Generic Process Algebras for Asynchronous Communication

F.S. de Boer[1] and G. Zavattaro[2]

[1] Utrecht University, Department of Computer Science, The Netherlands.
Email: frankb@cs.ruu.nl

[2] Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy.
Email: zavattar@cs.unibo.it

Abstract. We study at different levels of abstraction general semantic and algebraic properties of languages which are based on asynchronous communication. These different levels of abstraction concern the specific nature of the communication mechanism. At the highest level we introduce a process algebra which characterizes asynchronous communication in general, that is, when abstracting from the specific nature of the communication mechanism. This generic process algebra we further instantiate to algebras for different classes of languages. Considered are classes of languages which are based on a general monoid structure of the actions and classes of languages which are based on read/write operations.

1 Introduction

In this paper we investigate the semantics and equational theories, or process algebras as we call them, for the general paradigm of asynchronously communicating processes which has been introduced in [5]. This paradigm encompasses such diverse systems as described by concurrent logic languages [23], concurrent constraint languages [22], imperative languages in which processes communicate by means of shared variables [13], or asynchronous channels [18], dataflow languages [19], and coordination languages [15].

These systems have in common that processes communicate via some shared data structure. The asynchronous nature of the communication lies in the way access to this shared data structure is modelled: the data structure is updated by means of write primitives that have free access whereas the read primitives may suspend in case the data structure does not contain the information required by it. The execution of the read and write primitives are independent in the sense that they can take place at different times. This marks an essential difference with synchronously communicating processes, like CSP [17], where reading from and writing to a channel has to take place at the same time.

In [5] a generic concurrent language $\mathcal{L}$ is introduced which assumes given a set of basic (or atomic) actions. Statements are constructed from these actions by means of sequential composition, the plus operator for non-deterministic choice, the parallel operator, and recursion. The basic actions are interpreted by means of an interpretation function I as partially defined *state* transformations. When the interpretation $I(a)$ of a pure read action a is undefined in a state this means that the execution of a is suspended. A suspended process is forced to wait until actions of other processes produce a state in which it is enabled. A pure write action a is characterized by the fact that $I(a)$ is a totally defined function. It can always proceed autonomously. In general, an action can embody both a read and a write component.

Many languages for asynchronously communicating processes can be obtained as instances of $\mathcal{L}$ by choosing the appropriate set of actions, the set of states and the interpretation function for the basic actions. For example, the imperative language described in [13], based on shared variables, can be modelled by taking as states functions from variables to values, as actions the set of assignments, and then the usual interpretation of an assignment as a state transformation. Languages based on the blackboard model [14], like Linda [15] and Shared Prolog [8] can be modelled analogously, by taking as states the configurations of a centralized data structure (the blackboard) and as actions checks and updates of the blackboard. Another example is the class of concurrent constraint languages [22]. These are modelled by interpreting the abstract set of states as a constraint system and the actions as ask/tell primitives. Concurrent logic languages, like Flat Concurrent Prolog [23], can be obtained by interpreting the states as the bindings established on the logical variables, and the actions as the unification steps. An asynchronous variant of CCS [20, 21] is modelled by considering the state as a set (or a multi-set) of actions. Performing an action then corresponds to adding it to the set, while performing the complementary action corresponds to testing whether the action is already in the set. Finally, a variant of CSP [17], based on asynchronous channels (see also [18]), can be obtained by taking as states the configurations of the channels and as actions the input-output primitives on these channels.

The basic computation model of $\mathcal{L}$ is described in [5] by means of a labelled transition system which is defined parametric with respect to the interpretation function I. It specifies for every statement what (initial) steps it can take. Each step results in a state transformation, which is registered in the label: as labels we use pairs of states. Based on this transition system, we define the initial/final state semantics of statements which distinguishes between the results of successfully terminating and deadlocking computations.

In this paper we study the congruence relation induced by the initial/final state semantics at different levels of abstraction. Our first result is a denotational characterization of this congruence relation at the highest level of abstraction, that is, when abstracting from the actual interpretation function I. This denotational model is based on *reactive sequences* as introduced in [5]. It is also defined parametric with respect to the interpretation function. In this model a computation is described as a sequence of pairs of states, a so-called reactive sequence. A pair of states encodes the state transformation which occurred during a single (atomic) transition step. These sequences are not necessarily connected, i.e. the final state of a pair can be different from the initial state of the following pair.

These "gaps" represent the possible steps made by the environment. We show that this model based on reactive sequences is fully abstract with respect to the initial/final state semantics in the following *generic* sense: two processes have in *every* interpretation the same set of reactive sequences if and only if in *every* interpretation they are observationally congruent.

In general the standard full abstraction statement for a given interpretation I does not hold. For example, the interpretation I which models shared variable concurrency requires some further abstractions (notably, abstraction from *finite stuttering* and the *granularity of interleaving*). In general these abstractions concern the specific nature of the underlying computational model as described by the given interpretation.

Next, we show that two processes have in *every* interpretation the same set of reactive sequences if and only if they have the same *failure* semantics. This failure semantics as described in [3] is defined directly in terms of the uninterpreted actions of $\mathcal{L}$. As such we have obtained a correspondence between the *state-based* interpretation of the language $\mathcal{L}$ and its *action-based* interpretation as studied in process algebras like PA or ACP [1]. That is, the intersection of the congruence relations induced by the initial/final state semantics of any instantiation of $\mathcal{L}$ coincides with the failure semantics of $\mathcal{L}$. It is worthwhile to observe here that the (action-based) failure semantics of $\mathcal{L}$ does *not* coincide with the *maximal traces* (of actions) of $\mathcal{L}$ because $\mathcal{L}$ does not include the *encapsulation* operator of ACP which allows one to hide actions. As such the maximal traces semantics of $\mathcal{L}$ already constitutes a congruence relation. However this action-based semantics of $\mathcal{L}$ does not capture the deadlock behavior as described by the initial/final state semantics of $\mathcal{L}$. This deadlock behavior instead is captured by the failure semantics. Consequently the standard failure axioms of [3] provide a general complete axiomatization of asynchronous communication, that is, these axioms characterize what holds in *any* parallel programming language based on an asynchronous communication mechanism.

