Section 2.2 we give motivation for this terminology.

The set *Exp* of (process) expressions is generated by the following grammar:

$$x ::= \mid \ \delta \ \mid \ u \cdot x \ \mid \ x_1 + x_2 \ \mid \ x_1 \parallel x_2 \ \mid \ x_1 \mathbin{\|\!\lfloor} x_2 \ \mid \ \mu_c^\sigma(x) \ \mid \ \rho_f(x).$$

The constant δ represents the inaction or *deadlock*. For any $u \in Act$, $u\,\cdot$ is an *action prefixing*. The operators $+$, $\parallel$ and $\mathbin{\|\!\lfloor}$ are the *sum* (or alternative choice), the *merge* (or parallel composition), and the *left-merge* respectively. For any $c \in C$ and state σ, μ_c^σ is an *encapsulation operator* (σ represents the 'initial state' of c). Finally for $f : Act \to Act$ respecting the partition of Act, ρ_f is a *renaming* operator.

Bracket conventions are employed as usual. Instead of $u \cdot x$ we also will write $u\,x$. Often we will omit the 'end marker' δ in an expression, and render e.g. $a\,\delta$ as just a.

2.2 Operational semantics

In order to define the operational semantics of our processes in a uniform way, i.e. not depending upon the specific kind of channels that are intended, we treat channels as given by some abstract data type. To describe the meaning of the actions $c{\downarrow}d$ and $c{\uparrow}d$ we introduce two operations get: $State \times D \rightarrow State \cup \{\bot\}$ and put: $State \times D \rightarrow State$, where $State$ is the set of states (of the channel), and $\bot$ stands for *undefined.* The understanding is that $get(\sigma, d)$ checks if the datum d is available in σ and in that case it retrieves it. It is undefined otherwise. This is the reason why we call the actions in $C{\downarrow}$ 'dependent': their being enabled or not depends upon the content of the buffers, hence upon the actions performed by the environment. The operation $put(\sigma, d)$ in general modifies σ by adding the datum d. We assume that it is always defined, and this is the reason why we call the actions in $C{\uparrow}$ 'independent'. One could wonder what happens, in the case of bounded channels, when the channel is full. We assume that in such a case the datum is simply lost. The reason is that in general a test on the state of the channel, before perfoming a send action, is very expensive to implement. However, our theory can be modified so as to include such a test by adding the possibility that $put(\sigma, d)$ is undefined and by treating the actions in $C{\uparrow}$ as dependent.

The operational semantics of an expression $x \in Exp$ is defined by the transition system T in Table 1, where x, y, z are meta-variables ranging over Exp.

To each process expression $x \in Exp$ the transition system T assigns a *transition graph* (also called *process graph*), as illustrated in the examples shown in Figure 4. The symbol ϵ denotes the state of an empty channel. We denote by $\doteq$ the well known *bisimulation equivalence*, which identifies expressions with bisimilar transition graph (disregarding the labels in the nodes). Examples of bisimilar processes are the pairs (i), (ii) and (v) in Figure 4.

2.3 Axioms

The bisimulation equivalence $\doteq$ is a congruence and it is completely axiomatized by the system $\mathrm{aprPA}_{\delta,\mu}$ in Table 2, which is the one from [BKT85] restricted to action prefixing. In our paper we don't consider sequential composition, just for reasons of technical convenience. We expect that all results in the present paper can be obtained also in that more general case, but at the cost of sometimes having to include 'degenerate' versions of some axioms. The name $\mathrm{aprPA}_{\delta,\mu}$ means 'action prefixing process algebra with constant δ and operator μ'; it is based on the nomenclature proposal in [BB91].

Theorem 2.1 *Let $x, y \in Exp$. Then*

$$x \doteq y \ \ iff \ \ \mathrm{aprPA}_{\delta,\mu} \vdash x = y$$

3 Trace semantics

In this section we study the observability criterion which consists of observing all the possible traces generated by a process in a sequence of transition steps. It is convenient to introduce the following notation. For $x, y \in Exp, s \in Act^*$, $x \overset{s}{\twoheadrightarrow} y$ represents a sequence of transition steps from x to y and it is defined as follows:

1. $x \overset{\lambda}{\twoheadrightarrow} x$ (where λ is the empty sequence),
2. if $x \overset{u}{\longrightarrow} y$ then $x \overset{u}{\twoheadrightarrow} y$,

