Available online at www.sciencedirect.com
ScienceDirect

Electronic Notes in Theoretical Computer Science 212 (2008) 3–25

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Separation Logic Semantics for Communicating Processes

Tony Hoare

*Microsoft Research*

Peter O'Hearn

*Queen Mary, University of London*

**Abstract**

This paper explores a unification of the ideas of Concurrent Separation Logic with those of Communicating Sequential Processes. It extends separation logic by an operator for separation in time as well as separation in space. It extends CSP in the direction of the pi-calculus: dynamic change of alphabet is achieved by communication of channel names. Separation is exploited to ensure that each channel still has only two ends. For purposes of exploration, the model is the simplest possible, confined to traces without refusals. The treatment is sufficiently general to facilitate extensions by standard techniques for sharing multiplexed channels and heap state.

*Keywords:* Semantics of Programming Languages, Concurrency, Separation Logic, Communicating Sequential Processes

## 1 Introduction

This paper reports on work bringing together semantic ideas lying behind Concurrent Separation Logic (CSL, [18,4]) and Communicating Sequential Processes (CSP, [11]).

CSL provides a modular way of reasoning about shared-memory programs. It is based on the principle that that, at any time, it is possible to partition the state into that "owned" by separate processes. Ownership constrains the operations that processes are allowed to perform, and separation and ownership work together to allow independent reasoning about concurrent processes. In CSL the 'ownership" is dynamic, changing over time as heap objects are allocated, deallocated and transferred between processes.

CSP itself has a strong form of locality built in: a process has an associated alphabet, and it is only allowed to engage in events from the alphabet. There is thus a strong similarity to the CSL ownership idea. It is therefore natural to ask

doi:10.1016/j.entcs.2008.04.050

whether we can have a more dynamic model of ownership, as in CSL, for CSP-style message passing.

In this paper we take alphabets as the model of ownership, but allow them to change over time. This requires us to record the current alphabet before and after every event in a trace. Channel allocation has the effect of enlarging the alphabet while deallocation shrinks it. We also consider message passing primitives that can change the alphabet, by transferring channel permissions along with values.

The model that we present is a cousin of the standard trace semantics of CSP. The model includes two separating conjunctions: parallel composition is modelled by separation in space, and sequential composition by separation in time. The spatial separating conjunction is defined in a way that emulates the semantics of parallel composition in CSP, where synchronization is forced on common events in the (changing) alphabets of different processes. The spatial composition of alphabets ensures that only one process can own a channel end at any time. The sequential composition connective is similar to conjunctions ("chop" operators) at the basis of Interval Temporal Logic and Duration Calculus [16,10].

To develop these ideas we use an illustrative process language that borrows from several previous works. As in pi-calculus [15], we allow channels to be passed as the contents of messages and to be dynamically allocated. As in the occam language [14], we model point-to-point communication, where each channel has only one sender and one receiver at any given time. The sender and receiver for a given channel are, however, not fixed: permission to access a channel end can be transferred between processes, as is done in occam-pi [26]. Unusually for a process calculus, we allow dynamic deallocation as well as allocation of channels. This corresponds to the explicit channel-management capabilities used in systems programs, where the responsiblity for deallocating a channel is given to the processes that use it rather than a garbage collector. Our model applies equally well to programs and languages that never use deallocation, preferring to rely on garbage collection. But, even in a garbage-collected language, an explicit but non-executable delete may actually help to prove a program in a modular way: safe deallocation almost forces us to account for which processes are allowed to use what channels, and when.

The specific features in the illustrative language are chosen because of the way they mix together to make a somewhat simple mathematical model. Many variations could and should be considered. We would certainly like to consider many-to-many channels. And, we would like to mix program heap and channels; for example, we could directly model the situation where channels are held in linked lists representing queues of requests in a web server. Our mathematical definitions are phrased in a general way that potentially facilitates extension to account for these kinds of features.

In this paper we do not consider structures such as failures, divergences, or infinite traces, that have been used in the semantics of CSP to account for deadlock and liveness [23,3]. The extension to the these sorts of properties is a problem for future work.

