The expressive power of parallelism *

Joachim Parrow
Swedish Institute of Computer Science, Box 1263, S-164 28
Kista, Sweden

We explore an algebraic language for networks consisting of a fixed number of reactive units, communicating synchronously over a fixed linking structure. The language has only two operators: disjoint parallelism, where two networks are composed in parallel without any interconnections, and linking, where an interconnection is formed between two ports. The intention is that these operators correspond to the primitive steps when constructing networks, and that they therefore are conceptually simpler than the operators in existing process algebras. We investigate the expressive power of our language. The results are: (1) Definability of behaviours: with only three simple processing units, every finite-state behaviour can be constructed. (2) Definability of operators: we characterise the network operators which are definable within the language; these turn out to include most operators previously suggested for describing parallelism. Our results hold for any congruence between trace equivalence and observation equivalence.

1. Introduction

In this paper we will investigate an algebraic language for networks of synchronously communicating units. Several such algebras have been developed in recent years; examples are CCS [14], SCCS [12], CSP [4], MEIJE [1], ACP [2], CIRCAL [10]. These algebras give a semantic account of different ways to combine networks; consequently they contain a variety of operators such as nonde-terministic choice and sequential and parallel composition. In contrast, in the language in the present paper the only operators are parallel composition and interlinking of processes. We contend that these operators form a sufficient basis for the study of many fundamental properties of synchronising parallel processes; in particular we will here explore the expressive power as measured by the definable terms and operators in order to gain insight in what phenomena can be derived from parallelism and linking alone. This insight is relevant for an understanding and comparison of other semantic accounts of parallelism and may prove useful in situations such as hardware design, where parallel composition and linking are the only ways to combine networks.

An inspiration for our language comes from data flow networks. In a data flow network, each functional unit has a set of input ports and a set of output ports. The units receive data on the input ports, perform computations on these data, and transmit the results on the output ports. Communication between units is through links; each link connects one output port with one input port. The links remain unchanged when the units execute. Typically, a data flow network realises some complex function, using only a set of standard (predefined) functional units. The semantics of data flow has been the subject of many papers [8,15,9,21,7]. The networks in the present paper will be reminiscent of data flow nets, but the links will be used for synchronisations of rendezvous type (two units must participate in events on a link simultaneously). Any buffering on a link must be modelled explicitly as an intermediary unit acting as a buffer. This synchronous communication is in accordance with the usual semantics of process algebras.

In an algebraic language for description of networks the operators correspond to ways in which larger networks can be built from smaller networks. In our language there are two types of operators. One is disjoint parallelism. With this operator, two existing networks can be put in parallel without any internetwork links, thus the networks will execute independently of each other. This corresponds to "disjoint union" [21] and
"aggregation" [9]. The other operator type is linking: two ports of a network can be linked, thus enforcing synchronisation between the events on these ports. Each port can be attached to at most one link; more complicated structures (such as broadcasting or multi way communication) must be accomplished through intermediate units. The linking operators are inspired by the "linking" [21] and "loop" [9]; to our knowledge the present paper is the first attempt to give them an operational semantics.

Other algebras for description of synchronously communicating networks, for example CCS and CSP, use other families of operators in order to describe parallelism and linking between units. One idea common to these algebras is that the parallel operator implicitly achieves linking. For example, in CCS two ports with complementary names will automatically be linked in a parallel composition, and in CSP two ports with the same name will be linked. Moreover, both these algebras allow a port to be linked to more than one other port, but they give different semantics to such multiple linking: in CCS a synchronisation event always involves events on exactly two linked ports, while in CSP an event on a port must involve events on all linked ports. This makes it impossible to directly define the parallel operators in CCS and CSP in terms of each other. Similarly, in SCCS and ACP the linking between units is partially determined by a function on port names; this function is not an operator but rather a parameter which determines the semantics of the parallel operator. Our position is that our algebraic language, where parallelism and linking are directly reflected as different operators, provides a more natural basis for the study of fundamental properties of parallelism.

Our main results are that with only three simple types of predefined units, any finite-state behaviour can be defined, and the operators normally used to describe parallel structures in many other process algebras are definable. We also determine that some operators related to non-determinism and sequential composition are not definable.

Related work includes Milner's investigation [13] of the definable behaviours in a language containing only prefixing and choice operators, and de Simone's investigation [5] of the definable operators in the synchronous process algebras MEIJE and SCCS. One main difference between the present work and de Simone's is that de Simone considers a synchronous form of composition as a primitive operator. This operator forces its arguments to execute in lock-step, as if synchronised by a global clock. In contrast our primitive parallel operator is asynchronous; this implies that fewer operators are definable. Another difference is that the results in the present paper are valid for a wide range of behaviour equivalences, and not only for observation equivalence based on bisimulation.

The work presented here is an elaboration and continuation of [16,17] by the same author, and a related paper [19] examines different models and axiomatisations of the algebraic language.

In Sections 2 and 3 below we present the syntax and semantics of the algebraic language. Section 4 is devoted to examples of networks and their behaviours. In Section 5 we define notions of behaviour equivalence. Section 6 contains our first expressiveness result: with only three simple behaviours, any finite-state behaviour can be constructed. In Section 7 we present a class of operators which are definable within the language, and discuss to what extent operators from other algebras are definable. Section 8 contains some final remarks and open problems. We will only give sketches for the proofs; complete proofs can be found in the report [18].

2. Syntax

The purpose of this is to define an algebraic language where the terms correspond to networks of interconnected units. Fig. 1 illustrates an example of such a network. There are three units, each with a set of ports. Each port is assigned an
internal identification number which is unique within the unit. Two ports in a network may be connected by a link; a port so connected is called \textit{internal} to the network. A port not connected by a link is called \textit{external}, and has a unique (for the whole network) \textit{name}. In Fig. 1 the network has four internal ports, and three external ports named $a$, $b$ and $c$.

A network may contain several units which are instances of the same \textit{module}. A module can be thought of as a schematic unit or a template for a unit; intuitively, we expect different instances of a module to exhibit the same "behaviour" but on different ports. In Fig. 1 there are two instances of a module $M$ and one instance of a module $M'$. Each module has a nonnegative \textit{arity} which is the number of ports of the module. In the example, $M$ has arity 2 and $M'$ has arity 3.

So, in the following we assume a fixed potentially infinite set $\Lambda$, called the set of \textit{port names}, and use $a$, $b$, $c$, \ldots to range over $\Lambda$. We let $M$ range over sets of modules, and use $M$, $M'$ etc. to range over modules.