R1	$u\,x \xrightarrow{u} x$	
R2	$\dfrac{x \xrightarrow{u} y}{x + z \xrightarrow{u} y}$	$\dfrac{x \xrightarrow{u} y}{z + x \xrightarrow{u} y}$
R3	$\dfrac{x \xrightarrow{u} y}{x \parallel z \xrightarrow{u} y \parallel z}$	$\dfrac{x \xrightarrow{u} y}{z \parallel x \xrightarrow{u} z \parallel y}$
R4	$\dfrac{x \xrightarrow{u} y}{x \mathbin{\lfloor\!\lfloor} z \xrightarrow{u} y \parallel z}$	
R5	$\dfrac{x \xrightarrow{c\uparrow d} y}{\mu_c^{\sigma}(x) \xrightarrow{c\Uparrow d} \mu_c^{\sigma'}(y)}$	if $\sigma' = put(\sigma, d)$
R6	$\dfrac{x \xrightarrow{c\downarrow d} y}{\mu_c^{\sigma}(x) \xrightarrow{c\Downarrow d} \mu_c^{\sigma'}(y)}$	if $\sigma' = get(\sigma, d) \neq \perp$
R7	$\dfrac{x \xrightarrow{u} y}{\mu_c^{\sigma}(x) \xrightarrow{u} \mu_c^{\sigma}(y)}$	if $u \neq c\uparrow d, c\downarrow d$
R8	$\dfrac{x \xrightarrow{u} y}{\rho_f(x) \xrightarrow{f(u)} \rho_f(y)}$	

Table 1: The transition system T.

FIGURE figjw2.tex

(v)

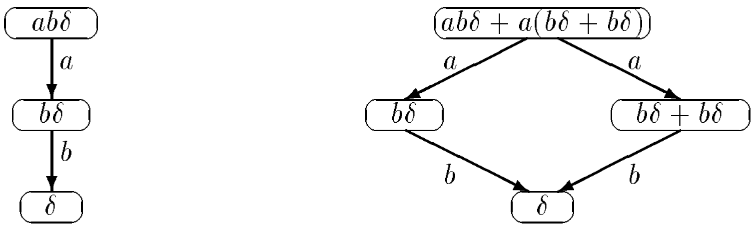


Figure 4: Examples of transitions.

A1	$x + y = y + x$
A2	$(x + y) + z = x + (y + z)$
A3	$x + x = x$
A4	$x + \delta = x$
M1	$x \parallel y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x$
M2	$\delta \mathbin{\lfloor\!\lfloor} x = \delta$
M3	$u\,x \mathbin{\lfloor\!\lfloor} y = u\,(x \parallel y)$
M4	$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$
E1	$\mu_c^\sigma(\delta) = \delta$
E2	$\mu_c^\sigma(u\,x) = u \cdot \mu_c^\sigma(x)$ if $u \neq c{\downarrow}d, c{\uparrow}d$
E3	$\mu_c^\sigma(c{\uparrow}d \cdot x) = c{\Uparrow}d \cdot \mu_c^{\sigma'}(x)$ if $\sigma' = put(\sigma, d)$
E4	$\mu_c^\sigma(c{\downarrow}d \cdot x) = c{\Downarrow}d \cdot \mu_c^{\sigma'}(x)$ if $\sigma' = get(\sigma, d) \neq \perp$
E5	$\mu_c^\sigma(c{\downarrow}d \cdot x) = \delta$ if $get(\sigma, d) = \perp$
E6	$\mu_c^\sigma(x + y) = \mu_c^\sigma(x) + \mu_c^\sigma(y)$
RN1	$\rho_f(\delta) = \delta$
RN2	$\rho_f(u\,x) = f(u) \cdot \rho_f(x)$
RN3	$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$

Table 2: $\mathrm{aprPA}_{\delta,\mu}$.

3. if $x \overset{s}{\twoheadrightarrow} y$ and $y \overset{t}{\twoheadrightarrow} z$ then $x \overset{st}{\twoheadrightarrow} z$.

Now we can define the traces of a process and the corresponding equivalence relation.

Definition 3.1 For $x \in \mathit{Exp}$, the *traces* of x are given by:

$$\mathrm{T}(x) \;=\; \{s \mid \exists y.\, x \overset{s}{\twoheadrightarrow} y \not\rightarrow\}$$

The notation $y \not\rightarrow$ indicates that for no u, z we have $y \overset{u}{\longrightarrow} z$.