The presentation that follows is designed to suit readers with some prior acquaintance either of process algebra or of separation logic; it enables them to gain acquaintance of the other. Section 2 starts with the general principle of separation logic, and its relation to interpretations of parallel composition. Section 3 then presents the basic ideas in our illustrative language and its semantics in an informal way, followed by the formal treatment in subsequent sections. The paper ends with a discussion of related and future work.

## 2 Separating Conjunctions and Process Semantics

This section summarizes the general background for the kind of semantic model we give. The notions here will be used in the construction of a particular model in the further sections of the paper.

### 2.1 Ternary Relation Models

Assume we are given a set $T$ and a ternary relation $S \subseteq T \times T \times T$. We can lift $S$ to a binary operation $\odot : \mathcal{P}(T) \times \mathcal{P}(T) \to \mathcal{P}(T)$ on the powerset of $T$ as follows:

$$t \in A \odot B \text{ iff } \exists u, v.\, Stuv \wedge u \in A \wedge v \in B.$$

We call $\odot$ the *separating conjunction induced by* $S$. In the concrete models later, $T$ will be a set of traces, and $Stuv$ will express that $t$ consists of two separate parts, namely $u$ and $v$.

The set $\mathcal{P}(T)$ of predicates is a complete Boolean algebra, and we will use standard logical notation ($\wedge$, $\vee$, $\neg$) to denote meet, join, and complement. $\odot$ is monotone in the subset order, and the induced functions $A \odot (\cdot)$ and $(\cdot) \odot A$ have right adjoints which give us the implication connectives corresponding to the separating conjunction. By virtue of being left adjoints, we obtain that each of these parametrized operations preserves all joins:

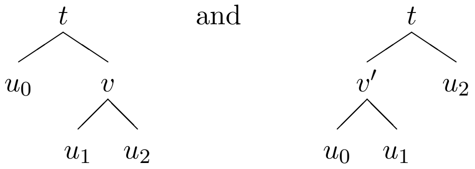
$$\bigvee_{B \in X} A \odot B \;=\; A \odot (\bigvee_{B \in X} B) \qquad \text{and} \qquad \bigvee_{B \in X} B \odot A \;=\; (\bigvee_{B \in X} B) \odot A$$

for $X$ a set of predicates.

We say that $S$ is

$$\begin{array}{ll} \textit{commutative} & \text{iff } Stuv \Rightarrow Stvu, \\ \textit{associative} & \text{iff } \exists v.\, Stu_0v \wedge Svu_1u_2 \Leftrightarrow \exists v'.\, Stv'u_2 \wedge Sv'u_0u_1. \end{array}$$

Associativity or commutativity of $S$ implies associativity or commutativity in the usual sense of $\odot$. The associativity condition can be pictured as asserting the equivalence of



where $v$ and $v'$ are existentially quantified in each tree, and where relation $Sabc$ is depicted as a tree with root $a$ and children $b$ and $c$.

If $S$ is such that $St_0uv \wedge St_1uv \Rightarrow t_0 = t_1$ then we say that $S$ is a deterministic model; otherwise it is nondeterministic. Deterministic models can be generated using the *function model construction*. Suppose we are given a partial function $\uplus : T \times T \rightharpoonup T$. This determines a ternary relation $S$ where $Stuv$ iff $t = u \uplus v$.

Conjunctions of the form given above were proposed by Routley and Meyer in their ternary-relation semantics of substructural logics [24]. Special cases, where the relation can be replaced by a (partial) function, have been used in Bunched Logic [19,21] and Duration Calculus [10]. On the other hand, such conjunctions can be seen as binary modal operators [5]. Like other modal operators, they are definable in predicate calculus by restricted forms of quantification over a designated parameter (a "possible world" semantics), a parameter that is usually left implicit in each predicate of the modal logic.