\textbf{Definition 1.} $\mathcal{T}_M$ (the set of terms over $M$), in the following ranged over by $A$, $B$, \ldots is the least set satisfying the following clauses:

1. If $M \in M$ and $M$ has arity $n$, and $a_1, \ldots, a_n$ are port names, then $M(a_1, \ldots, a_n) \in \mathcal{T}_M$.
2. If $A \in \mathcal{T}_M$ and $B \in \mathcal{T}_M$ then $(A \parallel B) \in \mathcal{T}_M$.
3. If $A \in \mathcal{T}_M$ and $a$ and $b$ are distinct port names then $(A(a \parallel b)) \in \mathcal{T}_M$.

We adopt the convention that $| \parallel $ associates to the left and that $(a \parallel b)$ binds tighter than $|$. This allows us to drop some of the parentheses. For example, $A \parallel B(a \parallel b) \parallel C$ means $((A \parallel (B(a \parallel b))) \parallel C)$. A term of type $M(a_1, \ldots, a_n)$ corresponds to an atomic network consisting of only one unit obtained from the module $M$; the external port names of this network are given by $a_1, \ldots, a_n$. The operator $|$ is called the \textit{parallel} operator; the intention is that $A \parallel B$ corresponds to a network obtained as the union of the networks of $A$ and $B$ without any internetwork links. The operators of type $(a \parallel b)$ are called \textit{linking} operators; a term $A(a \parallel b)$ denotes the network $A$ where the ports named $a$ and $b$ have been joined by a link. As an example, the network in Fig. 1 can be described by the term $((M(d, a) \parallel M'(e, b, f)) \parallel M(g, c)) \langle a \parallel e \parallel f \rangle$.

The sort of a term is the set of external port names of the corresponding network:

\textbf{Definition 2.} The sort $L(A)$ of a term $A$ is defined inductively as follows:

1. $L(M(a_1, \ldots, a_n)) = \{a_1, \ldots, a_n\}$
2. $L(A \parallel B) = L(A) \cup L(B)$
3. $L(A(a \parallel b)) = L(A) \setminus \{a, b\}$

We will use $L$, $L'$, \ldots to range over sorts (i.e. finite subsets of $\Lambda$). We can now formalise the requirement that external port names are unique in a network, and that each internal port can be attached to at most one link: we call such terms \textit{well-formed}.

\textbf{Definition 3.} (Well-formed terms).

1. $M(a_1, \ldots, a_n)$ is well-formed if all $a_i$ are pairwise distinct.
2. $A \parallel B$ is well-formed if $A$ and $B$ are well formed and $L(A) \cap L(B) = \emptyset$.
3. $A(a \parallel b)$ is well-formed if $A$ is well formed and $a, b \in L(A)$.

In the rest of this paper we will exclude non-well-formed terms, so we let $\mathcal{T}$ stand for the well-formed terms and call those just "terms". A summary of the definitions appears in Table 1.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Term} & \textbf{Condition on formation} & \textbf{Sort} \\
\hline
$M(a_1, \ldots, a_n)$ & $n$ is the arity of $M$, $i \neq j \Rightarrow a_i \neq a_j$ & $\{a_1, \ldots, a_n\}$ \\
$A \parallel B$ & $L(A) \cap L(B) = \emptyset$ & $L(A) \cup L(B)$ \\
$A(a \parallel b)$ & $a \neq b$, $a, b \in L(A)$ & $L(A) \setminus \{a, b\}$ \\
\hline
\end{tabular}
\caption{Summary of the formation rules of the algebraic language, and the definition of sort}
\end{table}

\section*{3. Operational semantics}

We will present the operational semantics for the language in a way that has become standard for process algebras: through a family of labelled binary relations $\rightarrow$, so called transition on relations, on terms. Our definition will be in the form of a Plotkin-style induction on the structure of terms. If desired, from this definition a formal interpretation of terms into transition diagrams can be obtained in a completely standard way.
The label \( \alpha \) of a transition \( A \xrightarrow{\alpha} B \) is called the action of the transition. An action is a set of port names\(^1\) and we use \( \alpha, \beta, \gamma, \ldots \) to range over actions. The intended meaning of the transition \( A \xrightarrow{\alpha} B \) is that the network corresponding to \( A \) can evolve to the network corresponding to \( B \) by participating in synchronisations on all ports in \( \alpha \). There may be several different transitions from a term with the same action; this amounts to non-determinism in the term. Note that the empty set \( \emptyset \) is also an action, this corresponds to an internal action (cf. \( \tau \) in CCS).

We take the view that the operational semantics of terms is dependent on an operational interpretation of the modules. An operational interpretation is a family of schematic transition relations where the “actions” refer to the internal port numbers within the module:

**Definition 4.** An operational interpretation on a set of modules \( \mathcal{M} \) is a set of schematic transitions of the form

\[
M \xrightarrow{K} M'
\]

where \( M, M' \in \mathcal{M} \) are of the same arity \( n \), and \( K \) is a subset of \( \{1, \ldots, n\} \).

The intention is that \( M \xrightarrow{K} M' \) means that the module \( M \) can evolve to the module \( M' \) by participating in synchronisations on the ports in \( K \). Note that \( M' \) must have the same arity as \( M \): a module cannot change its number of ports when it executes.

In the following we assume that every set of modules is associated with an operational interpretation. We now define the transition relations \( \xrightarrow{\alpha} \) on terms:

**Definition 5.** The family of labelled transition relations on \( \mathcal{F}_\mathcal{M} \) consists of the least relations \( \xrightarrow{\alpha} \) satisfying the rules in Table 2.

The rule for units says that \( M(a_1, \ldots, a_n) \) can do exactly what is decreed by \( M \) if the ports are named \( a_1, \ldots, a_n \). The first and second rules for parallel composition say that \( A \parallel B \) can do whatever \( A \) or \( B \) can do in isolation. The third rule says that if both \( A \) and \( B \) can do something, then \( A \parallel B \) can do the union of the actions. The parallel operator expresses a form of independent parallelism — \( A \) and \( B \) execute asynchronously side by side without affecting each other.

For the linking operator, the intuition is that if \( A \) can do an action involving neither \( a \) nor \( b \), then \( A(a^\land b) \) can do the same action (the link has no effect at all). If \( A \) can do an action involving both \( a \) and \( b \), then \( A(a^\land b) \) can do the same action, now with \( a \) and \( b \) removed (intuitively this action involves a synchronisation over the link). As a consequence, actions of \( A \) involving exactly one of \( a \) and \( b \) are disallowed in \( A(a^\land b) \) (such actions would correspond to synchronisations where only one endpoint of the link is involved). While the parallel operator is independent parallelism, the linking operator is used to explicitly express dependencies: actions involving different endpoints of the link must be synchronised.