We proceed our investigation by considering the specific nature of the basic actions of certain classes of instantiations of $\mathcal{L}$. First we consider instantiations which are based on a monoid structure of the basic actions. This monoid describes the internal structure of actions in terms of a state-based composition operation. That is, actions as state-transformations now can be viewed as being composed of other state-transformations. In this view, the 'silent step' τ of process algebras like ACP, denotes the (total) identity transformation, and the 'inaction' δ denotes the transformation which is undefined in every state. Our second main result is that a generic fully abstract state-based model for languages based on the (functional) composition of actions requires abstraction from *finite stuttering* and the *granularity of interleaving.* A complete and generic algebraic characterization is given of these abstractions.

We conclude with an axiomatization of read and write actions. In general a read action is a partially defined identity function (on the set of states) whereas a write action is defined in every state, that is, it never blocks. Moreover we further refine this axiomatization to the class of so-called monotonic languages. These languages are based on a monotonic interpretation of the actions with respect to a given information-ordering on states. Characteristic instances of this class are the concurrent logic (constraint) languages.

1.1 Comparison with related work

Most existing works on process algebras of asynchronous communication concern particular programming languages [4, 6, 7, 10–12, 18]. One of the main contributions of this paper is to provide a general framework for studying algebraic properties of *classes* of languages, instead of one particular language. These classes represent different asynchronous communication mechanisms which are defined in terms of some general semantic characteristics of the basic actions.

Our approach allows for interesting generalizations of various existing semantic models and algebras. For example, in [9] a fully abstract model is given for a concurrent imperative language based on shared-variables. We generalize this model in two dimensions. First of all we give a general semantic treatment of deadlock in languages based on asynchronous communication (in [9] only the results of successfully terminating computations are considered). Moreover, we show that the abstraction from finite stuttering and granularity of interleaving defined in [9] and [5] apply to all instantiations of $\mathcal{L}$ which are based on a monoid structure of actions.

Other generalizations concern the axiomatization of read and write actions and the axiomatization of certain commutativity properties of actions as described in [7, 4, 6]. We axiomatize these properties in terms of a general class of monotonic interpretations.

2 The language $\mathcal{L}$ and its operational semantics

In this section we introduce the language $\mathcal{L}$ and define its operational semantics by means of a transition system.

Definition 1. *Let $(a \in)A$ be an infinite but countable set of actions and let $(x \in) Var$ be a set of (statement) variables. We define the set $(s \in)\mathcal{L}$ of statements as follows:*

$$s ::= \ a \mid x \mid s;t \mid s+t \mid s \parallel t \mid \mu x.s$$

Moreover, $\mathcal{L}$ contains a special E, the terminated statement.

The symbols ';', '+' and '$\parallel$' represent the sequential, the choice and the parallel operator respectively. In the statement $\mu x.s$, the free ocurrences of x in s are interpreted recursively as the statement $\mu x.s$ itself. By $s[\mu x.s/x]$ we denote the statement obtained by substituting $\mu x.s$ for each free occurrence of x in s. In the following we consider only closed statements (without free variables).

Definition 2. *An* interpretation *I consists of a set $(\sigma \in)\Sigma$ of states and an assignment of a state-transformation $I(a) \in \Sigma \rightarrow \Sigma_\perp$ to each action a, where $\perp \notin \Sigma$ and $\Sigma_\perp = \Sigma \cup \{\perp\}$.*

Actions are thus interpreted as partially defined functions on the set of states: $I(a)(\sigma) = \perp$ indicates that $I(a)$ is not defined in σ.

Given an interpretation I the computational model of $\mathcal{L}$ is described by a *labelled transition-system* $(\mathcal{L}, \mathit{Label}, \rightarrow)$. The set $(\lambda \in)\mathit{Label}$ of labels is defined by $\mathit{Label} = \Sigma \times \Sigma$. A label represents the state tranformation caused by the action performed during the transition step.

Definition 3. *The transition relation* $\rightarrow \subseteq \mathcal{L} \times \mathit{Label} \times \mathcal{L}$ *is defined as the smallest relation satisfying the following axiom and rules:*

- *For* $\sigma' = I(a)(\sigma)$ *we have* $a \xrightarrow{\langle\sigma,\sigma'\rangle} E$.
- *If* $s \xrightarrow{\lambda} s'$ *then*

$$s;t \xrightarrow{\lambda} s';t \quad s+t \xrightarrow{\lambda} s' \quad t+s \xrightarrow{\lambda} s' \quad s \parallel t \xrightarrow{\lambda} s' \parallel t \quad t \parallel s \xrightarrow{\lambda} t \parallel s'$$

  If $s' = E$ *then read* t *for* $s';t$, $s' \parallel t$ *and* $t \parallel s'$ *in the clauses above.*
- *If* $s[\mu x.s/x] \xrightarrow{\lambda} s'$ *then* $\mu x.s \xrightarrow{\lambda} s'$.

For $s \neq E$ *we introduce* $s \xrightarrow{\langle\sigma,\Delta\rangle} E$ *to indicate that there does not exist a state* σ' *and a statement* s' *such that* $s \xrightarrow{\langle\sigma,\sigma'\rangle} s'$.