The relation $\sim_{\mathrm{T}}$ is the equivalence induced by the traces:

$$x \sim_{\mathrm{T}} y \;\text{ iff }\; \mathrm{T}(x) = \mathrm{T}(y).$$

This relation is not a congruence, essentially due to the presence of the encapsulation operator μ which enforces a dependency upon the state of the channels, as determined by the initial state and by the actions performed by the environment.

Example 3.2

1. The processes x and y represented by the transition graphs in Figure 3 have the same traces, but their encapsulation with respect to the channel c yields different results: $\mathrm{T}(\mu_c^\epsilon(x)) = \{ab\}$, whereas $\mathrm{T}(\mu_c^\epsilon(y)) = \{a, ab\}$.

2. The processes x and y represented in Figure 3, have the same traces, but if we now consider them in parallel with the process z then encapsulation with respect to c, e yields a difference: $e \Uparrow d \cdot c \Uparrow d \cdot c \Downarrow d \in \mathrm{T}(\mu_c^\epsilon \mu_e^\epsilon(y \parallel z)) \setminus \mathrm{T}(\mu_c^\epsilon \mu_e^\epsilon(x \parallel z))$.

In the following, $\mathcal{C}[\,]$ indicates an arbitrary (unary) context, i.e. an expression containing one occurrence of a 'hole'. We define the congruence relation $\equiv_{\mathrm{T}}$ as the *maximal trace-respecting congruence*, i.e.

$$x \equiv_{\mathrm{T}} y \;\text{ iff }\; \forall \mathcal{C}[\,].\, \mathcal{C}[x] \sim_{\mathrm{T}} \mathcal{C}[y].$$

In the rest of this section we will investigate a concrete model corresponding to this congruence, and its axiomatization.

3.1 Failure semantics

It has been shown in [BKO88] that in the synchronous case of ACP with one-to-one communication *failure semantics* ([BHR84]) is fully abstract with respect to trace equivalence. We will show how to modify it so that this property is maintained. We first construct the failure model for bags and then we show how to adapt it to the case of queues.

Let us first show why the standard failure model is not fully abstract.

Example 3.3 Figure 7 shows three examples of pairs of processes x, y which are not identified by the standard failure model; yet their traces are the same in every context.

The main reason why the expressions x and y in Figure 7 are trace-congruent is that the independent actions are always enabled regardless of the state. This is not true for the input actions.

40 x y ∘∘a70a70a70∘∘∘$c{\downarrow}d$70b70$c{\downarrow}d$70b70∘∘∘∘

Figure 5: Example of trace-equivalent processes x and y which can be distinguished by a μ-context.

40 x y z
∘∘∘$c{\uparrow}d$70$c{\uparrow}d$70$c{\uparrow}d$70$e{\uparrow}d$70∘∘∘∘$c{\downarrow}d$70$e{\downarrow}d$70$c{\downarrow}d$70$e{\downarrow}d$70$c{\downarrow}d$70$c{\downarrow}d$70∘∘∘∘∘∘a70a70a70a70a70∘∘∘∘∘

Figure 6: Example of processes x and y which can be distinguished by a parallel process z.

Example 3.4 Figure 8 shows that the input counterparts x' and y' of the processes x and y in Figure 7 can be distinguished by a parallel process z. In all cases there is a trace generated by $\mu_c^\epsilon(y' \parallel z)$, which is not generated by $\mu_c^\epsilon(x' \parallel z)$.

The relevant information is associated with the *resting points*, those points from where the process can only proceed by input actions, and consists of the trace which leads to such a point and the state which make such a point *stable*, i.e., none of the initial input actions are enabled. As such, we can represent a state by a subset of the complement of the ready set, i.e., the set of initial actions, associated with a resting point. We call such a representation a *refusal set*. However, the intuition is that when an input action $c{\downarrow}d$ occurs in the refusal set this indicates that the value d is available on channel c.

Definition 3.5 The *failure set* of x is

$$\mathrm{F}[\![x]\!] \;=\; \{\langle s, R\rangle \mid \;\exists y.\; x \stackrel{s}{\twoheadrightarrow} y,\; \mathit{Init}(y) \subseteq \mathit{Dep},\; R \subseteq \mathit{Dep} \setminus \mathit{Init}(y)\}$$

where $\mathit{Init}(x) = \{u \mid \;\exists y.\; x \xrightarrow{u} y\}$.

The relation $\equiv_{\mathrm{F}}$ is the equivalence induced by the failure sets, i.e.

$$x \equiv_{\mathrm{F}} y \;\text{ iff }\; \mathrm{F}[\![x]\!] = \mathrm{F}[\![y]\!]$$

This relation $\equiv_{\mathrm{F}}$ is actually a congruence. To show this, it is sufficient to prove that the definition of F is compositional with respect to all the operators of the language, i.e. every operator of the language has a semantical counterpart. We give only the definition for action prefixing and encapsulation, the other operators are as in the standard case.