Using the definitions of this section we can begin to set down a structure of a semantics of processes. We presume we are given a set $T$, a commutative and associative ternary relation $S_*$ on $T$, and an associative relation $S_;$ on $T$. We use "$*$" and "$;$" to denote the separating conjunctions induced by $S_*$ and $S_;$. The "$*$" connective will be used in the trace semantics of concurrent processes, "$;$" will be used for sequential composition, and the disjunction $A \vee B$ will interpret nondeterministic choice.

### *2.2 Basic Examples*

A first example is given by sets with disjoint union. Here, $T$ is the powerset of some set $F$, and $Stuv$ holds just if $t = u \cup v$ and $u \cap v = \emptyset$. This is a simple version of the models in previous work on separation logic, and is both commutative and associative. This example is obtained via the function model construction using the union $\uplus$ of disjoint sets.

An associative but noncommutative example is obtained by taking $S$ to mean separation by concatenation of sequences. If $T = F^*$ is the set of sequences of elements from a set $F$, then $Stuv$ just if $t = u \frown v$. Again, this is a functional model (in fact, given by a *total* function). A nondeterministic (and commutative and associative) $S$ is given by interleaving of sequences: $Stuv$ just if $t$ is an interleaving of $u$ and $v$.

An example related to the trace semantics of parallel composition in CSP is as follows. Again let $T = F^*$, and if $X \subseteq F$ and $t \in T$ let $t{\restriction}X$ be $t$ restricted to elements from $X$. Then $Stuv$ means that there are $U$ and $V$ where $u = t{\restriction}U$ and $v = t{\restriction}V$. This $S$ is both commutative and associative, and is it nondeterministic. An example related to the trace semantics of sequential composition in CSP is given by taking $T = F^{*\checkmark}$ to be sequences possibly terminated with $\checkmark$. $\checkmark$ stands for successful termination, and distinguishes termination from deadlock. Then $Stuv$ means that $t = u$ and neither has $\checkmark$, or $u = u'\checkmark$ and $t = u' \frown v$. This $S$ is noncomutative, associative and deterministic.

# 3 Illustrative Language

In this section we describe the process language we will interpret. We use several examples to explain the constructs in the language, and their semantics, in an informal way, as preparation for the more formal treatment in subsequent sections.

## *3.1 Traces, Informally*

The intuitive model is that at any point in time a process has a current alphabet, consisting of the channel ends – $c?$, $c!$ – that it is allowed to use. We also say that $c?$ in an alphabet records receive *permission*, and $c!$ send permission. The alphabet will be allowed to change over time, as a result of allocation, deallocation, and message passing. To model the changing alphabet we intersperse alphabets, which confer ownership of channel ends, with events, which record communications.

A trace $t$ is a non-empty alternating sequence

$$\alpha_0 ... E_n \alpha_n$$

of alphabets and events, beginning and ending with an alphabet, or an alternating sequence

$$\alpha_0 ... E_n \alpha_n \checkmark$$

ending in a $\checkmark$, signifying termination.

An alphabet $\alpha$ is a finite set of channel ends $c!$, $c?$, where $c$ is drawn from an infinite collection of channels. An event (or rather, event set) $E$ is a finite set of primitive events, drawn from

- $c!m$, send of a message,
- $c?m$, receive of message.

The messages $m$ themselves have structure, consisting of a value $v$ and a permission $\rho$:

$$m ::= v\rho \qquad v ::= c \mid 3 \mid \cdots \qquad \rho ::= \epsilon \mid ! \mid ? \mid !?$$

Permissions in messages are used to indicate transfer of ownership, from sender to receiver. For instance, a message $c!$ indicates that ability to send, but not to receive, on $c$ is transferred from the sending to the receiving process, along with the value $c$. $c?$ sends receive permission, $c!?$ sends both permissions, and $c\epsilon$ sends the plain value without any permissions. (In most examples we will simply write $c$ rather than $c\epsilon$, eliding the empty permission.)

A singleton event $[c!m]$ or $[c?m]$ represents an *offered* communication, whereas a combined event $[c!m, c?m]$ represents a *consummated* communication. In contrast the the standard definition of CSP, consummated communications cannot be the subject of further synchronization with other processes. This is because we will model point-to-point communication.