Note that the transition system depends on a given I (this dependence we leave implicit). On the basis of the transition system we introduce the following notion of *observables.*

Definition 4. *Let* $\Sigma^+ = \Sigma \times \{\Delta, E\}$. *Given an interpretation* I, *we define* $\mathcal{I}(s) \in \Sigma \rightarrow \mathcal{P}(\Sigma^+)$ *recursively by*

$$\mathcal{I}(s)(\sigma) = \bigcup\{\mathcal{I}(s')(\sigma') \mid s \xrightarrow{\langle\sigma,\sigma'\rangle} s'\} \cup \{\langle\sigma,\Delta\rangle \mid s \xrightarrow{\langle\sigma,\Delta\rangle} E\}$$

We define $\mathcal{I}(E)(\sigma) = \{\langle\sigma, E\rangle\}$, *for every* $\sigma \in \Sigma$. *This recursive definition of* $\mathcal{I} : \mathcal{L} \rightarrow (\Sigma \rightarrow \mathcal{P}(\Sigma \times \{\Delta, E\}))$ *can be justified formally as the least fixed point (defined with respect pointwise extended set-inclusion) of a corresponding operator.*

The observables $\mathcal{I}(s)$ records for each input state σ both the final results of successfully terminating and deadlocking computations of s in σ. Successfull termination is denoted by E and deadlock is indicated by Δ.

3 A generic process algebra

In this section we introduce a generic process algebra for asynchronous communication. Its genericity lies in the fact that we abstract from the specific nature of the asynchronous communication mechanism. This abstraction is captured by the following congruence relation.

Definition 5. *We define $s \cong t$ if and only if for every interpretation I, $I(c[s]) = I(c[t])$, for every context $c[x]$. A context $c[x]$ is simply a statement with a free statement-variable x. The result of replacing every occurrence of x in $c[x]$ by a statement s is denoted by $c[s]$.*

The relation $s \cong t$ thus describes the identities between statements which hold in every interpretation; this is reflected by the presence of universal quantification on both contexts and interpretations. The following definition (which already appeared in [5]) gives a denotational characterization of this relation which assigns to each statement a set of *reactive sequences.*

Definition 6. *Let I be an interpretation. Let $D = (\Sigma \times \Sigma_\perp)^*$, with typical element ω. Elements of D are also called reactive sequences. For $R \subseteq D$ and label λ we define $\lambda \cdot R = \{\lambda \cdot \omega \mid \omega \in R\}$ (here $\cdot$ denotes the prefix operation). We define $\tilde{I}(s) \subseteq D$ recursively by*

$$\tilde{I}(s) = \bigcup \{\lambda \cdot \tilde{I}(t) \mid s \xrightarrow{\lambda} t\} \cup \{\langle \sigma, \Delta \rangle \mid s \xrightarrow{\langle \sigma, \Delta \rangle} E\}$$

We define $\tilde{I}(E)(s) = \{\epsilon\}$ (the empty sequence is denoted by ϵ). The formal justification of this recursive definition of $\tilde{I}$ can be given as the least fixed point (defined with respect pointwise extended set-inclusion) of a corresponding operator.

The proof of the following theorem can be found in [5].

Theorem 1. *The relation $s \approx t$ defined by $\tilde{I}(s) = \tilde{I}(t)$, for all I, is a congruence relation.*

We are now able to state the following *generic* full abstractness result.

Theorem 2. *For any statement s and t, $s \cong t$* if and only if *$s \approx t$.*

Proof The *if* part directly follows from the compositionality result of Theorem 1 and by the fact that for every s (distinct from E) and σ the set $I(s)(\sigma)$ can be obtained from $\tilde{I}(s)$ in the following way

$$\begin{aligned} I(s)(\sigma_1) = \{\langle \sigma_{n+1}, E \rangle \mid \langle \sigma_1, \sigma_2 \rangle \langle \sigma_2, \sigma_3 \rangle \ldots \langle \sigma_n, \sigma_{n+1} \rangle \in \tilde{I}(s)\} \cup \\ \{\langle \sigma_n, \Delta \rangle \mid \langle \sigma_1, \sigma_2 \rangle \langle \sigma_2, \sigma_3 \rangle \ldots \langle \sigma_n, \Delta \rangle \in \tilde{I}(s)\} \end{aligned}$$

Note that we consider sequences of connected pairs of states, namely, the second state of a pair is the first one of the next pair. Such sequences represent computations of s viewed as a *closed* system.

In order to prove the *only if* part we proceed by contraposition. We assume $s \not\approx t$; this ensures the existence of an interpretation I such that $\tilde{I}(s) \neq \tilde{I}(t)$. Without loss of generality we may assume the existence of a reactive sequence ω such that $\omega \in \tilde{I}(s) \setminus \tilde{I}(t)$. There are two cases, $\omega = \langle \sigma_1, \sigma_1' \rangle \langle \sigma_2, \sigma_2' \rangle \ldots \langle \sigma_n, \sigma_n' \rangle$ or $\omega = \langle \sigma_1, \sigma_1' \rangle \langle \sigma_2, \sigma_2' \rangle \ldots \langle \sigma_n, \Delta \rangle$. We present only the proof for the first case; the other one is treated similarly.

Let Σ be the set of states of I. We define a new interpretation I' with the state space $(w \in)\Sigma^*$ consisting of all (finite) sequences of states in Σ and the following interpretation of the basic actions:

$$I'(a) \;=\; \lambda w.\text{if } w \neq \epsilon \text{ then } w \cdot I(a)(last(w)) \text{ else } \perp$$

where $\cdot$ denotes the append operation, as a special case we define $w \cdot \perp = \perp$; the last element of the non-empty sequence w is denoted by $last(w)$. In this way we have encoded the history of the computation in the actual state.