### Examples

In this section we present some examples of terms, their corresponding networks, and their operational semantics in the form of behaviours. A behaviour is a graph where nodes are terms and labelled edges correspond to transitions between terms; the behaviour of a term is simply a behaviour containing that term as well as all transitions from the terms in the graph. Networks will be displayed as in Fig. 1, and the internal port numbers of modules will be omitted whenever they are unimportant or can be inferred from the shape of the graphic symbol of the module. Note that behaviours and networks serve only to infor-

---

\(^1\) Actions as sets of port names are also used in CIRCAL [10]; other process algebras tend to use either single port names or multisets.
mally illustrate terms and their operational semantics. In a companion paper [19] we define behaviours and networks formally and explore algebras over them.

To begin we introduce four modules SY, AR, AL, AL'; the first two of arity three and the last two of arity two. We assume that the schematic transitions of these modules are

\[
\begin{align*}
\text{SY} & \rightarrow^{(1,2,3)} \text{SY} \\
\text{AR} & \rightarrow^{(1,2)} \text{AR} \\
\text{AL} & \rightarrow^{(1)} \text{AL'} \\
\text{AL} & \rightarrow^{(1,2)} \rightarrow^{(1,3)} \rightarrow^{(1,4)} \text{AL}
\end{align*}
\]

Instances of these modules are depicted in Fig. 2; for each unit we give its behaviour (top), and a symbol to be used in graphic descriptions of networks (bottom).

The first unit SY(a, b, c) is a three-way synchroniser. It has three ports; its behaviour is to repeatedly perform an action involving all three ports. A three-way synchroniser can thus be used as an intermediate unit to link three ports. In the same way, we could assume modules for n-way synchronisers SY_n of arity n for arbitrary n: the schematic transitions would be

\[
\text{SY}_n \rightarrow^{(1,\ldots,n)} \text{SY}_n.
\]

However, n-way synchronisers are definable by three-way synchronisers in the following sense: for each instance of an n-way synchroniser there exists a term with the same behaviour (this notion will be made precise in Section 5), containing only three-way synchronisers. For example, the term

\[
(\text{SY}(a, b, f) \mid \text{SY}(g, c, d))(f^* g)
\]

has the same behaviour as SY_4(a, b, c, d).

The second unit AR(a, b, c) is an arbiter. It repeatedly synchronises its right-hand port a with exactly one of its left-hand ports b and c. This represents another way to interconnect three ports: imagine the left-hand ports to be competing for the privilege of synchronising with the right-hand port. In analogy with the n-way synchronisers it is possible to define n-way arbiters, with n left-hand ports, as a combination of arbiters. For example, a three-way arbiter AR_3 would have three schematic transitions labelled \{1, 2\}, \{1, 3\} and \{1, 4\}. Again, arbiters of higher arity turn out to be definable by ordinary arbiters. For example, the term

\[
(\text{AR}(a, b, f) \mid \text{AR}(g, c, d))(f^* g)
\]

has the same behaviour as AR_3(a, b, c, d).

The third unit AL(a, b) in Fig. 2 is an alternator: it alternatingly synchronises on its two ports. We use two different modules (AL and AL') to represent the two states. In the network symbol we use a small filled circle to indicate the state of the alternator: when this circle is to the left (module AL), the next synchronisation will be on the left-hand port, and if it is to the right (module AL') it will be on the right-hand port. Since AL'(a, b) has the same behaviour as AL(b, a) we will only make use of the AL symbol.

A three-way alternator AL_3, i.e. a module which repeatedly synchronises on its three ports in se-
sequence, is depicted in Fig. 3. Formally the schematic transitions of $AL_3$ are

$$AL_3^{(1)} \rightarrow AL_3' \quad AL_3^{(2)} \rightarrow AL_3'' \quad AL_3^{(3)} \rightarrow AL_3.$$  

In fact, the three-way alternator is definable by using three ordinary alternators and three synchronisers: the term

$$(SY(a, f, g) | AL(h, i) | SY(j, b, k) | AL(l, m) | SY(n, c, o) | AL(q, p))$$

$$(f \cdot h)(i \cdot j)(k \cdot l)(m \cdot n)(o \cdot p)(q \cdot g)$$

has a behaviour which is precisely the transition graph in Fig. 3. The network corresponding to this term is also present in Fig. 3 (bottom).

As a more complicated example, assume that we wish to construct a Controller $CONT$ for a critical resource. This Controller should have the behaviour as indicated in Fig. 4 (left). It has four ports: $use_1$, $rel_1$, $use_2$, and $rel_2$. The intention is that $use_1$ and $rel_1$ are connected to one process, and $use_2$ and $rel_2$ to another process. A process signals the use of the critical resource with an action on its $use$ port and the release of the resource with an action on its $rel$ port. The Controller makes sure that at most one process has the

![Fig. 3. A three-way alternator: Behaviour and equivalent network.](image)

![Fig. 4. A Controller $CONT$ for a critical resource: Behaviour and network.](image)
resource at a time, and that only the process which currently has the resource may release it. A construction of such a Controller is also given in Fig. 4 (right). It uses two alternators to remember which process last acquired the resource, and one alternator to remember whether the resource is available; it also uses two arbiters and four synchronisers to connect these alternators with the external ports. The term corresponding to this network consists of nine units in parallel and ten linking operators — we will not exhibit this term here.

These examples make clear that interesting behaviours can be defined using only a small set of modules; we will return to this topic in Section 6. Our language can also be used to illuminate the relationships between other formalisms for concurrency. Recall for example the parallel operators in CCS and CSP (Fig. 5). In CCS, two parallel agents which share a port name $a$ will compete for the use of $a$ in synchronisations. When the environment of these agents perceives an $a$ action, then either of the agents, but not both, may be the source of the $a$. Thus the parallel composition in CCS can be considered to implicitly use arbiters to resolve the use of shared port names. When the environment of these agents perceives an $a$ action, then either of the agents, but not both, may be the source of the $a$. Thus the parallel composition in CCS can be considered to implicitly use arbiters to resolve the use of shared port names. On the other hand, in CSP two parallel processes sharing a port $a$ will synchronise on that port: when the environment perceives an $a$ action, then both processes must contribute on their respective ports. Thus parallel composition in CSP implicitly makes use of synchronisers to resolve shared port names.