Definition 3.6 Let P, Q range over the domain of F.

Prefixing $\widetilde{u\cdot} P \;=\; \begin{cases} \{\langle us, R\rangle \mid \;\langle s, R\rangle \in P\} & \text{if } u \notin \mathit{Dep} \\ \{\langle us, R\rangle \mid \;\langle s, R\rangle \in P\} \;\cup\; \{\langle \lambda, R\rangle \mid \; R \subseteq \mathit{Dep} \setminus \{u\}\} & \text{otherwise,} \end{cases}$

Encapsulation. We first define the operator $\tilde{\mu}_c^\sigma \,:\; \mathit{Act}^* \to \mathit{Act}^* \cup \mathit{Act}^*\delta$ as the operator on traces which transforms communication actions on c into completed ones, and the auxiliary operator $\mathit{St}_c^\sigma \,:\; \mathit{Act}^* \to \mathit{State}$ which computes the current state (on c) at every point of the trace:

Figure 7: Examples of trace-congruent processes x, y with different standard failure semantics.

$$
\begin{array}{lllllll}
\tilde{\mu}^\sigma_c(\lambda) & = & \lambda & St^\sigma_c(\lambda) & = & \sigma & \\
\tilde{\mu}^\sigma_c(u\,s) & = & u\cdot\tilde{\mu}^\sigma_c(s) & St^\sigma_c(u\,s) & = & St^\sigma_c(s) & \text{if } u\neq c{\uparrow}d, c{\downarrow}d \\
\tilde{\mu}^\sigma_c(c{\uparrow}d\cdot s) & = & c{\Uparrow}d\cdot\tilde{\mu}^{\sigma'}_c(s) & St^\sigma_c(c{\uparrow}d\cdot s) & = & St^{\sigma'}_c(s) & \text{if } \sigma'=put(\sigma,d) \\
\tilde{\mu}^\sigma_c(c{\downarrow}d\cdot s) & = & c{\Downarrow}d\cdot\tilde{\mu}^{\sigma'}_c(s) & St^\sigma_c(c{\downarrow}d\cdot s) & = & St^{\sigma'}_c(s) & \text{if } \sigma'=get(\sigma,d)\neq\perp \\
\tilde{\mu}^\sigma_c(c{\downarrow}d\cdot s) & = & \delta & St^\sigma_c(c{\downarrow}d\cdot s) & = & \perp & \text{if } get(\sigma,d)=\perp
\end{array}
$$

Next we extend $\tilde{\mu}^\sigma_c$ on pairs as follows:

$$
\begin{array}{rcll}
\tilde{\mu}^\sigma_c(P) & = & \{\langle\mu^\sigma_c(s),R\rangle \mid & \exists\langle s,R'\rangle\in P. \\
 & & & R\subseteq R'\cup\{c{\downarrow}d \mid\ d\in D\}, \\
 & & & St^\sigma_c(s)\neq\perp, \\
 & & & (c{\downarrow}d\notin R'\Rightarrow St^\sigma_c(s\cdot c{\downarrow}d)=\perp)\}
\end{array}
$$

The failure semantics is correct with respect to the traces, in fact $\mathrm{T}(x)=\{s \mid\ \langle s,Dep\rangle\in\mathrm{F}[\![x]\!]\}$. Since $\equiv_{\mathrm{F}}$ is a trace-respecting congruence, it is at least as fine as $\equiv_{\mathrm{T}}$ (which, by definition, is the maximal trace-respecting congruence).

3.2 Axioms

We now present the axioms to add to $\mathrm{aprPA}_{\delta,\mu}$ in order to obtain a complete axiomatization of the failure semantics introduced above.

In the synchronous case a complete axiomatization for failure semantics has been obtained by adding to the theory of ACP the ready axioms and the failure axiom ([BKO88]) which, restricted to the case of aprPA_δ, are respectively the axioms R and S in Table 3. Our failure semantics is at least as abstract as that semantics in [BKO88], in fact it can be retrieved by considering (only) those pairs in which the refusal set contains all independent actions. Therefore the axioms R and S are valid in our semantics. On the other hand, our semantics is strictly more abstract because it cancels branching points from which an independent action exits. Axiomatically this is modeled by factorizing the plus operator with respect to the independent actions: see axiom I in Table 3, where i stands for any independent action, i.e. $i\in Act\setminus C{\downarrow}$.