We now consider a statement $r = a_1; \ldots; a_{n-1}$ such that each action a_i is fresh in the sense that it does not occur in s or in t. Such an action a_i is used to fill the gap between the states $\sigma_1 \ldots \sigma_i'$ and $\sigma_1 \ldots \sigma_i' \sigma_{i+1}$:

$$I'(a_i) = \lambda\omega.\text{if } \omega = \sigma_1 \ldots \sigma_i' \text{ then } \omega \cdot \sigma_{i+1} \text{ else } \perp$$

It is now easy to see that $I'(s \parallel r)(\sigma_1) \neq I'(t \parallel r)(\sigma_1)$ (hence also $s \not\approx t$).
In fact $\langle \sigma_1 \ldots \sigma_{n-1}' \sigma_n \sigma_n', E \rangle \in I(s \parallel r)(\sigma_1) \setminus I(t \parallel r)(\sigma_1)$, otherwise we would have $\omega \in \tilde{I}(t)$ which contradicts the above assumption. □

However, given an interpretation I, in general the *usual* full abstraction statement

$$\tilde{I}(s) = \tilde{I}(t) \;\text{ if and only if for every context } c[\,],\, I(c[s]) = I(c[t])$$

does not hold. For example, the interpretation I which models shared variable concurrency requires some further abstractions (notably, abstraction from *finite stuttering* and the *granularity of interleaving*). These abstractions in general however concern the specific nature of the underlying computational model as described by the given interpretation.

We now show that the relation $\approx$ coincides with the *failure* semantics of the (uninterpreted) language $\mathcal{L}$.

Definition 7. *Consider the usual labelled transition system $s \xrightarrow{a} t$ indicating that the statement s can reach t performing an action a (e.g. $a; s \xrightarrow{a} s$). With $s \xrightarrow{w} t$ we mean that t can be reached after having performed the (finite) sequence of actions w (i.e. if w is the empty sequence then $s = t$, otherwise, if w is the sequence $a_1 \ldots a_n$ then $s \xrightarrow{a_1} \ldots \xrightarrow{a_n} t$). Let $Init(s)$ be the set of labels a such that $s \xrightarrow{a} t$ for some t. We define*

$$\begin{aligned} F(s) = \{&(w, E) \mid s \xrightarrow{w} E\} \cup \\ \{&(w, \mathcal{F}) \mid s \xrightarrow{w} t,\ t \neq E, \text{ and } \mathcal{F} \subseteq A \text{ s.t. } \mathcal{F} \cap Init(t) = \emptyset\} \end{aligned}$$

Given a statement s, $F(s)$ is the set of the pairs (w, E), where the sequence of actions w (performed by s) leads to the termination statement E, and $(w, \mathcal{F})$, where the sequence of actions w (performed by s) leads to a statement different from E that is not able to perform the actions in $\mathcal{F}$.

Theorem 3. *For any statement s and t, $s \approx t$ if and only if $F(s) = F(t)$.*

Proof The *if* part follows from the fact that for every s, $\tilde{I}(s)$ can be obtained from the set $F(s)$ (for example, a deadlock state $\langle\sigma, \Delta\rangle$ is derived from a failure set $\mathcal{F}$ by the condition that $I(a)(\sigma) = \perp$, for $a \notin \mathcal{F}$).

In order to prove the *only if* part we proceed by contraposition. Assume that $F(s) \neq F(t)$; without loss of generality we may assume the existence of a pair $(w, \mathcal{F})$ (or (w, E)) contained in $F(s) \setminus F(t)$. We present only the case of $(w, \mathcal{F})$, as the other one is treated similarly.

Consider the interpretation I whose actions are defined on (finite) sequences $A_1 \dots A_n$ of *sets* of actions (i.e. $A_i \subseteq A$) in the following manner:

$$I(a) = \lambda\, A_1 \dots A_n.\text{if } a \in A_1 \text{ then } A_2 \dots A_n \text{ else } \perp.$$

Let $w = a_1 \dots a_n$; we have that $I(s)(\{a_1\} \dots \{a_n\}\mathcal{F}) \neq I(t)(\{a_1\} \dots \{a_n\}\mathcal{F})$. In fact $\langle \mathcal{F}, \Delta\rangle \in I(s)(\{a_1\} \dots \{a_n\}\mathcal{F}) \setminus I(t)(\{a_1\} \dots \{a_n\}\mathcal{F})$, otherwise we have $(w, \mathcal{F}) \in F(t)$ which contradicts the above assumption. This ensures that $s \not\cong t$, and so we have $s \not\approx t$ by Theorem 2. □

By Theorem 3 the well-known algebraic laws which characterize the failure equivalence for PA processes (that is, ACP processes without the synchronous communication function, see [1]) in Table 1 give a complete axiomatization of $s \approx t$ for finite statements.

A1	$s + t \;=\; t + s$
A2	$(s + t) + u \;=\; s + (t + u)$
A3	$s + s \;=\; s$
A4	$(s + t); u \;=\; s; u + t; u$
A5	$(s; t); u \;=\; s; (t; u)$
M1	$s \parallel t \;=\; s \mathbin{\|\!\lfloor} t + t \mathbin{\|\!\lfloor} s$
M2	$a \mathbin{\|\!\lfloor} s \;=\; a; s$
M3	$a; s \mathbin{\|\!\lfloor} t \;=\; a; (s \parallel t)$
M4	$(s + t) \mathbin{\|\!\lfloor} u \;=\; s \mathbin{\|\!\lfloor} u + t \mathbin{\|\!\lfloor} u$
F1	$a; (b; s + t) + a; (b; s' + t') \;=\; a; (b; s + b; s' + t) + a; (b; s + b; s' + t')$
F2	$a; (b + t) + a; (b; s' + t') \;=\; a; (b + b; s' + t) + a; (b + b; s' + t')$
F3	$a; s + a; (t + u) \;=\; a; s + a; (s + u) + a; (t + u)$

Table 1. Axioms for failure equivalence.