As a final example, it is straightforward to encode a simple version of place/transition nets [20]. In such a net there are two types of modules: transitions and places. A transition with indegree $m$ and outdegree $n$ is an instance of a module $TR_{m,n}$ of arity $m + n$; this module has the only schematic transition

$$TR_{m,n} \xrightarrow{1, \ldots, m+n} TR_{m,n}.$$ 

In fact, $TR_{m,n}$ is just an $(m+n)$-way synchroniser: it requires interaction on all its ports. The intuition is that for a transition to fire, tokens must be received on all incoming arcs, and tokens will be produced on all outgoing arcs. A place with indegree $m$ and outdegree $n$ currently holding $k$ tokens is an instance of a module $PL_{m,n,k}$ of arity $m + n$; the schematic transitions for this module are all transitions

$$PL_{m,n,k} \xrightarrow{K} PL_{m,n,k'},$$

satisfying

$$k \geq \text{out}(K) \quad \text{and} \quad k' = k + \text{in}(K) - \text{out}(K)$$

where $\text{in}(K)$ is the number of integers between 1 and $m$ in $K$, and $\text{out}(K)$ is the number of integers between $m + 1$ and $m + k$ in $K$. The intuition is that a place can emit tokens on its outgoing arcs, but it cannot emit more tokens than it currently holds, and the number of tokens will be modified according to tokens received and emitted. Note that the “behaviour” of such a place is not finite-state (since $k$ can grow unboundedly), so it cannot be described in terms of the other modules in this section.

2 I am grateful to Tom Verhoeff for this example.
In the encoding of a place/transition net, a link will always connect an “in”-port of a unit (a port with a number less than or equal to \(m\) in an instance of either of \(TR_{m,n}\) or \(PL_{m,n,k}\)) with an “out”-port (a port which is not an in-port) of a unit of the other type. It is trivial to verify that with our encoding the firing rule for place/transition nets is enforced, and that a synchronisation on a link occurs when a token “travels” along that link (either from a place to a transition or vice versa). Note that our encoding allows several transitions to fire in the same action. If a strict interleaving of transitions is desired then the transitions can additionally be connected to a multiway arbiter which only allows one transition to fire at a time.

5. Behaviour equivalences

Intuitively, two terms should be regarded as equivalent if their respective behaviours are sufficiently similar. Many such equivalences have been proposed in the literature (observation equivalence, failure equivalence, testing equivalence, trace equivalence etc.). Our results will hold for any equivalence which lies between observation equivalence (=) and trace equivalence (=\(\tau\)) as defined below. We call such an equivalence a behaviour equivalence. We require equivalent terms to have the same sort; without this requirement the congruence result (Theorem 1) below would not hold.

We use \(\sigma\) to range over sequences of non-empty actions, and write \(\langle \rangle\) for the empty sequence.

Definition 6.

1. \(A \Rightarrow A'\) if

\[
\begin{align*}
3n \geq 0: & A \rightarrow \cdots \rightarrow A' \quad \text{if } \sigma = \emptyset \\
\text{n times} & \\
3n, m \geq 0: & A \rightarrow \cdots \rightarrow a \rightarrow \cdots \rightarrow A' \\
\text{n times} & \quad \text{m times}
\end{align*}
\]

2. A binary relation \(\mathcal{R}\) on terms is a simulation if \(A \mathcal{R} B\) implies

For all \(\sigma, A': A \Rightarrow A'\) implies that there exists \(B': B \Rightarrow B'\) and \(A' \mathcal{R} B'\)

3. \(\mathcal{R}\) is a bisimulation if \(\mathcal{R}\) and \(\mathcal{R}^{-1}\) are simulations.

4. \(A \Rightarrow B\) if \(L(A) = L(B)\) and there exists a bisimulation \(\mathcal{R}\) such that \(A \mathcal{R} B\).

5. \(A \Rightarrow A'\) if \(A \Rightarrow A'\), and \(A \Rightarrow A'\) if \(A \Rightarrow \cdots \Rightarrow A'\).

6. \(A \Rightarrow (\sigma\) is a trace of \(A\)) if for some \(A': A \Rightarrow A'\).

7. \(A =_\tau B\) if for all \(\sigma: (A \Rightarrow \sigma \text{ if } B \Rightarrow \sigma)\) and \(L(A) = L(B)\).

8. An equivalence \(=\) on terms is a behaviour equivalence if \(\approx \subseteq = \subseteq =\tau\).

Definition 7. An equivalence \(=\) is a congruence if it is preserved by the operators, i.e. if \(A \Rightarrow A'\) and \(B \Rightarrow B'\) then

1. \(A \| B\) is a term iff \(A' \| B'\) is a term, and if so \(A \| B = A' \| B'\).

2. \(A\langle a \cdot b\rangle\) is a term iff \(A'\langle a \cdot b\rangle\) is a term, and if so \(A\langle a \cdot b\rangle = A'\langle a \cdot b\rangle\).

Our first theorem is that observation equivalence and trace equivalence are congruences:

Theorem 1. \(\approx =\) and \(=_\tau\) are congruences.

We omit the proof since it is similar to proofs of similar known results. In the following we restrict attention to behaviour equivalences which are congruences.

6. A basis for finite-state terms

In a practical design situation, it will be useful to know what modules are necessary to build a certain set of behaviours (up to some behaviour equivalence). Conversely, it will be useful to know what behaviours can be constructed given a certain set of modules. Such results also have intrinsic theoretical interest: they illuminate the expressive power of the operators. The main result in this section is that the three modules SY, AR, and AL from Section 4 suffice to build any finite-state behaviour. This indicates that our operators exhibit a considerable expressive power.

In the following \(\mathcal{F}\) ranges over sets of terms, and we assume a fixed behaviour equivalence \(\approx\) on terms.
Definition 8.
1. \( M \) is a basis for \( \mathcal{T} \) if each term in \( \mathcal{T} \) is equivalent with a term in \( \mathcal{T}_M \).
2. \( M \) is a proper basis for \( \mathcal{T} \) if it is a basis, and each term in \( \mathcal{T}_M \) is equivalent with a term in \( \mathcal{T} \).
3. \( M \) is independent of \( M \) if some instance of \( M \) is not equivalent with any term in \( \mathcal{T}_M \).
4. \( M \) is independent if, for all \( M \in M \), the module \( M \) is independent of \( \mathcal{T}_M \).