Note that if we generalize i in the axiom I to be any action we obtain a system equivalent to the one presented in [vGla90] as the axiomatization of (completed) trace semantics for aprPA_δ.

Theorem 3.7 (Completeness) *For all $x,y\in Exp$ we have*

$$x\equiv_{\mathrm{F}} y \ \ \textit{iff}\ \ \mathrm{aprPA}_{\delta,\mu}\cup\{\mathrm{R},\mathrm{S},\mathrm{I}\}\vdash x=y$$

3.3 Full abstraction of failure semantics

In this section we show how to specialize the failure semantics given above to the cases of random access (bags) and sequential access (queues).

Bags. In the case of bags the fully abstract semantics is just the failure semantics F.

Proposition 3.8 (Full abstraction) *If channels are bags then* $\equiv_{\mathrm{T}} \subseteq \equiv_{\mathrm{F}}$.

Corollary 3.9 *If channels are bags then for* $x, y \in Exp$ *the following are equivalent.*

1. $x \equiv_{\mathrm{F}} y$,
2. $x \equiv_{\mathrm{T}} y$,
3. $\mathrm{aprPA}_{\delta,\mu} \cup \{\mathrm{R}, \mathrm{S}, \mathrm{I}\} \vdash x = y$.

Queues. The distinguishing feature of sequential access is that only one of the items stored in a channel can be read and consumed. As a consequence, the failure semantics as defined above is not fully abstract. For instance, it distinguishes the processes $x = a \cdot c{\downarrow}d_1 + a \cdot c{\downarrow}d_2 + a \cdot c{\downarrow}d_3$ and $y = a(c{\downarrow}d_1 + c{\downarrow}d_2) + a(c{\downarrow}d_2 + c{\downarrow}d_3) + a(c{\downarrow}d_3 + c{\downarrow}d_1)$ which in case of sequential access are observationally identical. Since for each channel only one item is relevant for the observable behaviour of a process, we must consider only those refusal sets in which no more than one input on the same channel is present.

Definition 3.10 The *queue failure set* of $x \in Exp$ is

$$\mathrm{F}^{\mathrm{Q}}[\![x]\!] \;=\; \{\langle s, R\rangle \in \mathrm{F}[\![x]\!] \mid \; c{\downarrow}d, c{\downarrow}d' \in R \Rightarrow d = d'\}$$

Let $\equiv_{\mathrm{F^Q}}$ be the associated equivalence relation. It is possible to show that $\equiv_{\mathrm{F^Q}}$ is a congruence and that it is as coarse as $\equiv_{\mathrm{T}}$. We will specify now a subset of the language for which the completeness holds. Consider the subset Exp' of expressions in which an input action on a channel c can occur only in contexts of the form $\Sigma_{d\in D} c{\downarrow}d \cdot x_d$. So, the typical feature of this sublanguage is that it cannot perform anymore a blocking test on the presence of a specific datum. Expressions of this forms are used in Process Algebra to implement 'input on a variable', i.e. CSP-like actions of the form $c{\downarrow}V \cdot x[V]$, where V is a variable of type D. For expressions in Exp' the semantics F and $\mathrm{F^Q}$ are the same, i.e. $\mathrm{F^Q}[\![x]\!] = \mathrm{F^Q}[\![y]\!]$ *iff* $\mathrm{F}[\![x]\!] = \mathrm{F}[\![y]\!]$. Therefore we obtain:

Corollary 3.11 *If channels are queues then for* $x, y \in Exp'$ *the following are equivalent*

1. $x \equiv_{\mathrm{F^Q}} y$,
2. $x \equiv_{\mathrm{F}} y$,
3. $x \equiv_{\mathrm{T}} y$,
4. $\mathrm{aprPA}_{\delta,\mu} \cup \{\mathrm{R}, \mathrm{S}, \mathrm{I}\} \vdash x = y$.

4 Abstraction from communication

In this section we investigate an axiomatic characterization of the *proper* behaviour of a process, i.e., we want to encapsulate all communications and treat them as invisible (silent) steps. To this end we extend the set *Act* to contain a special internal action τ, which corresponds to the standard notion of silent step in languages like CCS and ACP. In the following, we will use the notation $\mu(x)$ as an abbreviation for the encapsulation of the process x with respect to all channels, with initial content empty, i.e., if $C = \{c_1, \ldots, c_n\}$, then $\mu(x) = \mu^{\epsilon}_{c_1} \ldots \mu^{\epsilon}_{c_n}(x)$.