The failure axioms thus provide a general characterization of asynchronous communication when abstracting from the particular nature of the actions.

4 The monoid of actions

As the set of actions we now consider not just a generic set A but a monoid $(A, \circ)$, with neutral element τ and zero element δ (i.e. $\tau \circ a = a \circ \tau = a$ and $\delta \circ a = a \circ \delta = \delta$). The operation $\circ$ is interpreted as functional composition of state-transformations. We restrict our analysis to the class of interpretation functions which are *monoidal*.

Definition 8. *A monoidal interpretation I consists of a set $(\sigma \in)\Sigma$ of states and an assignment of a state-transformation $I(a) \in \Sigma \to \Sigma_\perp$ to each action a, which satisfies the monoid structure:*

- $I(\tau) = \lambda\sigma.\ \sigma$,
- $I(\delta) = \lambda\sigma.\ \perp$,
- $I(a \circ b) = \lambda\sigma.\ \textit{if } I(a)(\sigma) \neq \perp \textit{ then } I(b)(I(a)(\sigma)) \textit{ else } \perp.$

The action τ denotes the identity function and δ denotes the nowhere defined function. The composite action $a \circ b$ is interpreted as the composition of the interpretations of a and b.

Given the additional monoid structure on actions the congruence $s \approx t$ no longer coincides with $s \cong t$. For example we have that $a \cong a + a;\tau$ (in fact, $a \cong a;\tau$) and $a;b \cong a;b + a \circ b$ but we do not have $a \approx a;\tau$ or $a;b \approx a;b + a \circ b$: For every monoidal interpretation I we have

$$\tilde{I}(a) = \{\langle\sigma, \sigma'\rangle \mid I(a)(\sigma) = \sigma'\} \cup \{\langle\sigma, \Delta\rangle \mid I(a)(\sigma) = \perp\}$$

whereas

$$\tilde{I}(a;\tau) = \{\langle\sigma, \sigma'\rangle\langle\sigma'', \sigma''\rangle \mid I(a)(\sigma) = \sigma'\} \cup \{\langle\sigma, \Delta\rangle \mid I(a)(\sigma) = \perp\}.$$

Moreover,

$$\begin{array}{l}\tilde{I}(a;b) = \\ \{\langle\sigma_1, \sigma_2\rangle\langle\sigma_3, \sigma_4\rangle \mid I(a)(\sigma_1) = \sigma_2,\ I(b)(\sigma_3) = \sigma_4\} \cup \\ \{\langle\sigma_1, \sigma_2\rangle\langle\sigma_3, \Delta\rangle \mid I(a)(\sigma_1) = \sigma_2,\ I(b)(\sigma_3) = \perp\} \cup \\ \{\langle\sigma, \Delta\rangle \mid I(a)(\sigma) = \perp\}\end{array}$$

whereas

$$\tilde{I}(a;b + a \circ b) = \tilde{I}(a;b) \cup \tilde{I}(a \circ b).$$

Therefore we introduce the following abstraction operation.

Definition 9. *Let $\alpha(R)$, for $R \subseteq D$, be the smallest set containing R which is closed under the following conditions:*

$$\begin{array}{lll}\textbf{C1}\ w_1 w_2 \in R & \Rightarrow w_1\langle\sigma, \sigma\rangle w_2 \in R & (w_1 \neq \epsilon) \\ \textbf{C2}\ w \in R & \Rightarrow \langle\sigma, \sigma\rangle w \in R & (\langle\sigma, \Delta\rangle \notin R) \\ \textbf{C3}\ w_1\langle\sigma, \sigma'\rangle\langle\sigma'\sigma''\rangle w_2 \in R & \Rightarrow w_1\langle\sigma, \sigma''\rangle w_2 \in R & \end{array}$$

The first two closure conditions introduce *finite stuttering*. The third condition 'internalizes' interleaving points. The restrictions on the conditions **C1** and **C2** ensure correctness (without these restrictions we would identify, for example, δ and $\tau;\delta$).

Definition 10. *We introduce the semantics $\tilde{I}_\alpha$ defined by $\tilde{I}_\alpha(s) = \alpha(\tilde{I}(s))$. Moreover, we define $s \approx_\alpha t$ if and only if $\tilde{I}_\alpha(s) = \tilde{I}_\alpha(t)$, for every I.*

We have the following generic full abstraction result for monoidal interpretations.

Theorem 4. *For any statement s and t, $s \approx_\alpha t$ if and only if $s \cong t$.*

Proof The implication $s \approx_\alpha t \Rightarrow s \cong t$ follows from the compostionality of I_α and its correctness with respect to the initial/final state semantics.

With respect to the proof of the implication $s \not\approx_\alpha t \Rightarrow s \not\cong t$ we first observe that the construction of a distinguishing context in the proof of Theorem 2 which is based on sequences of states (of the state-space of the given interpretation) does not apply here because the resulting interpretation does not satisfy the monoid structure: the interpretation of τ does not correspond with the identity function and the interpretation of a composed action $a \circ b$ does not correspond with the (functional) composition of the interpretations of a and b.