Intuitively, “\( M \) is a basis for \( \mathcal{T} \)” means that the set of behaviours represented by \( \mathcal{T} \) can be constructed (up to equivalence) with modules in \( M \). If in addition \( M \) is proper, then \( \mathcal{T} \) represents all behaviours that can be constructed, and if \( M \) is independent, then all modules in \( M \) are actually necessary (no module can be removed).

We can now ask many intriguing questions. For example, given an interesting set of terms \( \mathcal{T} \), what is an independent proper basis? Conversely, given an interesting set of modules \( M \), is it independent, and is it a proper basis for an interesting set of terms? We will answer only one such question here, and leave the rest as topics for further research.

Definition 9. The state space of a term \( A \) is the set \( \{ B : \text{for some } \sigma : A \overset{\sigma}{\Rightarrow} B \} \). A term is finite-state if its state space is finite.

Consider the modules SY, AR, and AL with associated schematic transitions as defined in Section 4. The main theorem of this section is (note that it holds regardless of the choice of \( \overset{\sigma}{\Rightarrow} \)):

Theorem 2. \( \{ \text{SY, AR, AL} \} \) is an independent proper basis for the set of finite-state terms.

Proof (sketch). To show that this set of modules is a basis, we must show that for every finite-state term \( A \), there is an equivalent term \( B \) in \( \mathcal{T}_{\{ \text{SY,AR,AL} \}} \). The proof is by direct construction of a network from the behaviour of \( A \). The idea is that every state in \( A \) corresponds to one alternator. Transitions between states in \( A \) correspond to links between the alternators; each alternator has one port for incoming transitions and one port for outgoing transitions. If a state has several outgoing or incoming transitions then arbiters are used: all links corresponding to incoming transitions are routed through arbiters to one port of the alternator, and all links corresponding to outgoing transitions are routed to the other port. If a transition has a non-empty action, then a three-way synchroniser is interposed on the corresponding link: the third port of this synchroniser is used to generate the action. Examples of such constructions can be found in Section 4, Fig. 3 and Fig. 4. The desired term \( B \) can be defined as a term corresponding to this network; we can then prove \( A \overset{\sigma}{\Rightarrow} B \), which implies \( A \overset{\sigma}{=} B \).

To prove that the basis is proper, it suffices to observe that instances of \( \text{SY, AR and AL} \) are all finite-state, and that the operators (parallelism and linking) preserve the property of being finite-state.

Finally, to prove that \( \{ \text{SY, AR, AL} \} \) is independent, we establish for each \( M \in \{ \text{SY, AR, AL} \} \) a property \( \phi_M \) of terms satisfying the following conditions:
1. Instances of the two modules in \( \{ \text{SY, AR, AL} \} \) which are not \( M \) all have property \( \phi_M \).
2. \( \phi_M \) is preserved by the operators (parallel and linking);
3. \( \phi_M \) is preserved by trace equivalence;
4. Some instance of \( M \) doesn’t have property \( \phi_M \).

Thus, \( (1) \) and \( (2) \) guarantee that all terms over \( \{ \text{SY, AR, AL} \} \) have property \( \phi_M \), and \( (3) \) and \( (4) \) guarantee that an instance of \( M \) is not trace equivalent, and hence not \( = \), with any such term. The property \( \phi_{\text{AL}} \) is defined to hold for a term if it is trace equivalent with a term which has exactly one state. The property \( \phi_{\text{AR}} \) is intersection closure: a term is intersection closed if whenever it can do actions \( \alpha \) and \( \beta \) it can also do \( \alpha \cap \beta \). Finally the property \( \phi_{\text{SY}} \) is partition closure: this holds for a term \( A \) if whenever \( A \) can do an action containing more than two ports, then this action can be partitioned into smaller actions with at most two ports each such that \( A \) can do any union of these smaller actions.

It is interesting to note that this result holds also for equivalences stronger than observation equivalence. In fact, the proof carries over to “strong equivalence” (bisimulation equivalence defined in terms of \( \overset{\sigma}{\Rightarrow} \) rather than \( \Rightarrow \), i.e. internal actions are significant).
An alternative way to phrase the ideas in this section is to define a formal interpretation of terms into behaviours and define the equivalences directly on behaviours (rather than terms), and say that a behaviour is definable if there is a term with an equivalent behaviour. Theorem 2 then implies that any finite-state behaviour is definable even if no other modules than SY, AR and AL are used, i.e. that any finite-state behaviour can be built from three-way synchronisers, arbiters, and alternators.

One other result is that \{SY, AR\} is a proper independent basis for the set of “one state” terms. We do not know any interesting basis for any other set of terms. Note that some sets of terms which have traditionally been regarded as interesting, viz. the “deterministic” terms and the “confluent” terms [14] are not closed under the linking operator, and hence can not have a proper basis.

7. Definability of operators

In this section we examine to what extent other operators are definable within the language. Briefly put, an operator is definable if there is an equivalent combination of parallel and linking operators. As an example, assume that we desire a three-way linking operator \(\langle a \land b \land c \rangle\): the intention is that in the term \(A \langle a \land b \land c \rangle\), the three ports \(a, b, c\) in \(A\) should be linked. We could of course extend the language with a new class of operators \(\langle a \land b \land c \rangle\), and give these operators operational definitions (in the same way as \(\langle a \land b \rangle\):

\[
\begin{align*}
A & \to A' \ a, b, c \notin a \\
A \langle a \land b \land c \rangle & \to A' \langle a \land b \land c \rangle \\
A & \to A' \ a, b, c \in a \\
A \langle a \land b \land c \rangle & \to A' \langle a \land b \land c \rangle \quad \text{if } a \in \{a, b, c\}
\end{align*}
\]

However, such an extension would in one sense be redundant: the effect of \(\langle a \land b \land c \rangle\) can be achieved with a three-way synchroniser (cf. Fig. 2) as follows. Let \(A\) be a term with \(a, b, c\) in its sort, and choose port names \(d, e, f\) not in the sort of \(A\). We can then prove that

\[
A \langle SY(d, e, f) \rangle \langle a \land b \land c \rangle \approx A \langle a \land b \land c \rangle.
\]

Thus we say that if \(SY(d, e, f)\) (or an equivalent term) is definable, then three-way linking is definable in the language: a designer can confidently use three-way links when constructing networks, and later expand these links according to the definition. We will investigate which operators are definable in this sense. We will give a general definability result, and discuss some examples of operators from other process algebras. As might be expected, whether an operator is definable depends critically on which equivalence is used. It turns out that more operators are definable up to trace equivalence than up to observation equivalence.