This use of sets of primitive events, and particularly the respesentation of consummated communications, has been chosen because it leads to a particularly simple notion of event composition using set union. Also, our sets of primitive events represent several actions happening at the same time. For example, we can have offered communications on different channels, $[c!3, d!4]$, or even two simultaneous consummated communications, $[c!3, c?3, d!4, d?4]$.

### 3.2 Process Terms

The process terms we will interpret include the following

$$\begin{aligned} P &::= \texttt{SKIP} \mid \texttt{STOP} \mid P \parallel P \mid P + P \mid P; P \\ &\mid\;\; x!q\rho \mid x?(y\rho).P \mid \texttt{new}\, x.P \mid \texttt{dispose}\, x\; \cdots \\ q &::= 3 \mid x \mid \cdots \end{aligned}$$

We distinguish variables ($x$, $y$,...) from channels ($c$, $d$,...), and use an environment model where the semantics of a term is relative to a mapping $\eta$ from variables to channels (and other values such as numbers). Channels $c$ and $d$ are constants, like 3 and 5. The $\texttt{new}\, x.P$ construct extends the current alphabet with the ends $c?$ and $c!$ of a new channel, binds $c$ to $x$ in the environment, and then continues as $P$. So, $x$ is a bound variable in $\texttt{new}\, x.P$. Similarly, the form $x?(y\rho).P$ the variable $y$is bound in $P$. The nesting of ! and ? in messages will be used to express ownership transfer between processes.

The $\cdots$ in the grammar of processes is to indicate that our treatment will allow for the inclusion of additional constructs, such as recursion or equality-testing of channels, which can be interpreted in our model.

The ownership transfer primitives in our illustrative language follow a useful design pattern that has been built into occam-pi, where sending a channel end relinquishes ownership of the end. General CSL is more flexible, and can deal with algorithms in which the conventions about ownership transfer are under control of the programmer. For example, it could enable ownership of input and output ends to be exchanged on every communication, thus allowing (and requiring) an alternation of inputs and outputs on the same channel. occam dows not allow ownership transfer, and occam-pi requires all transfer of ownership to be signalled by an explicit communication.

### 3.3 Examples

We begin with a very simple example, the program

$\texttt{prog1} \;=\; x!3 \parallel (x?y.\, \texttt{SKIP}).$

Suppose that $x$ is bound to channel $c$ in the environment. Example traces for the left and right processes are

$$t_L \;=\; \{c!\} \;\; [c!3]\, \{c!\}\checkmark \quad \text{and} \quad t_R \;=\; \{c?\} \;\; [c?3]\, \{c?\}\checkmark\ .$$

The events (enclosed in $[\,]$) and alphabets (enclosed in $\{\}$) in these traces are disjoint at each step. The partial composition operator $* : T \times T \rightharpoonup T$ on traces simply unions up pointwise, giving us a trace

$$t_L * t_R \;=\; \{c!, c?\} \;\; [c!3, c?3]\, \{c!, c?\}\checkmark$$

of the parallel composition in `prog1`.

This example illustrates a crucial idea in our model: $*$ of traces is obtained by pointwise disjoint unioning. This operation is partial, in that $t*t'$ is undefined if the alphabets or events of $t$ and $t'$ overlap at any given step (or if they are of different lengths).

Our next illustration concerns allocation and deallocation. The traces of

$$\texttt{prog2} \;=\; \texttt{new}\,x.\,\texttt{dispose}\,x$$

include all traces

$$t(c) \;=\; \{\}\,[\,]\,\{c!, c?\}\,[\,]\,\{\}\checkmark$$

for any channel $c$. We have chosen to record only communication events in the traces. In particular, note that we do not regard allocation and deallocation as events: They are represented by their change of alphabet alone. In $t(c)$ we can see that `new` extends the alphabet with the two ends of a channel $c$ (also binding $c$ to $x$ in the environment), where `dispose` removes those channels from the alphabet. `dispose` $x$ must have both ends of the channel denoted by $x$ in its pre-alphabet in order to successfully execute.