Instead we define now the distinguishing context in terms of the state-space of the interpretation I for which $\tilde{I}_\alpha(s) \neq \tilde{I}_\alpha(t)$ holds: Let $\omega = \langle \sigma_1, \sigma'_1 \rangle \ldots \langle \sigma_n, \sigma'_n \rangle \in \tilde{I}_\alpha(s) \setminus \tilde{I}_\alpha(t)$ (the case of ω ending in deadlock is treated similarly). We define a new interpretation J with the *same* state space Σ as that of the given interpretation I such that for every action a occurring in s or t, $J(a) = I(a)$. Furthermore let a_i, $1 \leq i < n$, be some new actions not occurring in s or t. Without loss of generality we may assume that these actions are *atomic* in the following sense: If $a = b \circ c$ then either $a = b$ and $c = \tau$ or $a = c$ and $b = \tau$. We define $J(a_i) = \{\langle \sigma'_i, \sigma_{i+1} \rangle\}$ (so a_i transforms σ'_i into σ_{i+1} and blocks in every other state). Let $r = a_1; \ldots; a_{n-1}$. It follows that $\sigma'_n \in J(s \parallel r)(\sigma_1)$. Moreover, for any statement u, with $a_1, \ldots, a_{n-1}$ not occurring in u, we have that $\sigma'_n \in J(u \parallel r)(\sigma_1)$ implies $\omega \in \tilde{J}_\alpha(u)$. We sketch the basic reasoning pattern underlying this implication: Let ω' be a succesfully terminating computation (more precisely, a *connected* reactive sequence) of $u \parallel r$ starting in σ_1 and resulting in σ'_n. It follows that for every action a_i there exists a computation step $\langle \sigma'_i, \sigma_{i+1} \rangle$ in ω' such that the execution of u has transformed σ_i by some finite number of computation steps into σ'_i. By the closure condition **C3** these consecutive computation steps of u are represented in $\tilde{J}_\alpha(u)$ also by the one computation step $\langle \sigma_i, \sigma'_{i+1} \rangle$. In case $\sigma_i = \sigma'_{i+1}$ we can also apply the closure condition **C1** or **C2** (it is not difficult to see that $\sigma'_n \in J(u \parallel r)(\sigma_1)$ implies $\langle \sigma_1, \Delta \rangle \notin \tilde{J}_\alpha(u)$). $\square$

Also here we have that the usual full abstractness result for a given $\tilde{I}_\alpha$ does not hold. For example, the monoidal interpretation based on concurrent constraint programming requires some further abstractions which stem from the monotonicity of the basic actions (see [6] and below).

With the introduction of the monoid structure on actions the failure equivalence defined above must be changed in order to cope with the fixed interpretation of the τ and δ actions and the composition operator $\circ$.

Definition 11. *Let a be an action different from τ and δ: by $s \xrightarrow{a}_{\alpha} t$ we mean that there exist $a_1 \dots a_n$ and $s_1 \dots s_{n-1}$ such that $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} t$ and $a = a_1 \circ a_2 \circ \dots \circ a_n$. Let w be a sequence of actions different from τ and δ; by $s \xrightarrow{w}_{\alpha} t$ we mean the following: if w is the empty sequence then t can be reached from s via a non-empty sequence of τ actions, otherwise, if w is the sequence $a_1 \dots a_n$ then $s \xrightarrow{a_1}_{\alpha} \dots \xrightarrow{a_n}_{\alpha} t$. We define*

$$
\begin{aligned}
F_{\alpha}(s) = \{(w, E) \mid s \xrightarrow{w}_{\alpha} E\} \cup \\
\{(w, \mathcal{F}) \mid s \xrightarrow{w}_{\alpha} t,\ \tau \notin \mathit{Init}(t), \text{ and } \mathcal{F} \subseteq A \text{ s.t. } \mathcal{F} \cap \mathit{Init}(t) = \emptyset\}
\end{aligned}
$$

where Init has been defined in Definition 3.5 on the initial transition system $\xrightarrow{a}$.

In the definition of F_{α} we consider the condition $\tau \notin \mathit{Init}(t)$, where $\mathit{Init}(t)$ denotes the set of initial actions of t according to the original transition relation $t \xrightarrow{a} t'$ $(t \neq E)$, because only failures of *stable* statements (as they are called in [20, 21]) must be taken into account. A statement s which is able to perform an initial τ action is not stable, in the sense that it is always able to perform this kind of action whatever is the actual state of the computation.

Theorem 5. *For any statement s and t, $s \approx_{\alpha} t$ if and only if $F_{\alpha}(s) = F_{\alpha}(t)$.*

Proof We have to resort to a different proof than that of Theorem 3 because the interpretation defined in that proof does *not* satisfy the monoidal structure.

Instead, we show in the full paper that for every two statements s and t there exist statements s' and t' such that $I_{\alpha}(s) = I_{\alpha}(t)$ if and only if $I(s') = I(t')$, for every monoidal interpretation I, and $F_{\alpha}(s) = F_{\alpha}(t)$ if and only if $F(s') = F(t')$. And so we can apply Theorem 3. In fact, the statements s' and t' can be obtained by the rewrite rules $s; \tau \;\Rightarrow\; s$ and $a; (b; s + t) \;\Rightarrow\; a; (b; s + t) + a \circ b; s$ (or $a; (b + t) \;\Rightarrow\; a; (b + t) + a \circ b$). □

In order to give a complete axiomatization (for finite statements) of this new failure semantics we have to add to the axioms in Table 1 new axioms dealing with the special δ and τ actions, and the composition operator $\circ$. In Table 2 we present the deadlock laws D for the δ action, the axioms T for τ, and the contraction laws C for the composition operator $\circ$.