In the following we write \(\tilde{A}\) for the sequence of terms \(A_1, \ldots, A_n\) (here \(n \geq 0\), similarly \(\tilde{A}'\) will mean the sequence \(A'_1, \ldots, A'_n\). We write \(\tilde{L}\) for the sequence of sorts \(L_1, \ldots, L_n\), and say that \(\tilde{A}\) has sort \(\tilde{L}\) if \(L(A_k) = L_k\) for all \(k\) \((1 \leq k \leq n)\). We will introduce new operators ranged over by \(\langle a \land b \land c \rangle\), and assume that each such operator has an arity \(n \geq 0\) and a type \(L \to L\).

Definition 10. The set \(\mathcal{T}_{op, M}\) of terms over a set \(op\) of operators and a set \(M\) of modules is defined by generalising definitions 1-3 in the obvious way: if \(A_i \in \mathcal{T}_{op, M}\) for all \(i = 1, \ldots, n\), similarly \(\tilde{A'}\) will mean the sequence \(A'_1, \ldots, A'_n\). We write \(\tilde{L}\) for the sequence of sorts \(L_1, \ldots, L_n\), and say that \(\tilde{A}\) has sort \(\tilde{L}\) if \(L(A_k) = L_k\) for all \(k\) \((1 \leq k \leq n)\). We will introduce new operators ranged over by \(\langle a \land b \land c \rangle\), and assume that each such operator has an arity \(n \geq 0\) and a type \(L \to L\).

For example, a three-way linking operator \(\langle a \land b \land c \rangle\) would be of type \(L \to (L \setminus \{a, b, c\})\) for a sort \(L\) containing \(a, b, c\). Notice that each operator has a fixed type, so there will be one three-way linking operator for each such \(L\). Strictly speaking our original operators \(\langle a \land b \rangle\) are families of operators for the same reason, but for convenience we will continue to refer to them as “operators”. We let \(\langle a \land b \rangle\) be tacitly present in any set \(op\) under consideration; with this convention \(\mathcal{T}_{op, M}\) is just \(\mathcal{T}_M\).

In what follows we assume that whenever \(op\) is a set of operators, a fixed definition of the operational semantics of \(op\) determines the labelled transition relations on terms in \(\mathcal{T}_{op, M}\). For example, a family of three-way linking operators can be given an operational semantics as in the beginning of this section. We do not require that the definition of the operational semantics is presented in a particular format; formally an operational semantics is just a set of transitions of type \(A \xrightarrow{a} B\) which agree with the rules in Table 2. Definition 7
generalises directly to terms in $\mathcal{T}_{op,M}$, so it is possible to talk about behaviour equivalences on $\mathcal{T}_{op,M}$. This is essential for our definition of definability below.

**Definition 11.** An $n$-ary context $\mathcal{C}$ (here $n \geq 0$) over the modules $M$ is a term in $\mathcal{T}_M$ with $n$ numbered holes in it, and we write $\mathcal{C}(\bar{A})$ for the term obtained by inserting $A_1, \ldots, A_n$ in the holes in $\mathcal{C}$.

To make this definition precise we could introduce variables in the language and talk about terms with variables, and substitution of terms for variables. We trust the reader to accept our more simplistic definition.

**Definition 12.** An operator $op$ of type $\tilde{L} \rightarrow L$ is $\equiv$-definable over $M$ (or simply definable if $M$ and the equivalence $\equiv$ are implicit) if there exists a context $\mathcal{C}$ over $M$ such that for all $\bar{A}$ of sort $\tilde{L}$, $\mathcal{C}(\bar{A}) = op(\bar{A})$.

We call such a $\mathcal{C}$ an $\equiv$-defining context for $op$.

As an example, take a unary three-way linking operator $\langle a^* b^* c \rangle$ of sort $L \rightarrow L \{-a,b,c\}$ for an $L$ not containing $d$, $e$, or $f$. This operator is $\equiv$-definable over $\{SY\}$:

$\mathcal{C}(.) = (. \mid SY(d, e, f)) \langle a^* d \rangle \langle b^* e \rangle \langle c^* f \rangle$

is a defining context. In the rest of this section we aim to show that an operator is definable precisely if its operational semantics can be presented in a particular format. This will require some preliminary definitions.

**Definition 13.** A de Simone rule is a tuple $(op, \mu_1, \ldots, \mu_n, \alpha, op')$ where $op$ and $op'$ are (possibly the same) $n$-ary operators of the same type $\tilde{L} \rightarrow L$, each $\mu_i$ is either a subset of $L_i$ or the special symbol $\ast$, and $\alpha$ is a subset of $L$.

A de Simone rule $\langle op, \mu_1, \ldots, \mu_n, \alpha, op' \rangle$ will be considered as an inference rule

$\begin{align*}
\llbracket A_i = A'_i \mid \mu_i = \ast, \quad A_i^{\mu_i} A'_i \quad \text{otherwise} \rrbracket \text{ if } i = 1, \ldots, n,
\end{align*}$

$\begin{align*}
op(\bar{A}) \overset{\alpha}{\Rightarrow} op'(\bar{A'})
\end{align*}$

The rule has $n$ premises, and premise number $i$ is \"$A_i = A'_i$\" if $\mu_i = \ast$, otherwise the premise is \"$A_i^{\mu_i} A'_i$\". The rule should be read: \"If the premises hold for the terms $\bar{A}$, $\bar{A}'$, then the conclusion also holds.\"

**Definition 14.** A set $op$ of operators is called a de Simone set if the transitions on terms in $\mathcal{T}_{op,M}$ are exactly the transitions which can be proven from a set of de Simone rules. An operator is a de Simone operator if it is in a de Simone set.

The class of de Simone operators was first suggested by Robert de Simone [5] who proved that these operators are exactly the definable operators in the synchronous algebras MEIJE and SCCS. We use the same definition, only slightly adapted to our framework. Practically all operators studied in process algebras are de Simone operators (in two recent papers [6,3] more general rule formats are suggested). As an example, the singleton set containing only a three-way linking operator constitutes a de Simone set, so three-way linking is a de Simone operator. Also, from Table 2 it is evident that parallel and linking are de Simone operators. Intuitively, an operator $op$ is a de Simone operator if the transitions from a term $op(\bar{A})$ can be inferred from the transitions of $\bar{A}$ alone, and if the derivative $op'(\bar{A}')$ contains each derivative $A'_i$ from the premises exactly once.