It is instructive to consider what happens when we compose `prog2` with itself:

$$\texttt{prog2} \parallel \texttt{prog2}.$$

For a given channel $c$, both processes have trace $t(c)$. However, when we try to $*$-compose $t(c)$ with itself

$$\{\}\,[\,]\,\{c!, c?\}\,[\,]\,\{\}\checkmark \quad * \quad \{\}\,[\,]\,\{c!, c?\}\,[\,]\,\{\}\checkmark$$

we find that the alphabets are not disjoint after the first step. So, $t(c) * t(c)$ is undefined. On the other hand, if $c$ and $d$ are distinct then $t(c)$ and $t(d)$ will be disjoint at every step, and we can union them up to obtain a trace $t(c) * t(d)$ for the parallel composition:

$$t(c) * t(d) \;=\; \{\}\,[\,]\,\{c!, c!, d!, d?\}\,[\,]\,\{\}\checkmark$$

This example illustrates how the disjointness requirement of $*$ (and partiality of $t_1 * t_2$) is used to ensure that allocations done in different processes are consistent with one another.

Here is an example of ownership transfer by message passing:

$\texttt{prog3} \;=\; x!(y!);\ y?z.\,\texttt{SKIP}$

This program sends the write end of $y$ along channel $x$, and then receives a message on $y$. A possible trace for it is

$$\{y!, y?, x!\}\ [x!(y!)]\ \{y?, x!\}\ [y?3]\ \{y?, x!\}\checkmark$$

Notice how the $y!$ end has disappeared from the trace in the alphabet after the first communication: when a process sends a channel end in a message, it relinquishes ownership.

If we compose prog3 with a process that receives on $x?$ then the relinquishing of $y!$ by prog3 is matched by the other process aquiring it:

$x!(y!);\ y?z.\,\texttt{SKIP} \quad \| \quad x?(w!);\ w!3.$

A trace for the process on the right of $\|$ is

$$\{x?\}\ [x?(y!)]\ \{x?, y!\}\ [y!3]\ \{x?, y!\}\checkmark$$

and the alphabets and event sets of this trace are disjoint, at every step, from the corresponding events and alphabets in the previously-quoted trace for prog3: we can union these traces up using $*$, to obtain a trace for the parallel composition.

It can become tiresome to always mention the environment in examples, and this time we did not bother to. A more accurate description of the first trace for prog3 would be to say that there were channels $c$ and $d$ bound to $x$ and $y$ in the initial environment, that $z$ became bound to $d$, and that the final trace was

$$\{d!, d?, c!\}\ [c!(d!)]\ \{d?, c!\}\ [d?3]\ \{d?, c!\}\checkmark$$

The other trace would similarly mention $c$ and $d$, not variable names $x$ and $y$. We say this to emphasize that we use an environment mode instead of utilizing the technique of *scope extrusion* [15] in the semantics of processes that pass channels in messages.We thereby allow the logical possibility of *aliasing* where, for example, distinct variables denote the *same* channel, and separation of channels owned by different processes is captured using "$*$" rather than scoping (as illustrated in our example with prog2 $\|$ prog2 above).

A further principle is that you can only use a channel end that you own. So,

$\texttt{prog4} \;=\; x!(y!);\ y!3.\,\texttt{SKIP}$

never gets to do $y!3$, because it has given up the $y!$ channel end in the previous step.

Finally, it is the disjoint unioning of alphabets that implements the point-to-point channel notion. In a parallel composition, a channel end cannot be sent to both processes by a $*$-partitioning. For example, in the basic trace model for the program

$\texttt{prog5} \;=\; x!3 \;\|\; x!4$

we will not be able to do both communications.

We have given these examples to provide some intuition on the way that our trace semantics works. The informal presentation has (purposely) hidden some of the essential and yet intricate technical aspects of the model. Most importantly, we indicated that $t * t'$ is undefined when events or alphabets overlap; but there are other cases when it is undefined, which are needed to implement the "forcing synchronization" idea of CSP. These cases are examined formally in Section 4.3.