Theorem 6. *For any statement s and t,*

$$s \approx_{\alpha} t \text{ if and only if } \mathrm{A} + \mathrm{M} + \mathrm{F} + \mathrm{D} + \mathrm{T} + \mathrm{C} \vdash s = t$$

It is of interest to observe that we do not consider the standard τ-law $s = s; \tau$ but a weaker version $s = s + s; \tau$. In fact, we will see in the following that the standard law is now derivable by its weaker version due to the presence of the contraction laws. Moreover, we introduced this kind of axiomatization because it

D1	$s + \delta \; = \; s$
D2	$\delta ; s \; = \; \delta$
T1	$s \; = \; s + s ; \tau$
T2	$a ; (\tau ; s + t) \; = \; a ; (\tau ; s + t) + a ; \tau ; s$
T3	$a ; (\tau + t) \; = \; a ; (\tau + t) + a ; \tau$
C1	$a ; (b ; s + t) \; = \; a ; (b ; s + t) + a \circ b ; s$
C2	$a ; (b + t) \; = \; a ; (b + t) + a \circ b$

Table 2. The Laws for δ, τ and $\circ$.

is possible to prove that there is a strong connection between, on the one hand, abstraction of granularity and the contraction laws C1 and C2, and, on the other hand, finite stuttering and the τ-laws T1, T2, and T3.

Theorem 7. *Let α_1 denote the abstraction operation corresponding to the closure conditions* **C1** *and* **C2**. *We define $\tilde{I}_{\alpha_1}(s) \subseteq D$ by $\alpha_1(\tilde{I}(s))$. Here $\alpha_1(R)$, for $R \subseteq D$, is defined as the smallest set containing R which is closed under the closure conditions* **C1** *and* **C2**. *We have*

$$s \approx_{\alpha_1} t \text{ if and only if } \mathrm{A} + \mathrm{M} + \mathrm{F} + \mathrm{D} + \mathrm{T} \vdash s = t$$

where $s \approx_{\alpha_1} t$ if and only if $\tilde{I}_{\alpha_1}(s) = \tilde{I}_{\alpha_1}(t)$, for every I.

Let α_2 denote the abstraction operation corresponding to the closure condition **C3**. *We define $\tilde{I}_{\alpha_2}(s) \subseteq D$ by $\alpha_2(\tilde{I}(s))$. Here $\alpha_2(R)$, for $R \subseteq D$ is defined as the smallest set containing R which is closed under the closure condition* **C3**. *We have*

$$s \approx_{\alpha_2} t \text{ if and only if } \mathrm{A} + \mathrm{M} + \mathrm{F} + \mathrm{D} + \mathrm{C} \vdash s = t$$

where $s \approx_{\alpha_2} t$ if and only if $\tilde{I}_{\alpha_2}(s) = \tilde{I}_{\alpha_2}(t)$, for every I.

An alternative axiomatization of the contraction laws consists of the introduction of the $\circ$ operator to statements.

Definition 12. *We have the following transition rules for $\circ$:*

- *If $s \xrightarrow{\lambda} s'$ then $s \circ t \xrightarrow{\lambda} s' \circ t$ where s' is assumed to be distinct from E.*
- *If $s \xrightarrow{\langle \sigma, \sigma' \rangle} E$ and $t \xrightarrow{\langle \sigma', \sigma'' \rangle} t'$ then $s \circ t \xrightarrow{\langle \sigma, \sigma'' \rangle} t'$*

The operator $\circ$ is associative and is distributive wrt $+$ and ; (see Table 3). The 'contraction' law can be now described by the axiom in Table 4.

B1	$(s \circ t) \circ u$	$= s \circ (t \circ u)$
B2	$s \circ (t; u)$	$= (s \circ t); u$
B3	$(s; t) \circ u$	$= s; (t \circ u)$
B4	$s \circ (t + u)$	$= (s \circ t) + (s \circ u)$
B5	$(s + t) \circ u$	$= (s \circ u) + (t \circ u)$

Table 3. Associativity and Distributivity of $\circ$.

B6	$s; t = s; t + s \circ t$

Table 4. The Contraction Law.

Theorem 8. *For any statement s and t,*

$$s \approx_\alpha t \text{ if and only if } \mathrm{A} + \mathrm{M} + \mathrm{F} + \mathrm{D} + \mathrm{T} + \mathrm{B} \vdash s = t$$

Under this alternative characterization it is clear that the standard τ-law $s = s; \tau$ is derivable. Indeed, we have that $s = s + s; \tau$ by $T1$ and $s; \tau = s; \tau + s \circ \tau$; but it is easy to prove that $s \circ \tau = s$ hence also $s; \tau = s; \tau + s$.

Of interest is that a similar operator $\circ$ has been introduced already in [2] for modeling mutual exclusion. The main difference with our axiomatization is that we use the operator of mutual exclusion to model the abstraction from internal interleaving points. This abstraction is captured by the Contraction Law.

5 Read and write actions

In this section we briefly discuss a generic process algebra for read and write actions (in the context of a monoid of actions). We assume a partitioning of the set of actions into the sets *write* and *read* of write and read actions. Write actions are autonomous actions which never block, i.e. for every interpretation I and write action a we have that $I(a)(\sigma) \neq \perp$. For $a \in$ *read* we require that $I(a)(\sigma) = \sigma$ if $I(a)(\sigma)$ is defined. Note that thus τ and δ are special kind of read actions. Table 5 gives an algebraic characterization of the read and write actions. The axioms W1 and W2 describe the autonomous nature of the write action with respect to the plus operator. The axiom R captures the interplay between write and read actions. Note that we do *not* have $a = b; a$ because the read action b in general introduces deadlock. For example, the initial/final state semantics of $a + b$ and $b; a + b$ in general differ: Let I be an interpretation such that $I(b)(\sigma)$ is undefined. The execution of $b; a + b$ in σ then will deadlock whereas $a + b$ will result in $I(a)(\sigma)$.

Next we consider monotonic read and write actions.