Since our parallel operator is asynchronous, it turns out that not all the de Simone operators are definable. As a simple example consider the unary operator $ext$ of type $\emptyset \rightarrow \{a\}$ with the only rule

$\begin{align*}
\frac{A \rightarrow A'}{ext(A) \overset{\{a\}}{\Rightarrow} ext(A')}
\end{align*}$

(in the more compact rendering as a tuple this rule is $\langle ext, \emptyset, \{a\}, ext \rangle$). Intuitively, $ext$ transforms internal actions $\emptyset$ into external actions $\{a\}$. This operator is not definable in our language. A simple proof of this is to observe that $ext$ does not respect trace equivalence: if $A$ and $B$ are two terms of sort $\emptyset$ which differ in the number of internal actions they can perform, then $A = _T B$ but $ext(\bar{A}) \neq _T ext(\bar{B})$. Since parallel and linking respect $=_T$ it follows that $\mathcal{C}(A) = _T \mathcal{C}(B)$ for all contexts $\mathcal{C}$. Hence no context can be a defining context for $ext$.

In essence, the definable operators in our algebraic language turn out to be the de Simone
operators that cannot distinguish between internal actions and absence of actions in their arguments:

Definition 15. A set of de Simone rules is called an asynchronous rule set if it satisfies the following requirements:

1. \( \emptyset \) and \(*\) occur interchangeably in the set, i.e. a rule \( r = \langle op, \mu_1, \ldots, \mu_n, \alpha, \ op \rangle \) with \( \mu_i = \emptyset \) is in the set if and only if a rule which differs from \( r \) only in that \( \mu_i = * \) also is in the set. There is one exception from this requirement: rules of type \( \langle op, *, \ldots, *, \emptyset, \emptyset \rangle \), where the operators are the same, all \( \mu_i \) are \(*\), and \( \alpha = \emptyset \) are not required to be in the set.

2. The set contains an idling rule \( \langle op, \emptyset, \ldots, \emptyset, \emptyset, \emptyset \rangle \) for each operator \( op \) occurring in a rule in the set.

The first requirement above ensures that an operator treats the absence of an action in an argument \((*)\) in the same way as an internal action \((\emptyset)\). An idling rule in the second requirement can be rendered

\[
A_1 \rightarrow A'_1 \cdots A_n \rightarrow A'_n.
\]

\( op(\vec{A}) \rightarrow op(\vec{A}') \)

The idling rule ensures that if all arguments \( A_i \) of \( op \) idle (i.e. do an internal action) then \( op(\vec{A}) \) must also have the possibility to idle. In other words, \( op \) cannot prevent its arguments from doing internal actions.

The exception in the first requirement of Definition 15 is a technical convenience only. Without this exception the requirements would imply that an asynchronous set would always contain a rule

\[
\begin{align*}
A &\rightarrow^\emptyset A \\
\Rightarrow op(\vec{A}) \rightarrow \emptyset op(\vec{A})
\end{align*}
\]

for each operator \( op \). Such a rule would mean that each term of type \( op(\vec{A}) \) has a transition with an internal action leading back to itself. Clearly, the presence or absence of such transitions is unimportant for the purpose of determining observation equivalence, and hence for any behaviour equivalence.

In analogy with Definition 14 we say that a set of operators \( op \) is asynchronous if the transitions over \( \mathcal{T}_{op,M} \) can be determined by an asynchronous set of rules, and that an operator is asynchronous if it is a member of an asynchronous set. One way to think about an asynchronous operator is as a large network where the arguments represent sub-networks. This large network can control the sub-networks only through the ports which are external to the subnetworks; in particular the large network cannot distinguish between internal actions and absence of actions in the subnetworks. Also, the larger network must always be able to idle when all subnetworks idle. For example, parallel and linking are asynchronous as is the three-way linking suggested in the beginning of this section. This fact can be seen immediately since the rules for these operators form asynchronous sets. In contrast, the singleton set containing the only rule for the operator \( ext \) above does not form an asynchronous set: it satisfies neither of the two conditions.

The main theorem in this section can now be stated. It holds regardless of the choice of behaviour equivalence.

**Theorem 3.** For any asynchronous operator \( op \) there is a set of modules \( M \) such that \( op \) is \( = \)-definable over \( M \).

**Proof (sketch).** Let \( op \) be an asynchronous operator of type \( L \rightarrow L \). The main idea is to derive a defining context for \( op \) from the set of rules for \( op \). This context consists of a “controller” \( C_{op} \) interlinked with the operands \( \vec{A} \) of \( op \) in the following way:

Intuitively, \( C_{op} \) encodes the rules of \( op \): it controls the transitions of the operands and generates the appropriate external actions. By choosing an appropriate \( M \) (with appropriate schematic transitions) we can use an instance of a module in \( M \) for \( C_{op} \). Unfortunately the linking cannot be achieved directly: \( L_k \) is not necessarily disjoint from \( L \), so the parallel composition of \( A_k \) and \( C_{op} \) may be undefined. We can however make the linking in an indirect way by enclosing the \( A_k \):s
in “relabelling contexts”; these contexts effectively act as port name relabellings and make sure that all involved port names are unique.

A variant of the main result can be obtained by restricting attention to finite-state operators:

Definition 16. An asynchronous operator is finite-state if it is in a finite asynchronous set of operators.

In other words, an operator is finite state if only finitely many auxiliary operators are needed in order to define its semantics. Notice that a de Simone set of rules for a finite set of operators is necessarily finite (since each operator has a fixed type), hence the semantics of a finite-state operator can be defined with a finite set of rules. We also say that a module $M$ is finite-state if its associated schematic transitions imply that any instance of it is finite-state.

Theorem 4. For any finite-state asynchronous operator $op$ there is a set of finite-state modules $M$ such that $op$ is $\simeq$-definable over $M$.

Proof. In the proof of Theorem 3, we only use modules in the definitions of the relabelling contexts (these will always be finite-state) and in $C_{op}$; and $C_{op}$ is finite-state if $op$ is finite-state.

This relates nicely with the result in Section 6: the modules SY, AR and AL are sufficient to define any finite-state term. Thus, if $M$ consists of finite-state modules, then any context over $M$ has an equivalent context over $\{SY, AR, AL\}$.

Corollary 5. Any finite-state operator is $\simeq$-definable over $\{SY, AR, AL\}$.