# 4 Traces, Formally

In this section we define the composition operators for traces. We will also identify a subset of the traces (the legal ones) that are needed in the construction of our model.

## *4.1 Composition Operators*

We have already said in Section 3.1 that a trace is an alternating sequence of alphabets and events (or event sets), possibly terminated with $\checkmark$. We say that trace $t$ is completed if it has $\checkmark$, and that it is incomplete, otherwise. We let $T$ denote the collection of all traces, and **completed** be the set of all $t \in T$ which have $\checkmark$.

With this data we can define the partial composition function $* : T \times T \rightharpoonup T$. For incomplete traces it is

$$(\alpha_0 ... E_n \alpha_n) * (\alpha'_0 ... E'_n \alpha'_n) \;=\; (\alpha_0 \uplus \alpha'_0) ... (E_n \uplus E'_n)(\alpha_n \uplus \alpha'_n)$$

where we understand that the left-hand-side is undefined if any of the $\uplus$ expressions on the right are. We stipulate that $t_1 * t_2$ is undefined if $t_1$ and $t_2$ do not have the same length. For completed traces, $t_1\checkmark * t_2\checkmark$ is $(t_1 * t_2)\checkmark$. $t_1 * t_2$ is undefined if one argument is completed and the other incomplete.

Using $\checkmark$, we can define the sequential composition operation $; : T \times T \rightharpoonup T$. $t\,;\,t' = t$ if $t$ is incomplete. If $t$ is completed we concatentate $t'$ onto $t$ after we remove the $\checkmark$ and last alphabet from $t$ taking care to ensure that the last alphabet of $t$ is the first alphabet of $t'$. That is, suppose $t = u_0\alpha\checkmark$ and $t' = \alpha' u_1$ and. Then $t; t' = u_0 \alpha u'_1$ if $\alpha = \alpha'$, and $t; t'$ is undefined otherwise.

By the function model construction, this gives us ternary relations $S_*$ and $S_;$ on $T$, and their induced conjunctions which we will denote "$*$" and "$;$". Context will always disambiguate where a use of "$*$" or "$;$" is as an operation on traces or on predicates.

## *4.2 Concurrency and Ownership*

Not all sets of primitive events are sensible. For example, a set $[d!3, d!4]$ indicates two concurrent sends on $d$, with different values, at the same time. Another, subtler, example is $[c!(d!), d!3]$ where 3 is sent on channel $d$, and concurrently the send permission for $d$ is sent on channel $c$; this second example conflicts with the point-to-point idea, that two processes cannot sumultaneously possess the send permission on a channel.

More generally, if $e$ and $e'$ are two events in an event set, we want it to be consistent for them to occur at the same time. We can formalize this notion of consistency by appealing to the notion $\mathsf{pre}(e)$ of the pre-alphabet for an event. The alphabet $\mathsf{pre}(e)$ describes the resources needed for the event $e$ to occur. Consistency of $e$ and $e'$ then means that $\mathsf{pre}(e)$ and $\mathsf{pre}(e')$ do not overlap; that is what is needed for the events to occur concurrently.

We establish some notation. If $m$ $(= v\rho)$ is a message then $\mathsf{res}(m)$ is an alphabet describing the permissions sent with the message value $v$, and this is used in the definition of $\mathsf{pre}(e)$:

$$\mathsf{res}(v\epsilon) = \{\} \qquad \mathsf{res}(v!?) = \{v!, v?\}$$
$$\mathsf{res}(v!) = \{v!\} \qquad \mathsf{res}(v?) = \{v?\}$$

$$\mathsf{pre}(c!m) = \{c!\} \cup \mathsf{res}(m) \qquad \mathsf{pre}(c?m) = \{c?\}.$$

Note that the$\mathsf{pre}$ of a send includes the resources of its message, while the $\mathsf{pre}$ of a receive does not; this reflects the direction in which the permissions travel.

The following is our restriction on event sets.