Figure 8: Examples of processes x', y' which can be distinguished by a process z.

$$
\begin{array}{ll}
\mathrm{R} & u\,(v\,x_1 + y_1) + u\,(v\,x_2 + y_2) \;=\; u\,(v\,x_1 + v\,x_2 + y_1) + u\,(v\,x_1 + v\,x_2 + y_2) \\
\mathrm{S} & u\,x + u\,(y + z) \;=\; u\,x + u\,(x + y) + u\,(y + z) \\
\mathrm{I} & u\,(i\,x + y) \;=\; u\,(i\,x + y) + u\,i\,x
\end{array}
$$

Table 3: The failure axioms for asynchronous communication.

Furthermore, we use the notation $\rho(x)$ to denote the process x where all the completed actions are renamed into τ. To define the proper traces we need an operator α which removes from a trace $s \in Act^*$ all the τ actions. The proper traces of a process are then defined as follows:

$$\mathrm{T}_\tau(x) \;=\; \alpha \circ \mathrm{T}(\rho(\mu(x)))$$

The relation $\sim_{\mathrm{T}_\tau}$ is the equivalence induced by the proper traces:

$$x \sim_{\mathrm{T}_\tau} y \;\text{ iff }\; \mathrm{T}_\tau(x) = \mathrm{T}_\tau(y)$$

The congruence relation $\equiv_{\mathrm{T}_\tau}$ is the maximal proper-trace-respecting congruence:

$$x \equiv_{\mathrm{T}_\tau} y \;\text{ iff }\; \forall \mathcal{C}[\,].\, \mathcal{C}[x] \sim_{\mathrm{T}_\tau} \mathcal{C}[y]$$

We now investigate an axiomatization for $\equiv_{\mathrm{T}_\tau}$. The axioms for τ-abstraction which have been studied for the failure semantics (and therefore for the maximal proper-trace-respecting congruence) in the presence of synchronous communication ([BKO88]) are, of course, still valid in our asynchronous case. Restricted to the case of action prefixing only (i.e. no general sequential composition) these axioms are the ones shown in Table 4.

Note that the presence of T1 and T2 makes the axiom S of Table 3 superfluous (cf. [BKO88]).

However, these axioms for τ-abstraction are not complete in the presence of independent actions. When τ is prefixed to an independent action i, it can be deleted. Formally, this is captured by the axiom in Table 5. In the presence of the axiom T3, the axiom I of Table 3 can be shown to be derivable.

Furthermore we observe that processes like $c{\uparrow}d \cdot a$ and $c{\uparrow}d \cdot a + a \cdot c{\uparrow}d$ are now observably equivalent. This identification is captured axiomatically by the law in Table 6.

Note that the reverse of this law is not correct (an output cannot be anticipated). For instance, $a \cdot c{\uparrow}d$ and $a \cdot c{\uparrow}d + c{\uparrow}d \cdot a$ are not observably equivalent. A distinguishing context is $\mathcal{C}[\,] = c{\downarrow}d \cdot b \parallel [\,]$. The process $\mathcal{C}[a \cdot c{\uparrow}d]$ will produce ab only, whereas $\mathcal{C}[a \cdot c{\uparrow}d + c{\uparrow}d \cdot a]$ will produce additionally ba.

In the full paper we show that the above axiom system completely characterizes T_τ in the case of queues.

T1	$u\,\tau\,x$	$=$	$u\,x$
T2	$\tau\,x + y$	$=$	$\tau\,x + \tau\,(x+y)$

Table 4: τ-abstraction laws for the failure semantics in the synchronous case.

T3	$\tau\,i\,x = i\,x$

Table 5: Additional τ-abstraction law for independent actions.

Theorem 4.1 (Completeness for queues) *For all $x, y \in Exp'$ we have*

$$x \equiv_{\mathrm{T}_\tau} y \;\; \mathit{iff} \;\; \mathrm{aprPA}_{\delta,\mu} \cup \{\mathrm{R}, \mathrm{T1}-3, \mathrm{OP}\} \vdash x = y$$

However, when we consider abstraction from communication in the case of bags, many additional identifications have to be made. First we observe that we have to abstract from the order between (intended) output actions, i.e., the processes $c{\uparrow}d \cdot e{\uparrow}d'$ and $e{\uparrow}d' \cdot c{\uparrow}d$ are observably equivalent. A simple axiomatization of this phenomenon can be given in terms of multisets of output actions, so-called *combined* actions, which we denote by $\overline{c}$. The intended meaning of a combined action is that of a concurrent execution of its components. In Table 7 the multiset union is denoted by $\sqcup$.