Say that two operators $op$ and $op'$ are $\simeq$-equivalent, written $op \simeq op'$, if they are of the same type $L \rightarrow L$, and for all $\bar{A}$ of sort $\bar{L}$ it holds $op(\bar{A}) \simeq op'(\bar{A})$. In essence the converses of Theorems 3 and 4 hold up to $\simeq$-equivalence of operators:

Theorem 6. If an operator $op$ is $\simeq$-definable over $M$ then there is an asynchronous $op^*$ such that $op \simeq op^*$. If $M$ additionally contains only finite-state modules then there is such an $op^*$ which is finite-state.

Proof (sketch). Assume that $op$ is $\simeq$-definable over $M$. Then there is a context $C_{op}$ over $M$ such that $C_{op}(\bar{A}) \simeq op(\bar{A})$ for all $\bar{A}$ of the appropriate sort. We can prove by structural induction on contexts that for each context $\bar{C}$ there is an asynchronous operator $op^*$ such that $\bar{C}$ is a $\simeq$-defining context for $op^*$. If the modules in $\bar{C}$ are finite state then $op^*$ will also be finite state. Put $op^* = op^*(\bar{A})$; we then have that $op^*(\bar{A}) \simeq C_{op}(\bar{A})$ holds for all $\bar{A}$ of the appropriate sort; this proves $op^* \simeq op$.

Notice that if $op$ is $\simeq$-equivalent with an asynchronous operator $op^*$ then $op$ is $\simeq$-definable even if $op$ is not asynchronous. This is demonstrates that $\simeq$-definability varies with the choice of $\simeq$.

It is interesting to consider some popular operators from other process algebras here. Table 3 summarises the $\simeq$-definability and $\tau$-definability. In this table “Yes” means that the operator is definable over any basis for finite-state terms, while “No” means that the operator is not definable over any basis for finite-state terms.

As can be seen, the operators fall into three groups. The first group consists of operators which are $\simeq$-definable, and hence also $\tau$-definable. The proof that these operators are definable follows from Theorem 4 by providing $\simeq$-equivalent finite-state asynchronous operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usual notation</th>
<th>$\simeq$</th>
<th>$\tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCS parallel</td>
<td>$A\parallel B$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CCS relabelling</td>
<td>$A[f]$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CCS restriction</td>
<td>$A\mid\alpha$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CCS nondeterministic choice</td>
<td>$A + B$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CCS prefixing</td>
<td>$a.A$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SCSS synchronous parallel</td>
<td>$A \times B$</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CSP parallel</td>
<td>$A\parallel B$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CSP hiding</td>
<td>$A\mid\alpha$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CSP external nondeterminism</td>
<td>$A\Box B$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CSP internal nondeterminism</td>
<td>$A \triangleright B$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CSP sequential composition</td>
<td>$A; B$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>LOTOS interrupt</td>
<td>$A\triangleright B$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MEUJE pilot</td>
<td>$a\cdot A$</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
The second group consists of \( =_\tau \)-definable but not \( \approx \)-definable operators. Again, the proofs of definability amount to giving \( =_\tau \)-equivalent finite-state asynchronous operators. The undefinability proofs are perhaps more interesting. As an example consider the CCS prefixing operator. This operator has the only rule

\[
a.A \rightarrow A
\]

The proof that prefixing is not \( \approx \) definable over any basis \( M \) for finite-state terms uses Theorem 6 and is a proof by contradiction as follows. Assume that there is an asynchronous operator \( \text{op}^* \) such that \( \text{op}^*(A) = a.A \) for all \( A \). Choose \( A \) such that \( \not\approx \) \( A \Rightarrow A' \) with \( a.A \not\approx a.A' \). Since all finite-state behaviours are definable such a term \( A \) must exist in \( \mathcal{F}_M \). Now since \( \text{op}^* \) is asynchronous and \( A \Rightarrow A' \) the idling rule of \( \text{op}^* \) yields \( \text{op}^*(A) \Rightarrow \text{op}^*(A') \). Then because \( \text{op}^*(A) \approx a.A \) there must be a term \( B \) such that \( a.A \Rightarrow B \) and \( \text{op}^*(A') \approx B \). We deduce from \( a.A \Rightarrow B \) and the only rule for prefixing that \( a.A = B \). Hence \( A \) and \( A' \) must satisfy \( a.A = \text{op}^*(A') \approx a.A' \). But this contradicts \( a.A \not\approx a.A' \). Hence no such \( \text{op}^* \) can exist.

Finally the third group consists of two operators which are not \( =_\tau \)-definable and hence not \( \approx \)-definable. For example, the MEJE pilot operator is not definable because internal actions \( (\emptyset) \) in its operand are significant for determining external actions of the operator (cf. the ext operator in this section). Thus, two terms which are trace equivalent but differ in their capability to perform internal actions may, when used as operands to the pilot operator, yield nonequivalent terms. So this operator can by Theorem 1 not have a defining context. The situation for SCCS synchronous parallel is similar.

8. Conclusion

We have explored an algebraic language with an operational semantics for description of networks of processes. The aim has been to keep to primitives as simple as possible, and then explore the expressive power as measured by the definable terms and operators. It has turned out that this language is expressive enough to cover the static parts, i.e. the operators normally used to combine processes in parallel, of many existing process algebras. Our conclusion is that the language is simple and expressive enough to throw light on fundamental properties of parallelism, and on other formalisms which describe parallelism.

There are many interesting questions left unanswered:

- Are there other interesting expressiveness results for terms? For example, is there a notion of "computable behaviour", and do the terms corresponding to computable behaviours have a proper basis?
- What is the definability of the operators in Table 3 with respect to some other equivalence, e.g. failure equivalence or testing equivalence?
- Is there a natural extension of our results to an algebra where the communications carry data values from one unit to another?

Much work in related process algebras has focused on complete axiomatisations of particular behaviour equivalences. Say that a statement \( A \approx B \) is valid if it is true in all operational interpretation of module symbols. In our companion paper [19] we formulate a complete axiomatisation for validity in this sense, and compare it with an axiomatisation of equality of networks. It turns out that validity coincides for a large class of behaviour equivalences, and that the complete axiomatisation is similar to the flow graph axiomatisation [11].

Acknowledgements

I am deeply grateful to Robin Milner, David Walker, Bengt Jonsson, Frits Vaandrager, and Tom Verhoeff for inspiring discussions on this and related subjects, and for critical readings of draft versions of this paper. Many illuminating comments, in particular related to Section 7, have been received from Kim Larsen.

References