Concurrency Property. *An event set $E$ must satisfy*

$$\forall e, e' \in E.\ e \neq e' \Rightarrow \mathsf{pre}(e) \cap \mathsf{pre}(e') = \emptyset\ .$$

This condition could be phrased much more generally, for partial monoids of events and alphabets. Then, the condition says that if the composition of two events is defined, so is the composition of their pre's, where "pre" is a function from the event monoid to the alphabet monoid.

We impose a second condition which rules out traces in which an event occurs when it is not justified by the pre-alphabet.

Ownership Property: *any pair $\alpha E$ of a consecutive alphabet and event in a trace must satisfy*

$$\forall e \in E.\, \mathsf{pre}(e) \subseteq \alpha\ .$$

The Ownership and Concurrency properties together can be characterized equivalently by a proof system for allowed transitions $\alpha E$.

$$\frac{}{\{c?\}[c?m]} \qquad \frac{}{(\{c!\} \cup \mathsf{res}(m))[c!m]}$$

$$\frac{}{\alpha[\,]} \qquad \frac{\alpha_0 E_0 \quad \alpha_1 E_1}{(\alpha_1 \uplus \alpha_1)(E_1 \uplus E_2)}$$

As an example of a transition $\alpha E$ allowed by the Ownership Property is

$$\{c?, d!, e!\}[c?3, d!(e!)].$$

It is obtained by concurrently composing the two events, of a receive on $c$ and an ownership-sending send on $d$. It is important that the restrictions imposed by these properties are not exactly "unique occurrence of a channel end," as illustrated by

$$\{c?, d!\}[c?3, d!(d!)],$$

where there are two occurrences of $d!$ in the event. An example of a programming idiom utilizing this capability will be given in Section 5.2.

### *4.3 Forcing Synchronization*

Our final restriction constrains traces in a way which forces CSP-style synchronization.

> Synchronization Property: *any pair $\alpha E$ of a consecutive alphabet and event in a trace must satisfy*

$$\{c!, c?\} \subseteq \alpha \Rightarrow \ \forall m.(c!m \in E \Leftrightarrow c?m \in E).$$

The requirement says that all communications must be consummated when both ends of a channel are present. This is related to the point-to-point communication idea: when both ends of a channel are owned, no other process can synchronize with one of the ends.

To see the effect of the Synchronization Property, consider the traces

$$t_1 = \{c?\}\ [c?3]\ \{c?\}\quad [\,]\ \ \{c?\}$$

$$t_2 = \{c!\}\quad\ [\,]\ \ \{c!\}\ [c!3]\ \{c!\}.$$

These are traces for the left and right processes in $(c?y.\mathtt{SKIP}) \parallel (c!3\,;\,\mathtt{SKIP})$, but if we union together all the components the resulting trace

$$t = \{c!, c?\}\ [c?3]\ \{c!, c?\}\ [c!3]\ \{c!, c?\}$$

violates the Synchronization Property. For, if the first event has $c?3$ it must also have $c!3$, since the start-alphabet of $t$ has both $c!$ and $c?$. This illustrates how the Synchronization Property prevents communications from being ignored.

### *4.4 Legal Traces*

The three conditions we have given in this section have the effect of restricting the set of traces. To be explicit:

> Legal Traces. *We define the set $L$ of legal traces to be those $t$ where every event set $E$ satisfies the Concurrency Property and every transition $\alpha E$ in $t$ satisfies the Synchronization and Ownership Properties.*

The $*$ operation from Section 4.1 restricts to an operation $*_L : L \times L \rightharpoonup L$ as follows:

$$
\begin{aligned}
t *_L t' &= t * t' \qquad\quad \text{if } t * t' \in L \\
t *_L t' &= \text{undefined if } t * t' \text{ undefined or } t * t' \notin L
\end{aligned}
$$

; restricts to $;_L$ similarly.

We included these $L$ subscripts just to make clear that we are defining new binary operators, in terms of those defined previously. From now on, though, we will simply write $*$ and ;, without the subscripts, for these induced operations of type $L \times L \rightharpoonup L$.