Another class of identifications stems from the fact that in the case of bags we are not in general able to detect whether a process intends to read an item.

Example 4.2 The process $a \cdot c{\downarrow}d$ is observably equivalent to the process $a \cdot c{\downarrow}d + a$. Note that in the case of queues these processes can be distinguished by the process $c{\uparrow}d \cdot c{\uparrow}d' \cdot c{\downarrow}d' \cdot b$ $(d \neq d')$, which will always produce b when put in parallel with the first process, while the input action $c{\downarrow}d'$ can be blocked in case the second process chooses the branch a.

Example 4.3 Another characteristic example is that of the process $c{\downarrow}d \cdot e{\downarrow}d' + e{\downarrow}d'$ which is observably equivalent to $e{\downarrow}d' \cdot c{\downarrow}d + c{\downarrow}d$. Note that $c{\downarrow}d \cdot e{\downarrow}d'$ and $e{\downarrow}d' \cdot c{\downarrow}d$ are *not* observably equivalent: $c{\downarrow}d \cdot e{\downarrow}d'$ in parallel with $c{\uparrow}d \cdot c{\downarrow}d \cdot a$ can produce the empty sequence while putting $c{\uparrow}d \cdot c{\downarrow}d \cdot a$ in parallel with $e{\downarrow}d' \cdot c{\downarrow}d$ will always produce a.

These and other related identifications we describe in terms of combined input actions, which we denote by $\underline{c}$. Below we give an informal explanation of the axioms of Table 8.

Axiom I1 introduces combined input actions, and states that, when both $\underline{c}$ and $\underline{e}$ are enabled, the process $\underline{c}\,(\underline{e}\,x + y)$ can always select the branch $\underline{e}\,x$. It should be noted that the equality $\underline{c}\,(\underline{e}_0\,x_0 + \underline{e}_1\,x_1) = (\underline{c} \sqcup \underline{e}_0)\,x_0 + (\underline{c} \sqcup \underline{e}_1)\,x_1$ does *not* hold: the first process, when put in parallel with $\overline{c} \cdot \underline{c} \cdot a$ can generate the empty sequence by "stealing" the data.

OP	$c{\uparrow}d\,(a\,x+y) = c{\uparrow}d\,(a\,x+y) + a\cdot c{\uparrow}d\cdot x$

Table 6: Delay axiom for output actions.

O	$\overline{c}\,\overline{e}\,x \;=\; (\overline{c}\sqcup\overline{e})\,x$

Table 7: Abstraction axiom for output actions.

Axiom I2 allows to determine an order in the input actions of $\underline{c}\sqcup\underline{e}$, say, executing $\underline{c}$ before $\underline{e}$, in the presence of an alternative starting with $\underline{c}$.

Axiom I3 can be informally justified as follows: suppose the process $u\,x$ deadlocks after u. In that case the process $u\,(\underline{c}\,\delta + x)$ either deadlocks immediately after u or it can select the branch $\underline{c}\,\delta$ and so will deadlock eventually.

Axiom I4 The auxiliary operator $\circ$ in this axiom reveals the "hidden eager nature" of a process. It is axiomatized in Table 9.

Axiom I5 Here i is to be understood to range over I, j over J, and k over $K \subseteq \{i_j \mid i \in I, j \in J\}$. This axiom can be justified as follows: suppose the process $\Sigma_k \underline{c_k}\, x_k$ deadlocks after u, so all the $\underline{c_k}$ are disabled. This deadlock possibility then should be covered by a process $u\cdot\Sigma_j \underline{c_{i_j}}\, x_{i_j}$, that is, all the $\underline{c_{i_j}}$ are disabled, for some i. Otherwise we would have that there exists for every $i \in I$ a $j \in J$ such that $\underline{c_{i_j}}$ is enabled, which, in its turn, would imply the existence of an enabled $\underline{c_k}$.

The axioms of Table **??** allow abstraction from the order of certain events.

The first axiom of Table **??** allows the abstraction from an output action immediately followed by its corresponding input. The second axiom is a slightly stronger version of the first one. The last axiom states that putting a read datum immediately back, is essentially, unobservable.