W1	$a;(b;s+t)$	$= a;(b;s+t)+a;b;s$	$b \in \mathit{write}$
W2	$a;(b+t)$	$= a;(b+t)+a;b$	$b \in \mathit{write}$
R	a	$= a+b;a$	$a \in \mathit{write},\ b \in \mathit{read}$

Table 5. Read and write actions.

Definition 13. *Given an* information *ordering* $\sqsubseteq$ *on the set of states a monotonic interpretation assigns to each action a monotonic function.*

Characteristic instances of this class are the concurrent constraint languages. Table 6 gives an algebraic characterization of these monotonic interpretations. The axioms MR1 and MR2 describe a characteristic commutativity property of the read actions in monotonic interpretations.

MR1	$a;(b;s+t)$	$= a;(b;s+t)+b;a;s$	$b \in \mathit{read}$
MR2	$a;(b+t)$	$= a;(b+t)+b;a$	$b \in \mathit{read}$

Table 6. Monotonic interpretations.

We refer to the full paper for the completeness proof of the above axiomatizations.

6 Conclusion and future work

We have introduced a general framework $\mathcal{L}$ for the study of process algebras of concurrent languages based on asynchronous communication. We have argued that in this framework the failure semantics as developed for action-based process algebras like ACP coincides with the initial/final state semantics of $\mathcal{L}$. Furthermore we have given a generic process algebra for instantiations of $\mathcal{L}$ which are based on a state-based composition operation of the actions. Finally, we concluded with a generic characterization of read and write actions.

Currently we are investigating instantiations of $\mathcal{L}$ which involve an information ordering on the set of states. An interesting class of such instantiations are those which interpret the actions as *closure* operators (concurrent constraint programming is such an instantiation), i.e.

- $I(a)$ is extensive ($\sigma \sqsubseteq I(a)(\sigma)$)
- $I(a)$ is idempotent ($I(a)(I(a)(\sigma)) = I(a)(\sigma)$).

Of interest also is a generic process algebra of so-called *get* actions which remove information, i.e. $I(a)(\sigma) \sqsubseteq \sigma$.

Another line of research involves the introduction of a general state-based *hiding* operator in the language $\mathcal{L}$. To this end we envisage the introduction of instantiations of $\mathcal{L}$ which are based on *cylindric algebras* [16]. The general notion of existential quantification as given by these algebras then will provide a basis for the introduction of a corresponding programming construct in the language $\mathcal{L}$.

References

1. J.C.M. Baeten and W.P. Weijland. Process algebra. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press 1990.
2. J.A. Bergstra and J.W. Klop. Process algebra for communication and mutual exclusion. Technical report CS-R8409, CWI, 1984.
3. J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Readies and failures in the algebra of communicating systems. *SIAM J. Comp.*, 17(6):1134–1177, 1988.
4. F.S. de Boer, J.W. Klop and C. Palamidessi. Asynchronous communication in process algebra. Proc. of the seventh annual IEEE symposium on Logics in Computer Science (LICS), pages 137–147. IEEE Computer Society Press, Los Alamitos, California, 1992.
5. 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. Proc. of Concur '91, Lecture Notes in Computer Science, Vol. 527, pages 111-126, Amsterdam, The Netherlands, 1991.
6. F.S. de Boer and C. Palamidessi. A process algebra of concurrent constraint programming. Proc. of the Joint International Conference and Symposium on Logic Programming, JICSLP'92, pages 463–477, The MIT press, 1992.
7. M. Boreale, R. De Nicola, R. Pugliese. Asynchronous observations of processes. Proc. of FoSSaCS'98, Lecture Notes in Computer Science, Vol. 1378, pages 95–109, 1998.
8. A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. ACM Transaction on Programming Languages and Systems, 13(1):99–123, 1991.
9. S. D. Brookes. Full abstraction for a shared-variable parallel language. Information and Computation, 127(2):145-163, 15 June 1996.
10. N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of Linda coordination primitives. Theoretical Computer Science, 192(2): 167–199, 1998.
11. P. Ciancarini, R. Gorrieri, and G. Zavattaro. An alternative semantics for the parallel operator of the calculus of gamma programs. Coordination Programming: Mechanism, Models and Semantics, pages 232–248, Imperial College Press, 1996.
12. R. De Nicola and R. Pugliese. A process algebra based on Linda. Proc. of Coordination'96, Lecture Notes of Computer Science, Vol. 1061, pages 160–178, 1996.
13. E.W. Dijkstra. Co-operating sequential processes. In Academic Press, New York, 1968.
14. D. Erman, F. HayesRoth, V. Lesser, and D. Reddy. The Hearsay2 speech understanding system: Integrating knowledge to resolve uncertainty. ACM Computing Surveys, 12:213–253, 1980.
15. D. Gelenter. Generative communication in Linda. ACM TOPLAS, 7(1):80–112, 1986.

16. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland, 1971.
17. C.A.R. Hoare. Communicating sequential processes. Communications of the ACM, 21(8):666–677, 1978.
18. He Jifeng, M.B. Josephs, and C.A.R. Hoare. A theory of synchrony and asynchrony. Proc. of IFIP Working Conference on Programming Concepts and Methods, pages 459–478, 1990.
19. G. Kahn. The semantics of a simple language for parallel programming. Information Processing 74: Proc. of IFIP Congress, pages 471–475, New York, 1974. North-Holland.
20. R. Milner. A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol. 92, New York, 1980.
21. R. Milner. Calculi for synchrony and asynchrony. Theoretical Computer Science, 25:267–310, 1983.
22. V.A. Saraswat. Concurrent Constraint Programming Languages. PhD thesis, january 1989. The MIT Press.
23. E.Y. Shapiro. The family of concurrent logic programming languages. ACM Computing Surveys, 21(3):412–510, 1989.