In the full paper we prove the completeness of the above axiom system for normal processes, i.e., processes in which there occur no combined input or output actions. The main idea of this proof is the semantic modelling of the additional identifications introduced by each axiom by means of a corresponding closure condition. We then show that for the resulting compositional model $\mathrm{F}^{\mathrm{B}}_{\tau}$ the following holds: for every process x there exists a process y such that $\mathrm{F}^{\mathrm{B}}_{\tau}[\![x]\!] = \mathrm{F}[\![y]\!]$ and $x = y$ is derivable from the axioms.

References

[BB91] J.C.M. Baeten and J.A. Bergstra. A survey of axiom systems for process algebras. Tech. rep., (CWI), Amsterdam, 1991.

I1	$\underline{c}\,(\underline{e}\,x+y) \;=\; \underline{c}\,(\underline{e}\,x+y)+(\underline{c}\sqcup\underline{e})\,x$
I2	$\underline{c}\,(\underline{e}\,x+y)+\underline{c}\,y \;=\; (\underline{c}\sqcup\underline{e})\,x+\underline{c}\,y$
I3	$u\,(\underline{c}\,\delta+x) \;=\; u\,(\underline{c}\,\delta+x)+u\,x$
I4	$u\,(\underline{c}\,x+y)=u\,(\underline{c}\,x+y)+u\,(\underline{c}\circ x+y)$
I5	$\Sigma_i u\cdot\Sigma_j\underline{c}_{i_j}\,x_{i_j} = \Sigma_i u\cdot\Sigma_j\underline{c}_{i_j}\,x_{i_j}+u\cdot\Sigma_k\underline{c}_k\,x_k$ $(\forall f\in I\rightarrow J.\ \exists\underline{c}_k\subseteq\bigsqcup_i\underline{c}_{i_{f(i)}})$

Table 8: Abstraction axioms for input actions.

$$\underline{c}\circ a\,x=\underline{c}\,a\,x$$

$$\underline{c}\circ\underline{e}\,x=(\underline{c}\sqcup\underline{e})\,x$$

$$\underline{c}\circ(x+y)=\underline{c}\circ x+\underline{c}\circ y$$

Table 9: Axioms for the operator $\circ$.

[BHR84] S.D. Brookes, C.A.R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *Journal of ACM*, 31:499–560, 1984.

[BK86] J.A. Bergstra and J.W. Klop. Process algebra: specification and verification in bisimulation semantics. In *Mathematics and Computer Science II*, CWI Monographs, pp. 61 – 94. North-Holland, 1986.

[BKO88] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Readies and failures in the algebra of communicating processes. *SIAM J. on Computing*, 17(6):1134 – 1177, 1988.

[BKPR91] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures: Towards a paradigm for asynchronous communication. In J.C.M. Baeten and J.F. Groote, editors, *Proc. of CONCUR 91*, LNCS 527, pp. 111 – 126. Springer-Verlag, 1991.

[BKT85] J.A. Bergstra, J.W. Klop, and J.V. Tucker. Process algebra with asynchronous communication mechanisms. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Proc. Seminar on Concurrency*, LNCS 197, pp. 76 – 95. Springer-Verlag, 1985.

[BP90] F.S. de Boer and C. Palamidessi. Concurrent logic languages: Asynchronism and language comparison. In *Proc. of the North American Conference on Logic Programming*, pp. 175–194. The MIT Press, 1990.

[BP91] F.S. de Boer and C. Palamidessi. A fully abstract model for Concurrent Constraint Programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP*, LNCS 493, pp. 296–319. Springer-Verlag, 1991.

[vGla90] R.J. van Glabbeek The linear time - branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proc. of CONCUR 90*, LNCS 458, pp. 278 – 297. Springer-Verlag, 1990.

[Jon85] B. Jonsson. A model and a proof system for asynchronous processes. In *Proc. of the 4th ACM Symp. on Principles of Distributed Computing*, pp. 49–58, 1985.

[Jos90] M.B. Josephs. Receptive process theory. Tech. rep. CS 90/8, Eindhoven University of Technology, 1990. To appear in *Acta Informatica*.

[JHJ90] M.B. Josephs, C.A.R. Hoare, and He Jifeng. A theory of asynchronous processes. Tech. rep., Oxford University Computing Laboratories, 1990.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proc. of IFIP Congress*, pages 471–475, New York, 1974. North-Holland.

[Sar89] V.A. Saraswat. *Concurrent Constraint Programming languages*. PhD thesis, Carnegie-Mellon University, January 1989. To be published by The MIT Press.

[Sha89] E.Y. Shapiro. The family of Concurrent Logic Programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.