### 4.5 Generalities

Suppose $S$ is a ternary relation on $T$ and $L \subseteq T$. Then the induced relation $S\restriction L \subseteq L \times L \times L$ is just the restriction of $S$ to $L$. In terms of the framework from Section 2, Section 4.1 presented a model determined by two ternary relations $S$ on $T$, and Section 4.2 presented further models gotten by applying this subset construction.

We warn that the relation $S\restriction L$ is not automatically associative if $S$ is: when we set down any such $S$ and we want an associative operation, we are obliged to prove associativity for the particular $S\restriction L$. ($S\restriction L$ is, however, commutative if $S$ is.) The particular choice of $L$ in this section was partly determined by this obligation. If we were to impose the Synchronization Property but not Ownership, then the induced $*_L$ (or its relation counterpart) is not associative. However, when we impose Ownership as well, associativity holds.

### 4.6 Other Models

Much of the work in this paper can be done in a setting where the sets of alphabets and events are replaced by arbitrary partial commutative monoids, with legality conditions represented by a subset $L$ of traces.

In a heap model we could replace alphabets by a set of partial functions $Heaps = L \rightharpoonup V$, where $\uplus$ of alphabets can be replaced by union of functions with disjoint domain. The elements $E \in Events$ can be taken to be sets of semaphore operations $P(s)$, $V(s)$ for a fixed collection of semphores $s$, with the *exclusion restriction* that we never have $\{P(s), V(s)\} \subseteq E$, two operations on a single semaphore in a single event set. The event composition $\uplus$ is union of disjoint sets. This model is similar to the one from [6], but allows operations on different semaphores to happen at the same time. A variation on this model, which allows concurrent read access to a location, is obtained using permission models [2].

We can combine heap and message passing models using a product $Heaps \times Alphabets$. If we keep $Events$ as in the communication model in this paper, then we could model a situation where each process has its own state, and only interacts with other processes by explicit message passing.

Finally, we could perhaps model many-to-many communication by considering an alphabet monoid which allows the composition of overlapping channel ends to be defined. To make this work with channel deallocation the best current technology is permissions [2].

# 5 The Traces of a Process

The semantics $\mathsf{traces}(P)$ of a process is a predicate denoting a set of traces. To be literal, it denotes a function from environments to trace sets. For simplicity, we will use logical notation in the usual semi-formal way to write down these predicates, relying on predicate calculus to do all environment manipulation (see later in the section for elaboration).

We can immediately give the semantics of several of our constructs.

$$\begin{aligned}
\mathsf{traces}(P \parallel Q) &= \mathsf{traces}(P) * \mathsf{traces}(Q)\\
\mathsf{traces}(P\,;\,Q) &= \mathsf{traces}(P)\ ;\ \mathsf{traces}(Q)\\
\mathsf{traces}(P + Q) &= \mathsf{traces}(P) \vee \mathsf{traces}(Q)\\
\mathsf{traces}(\mathtt{SKIP}) &= \mathbf{skip}\\
\mathsf{traces}(\mathtt{STOP}) &= \mathbf{skip} \wedge \neg\mathbf{completed}
\end{aligned}$$

Here, the predicate **skip** is the set consisting of all traces

$\alpha[\,]\cdots[\,]\alpha$ and $\alpha[\,]\cdots[\,]\alpha\checkmark$

where the same alphabet is repeated, interspersed with the empty event. This definition allows arbitrary stuttering (see below) and is also closed under prefixes.

For the remaining constructs we first make some preliminary definitions.

- $A^\dagger$ *is the prefix-stuttering closure of* $A$. That is, if $t \in A$ and $t'$ is a prefix of a trace obtained from $t$ by some number of stuttering steps, then $t' \in A^\dagger$. Formally, if $t = u\alpha u'$ then we say that $t' = u\alpha[\,]\alpha u'$ is obtained from $t$ by a stuttering step. We require that $t' \in T$ if $t \in T$. If $t$ is an initial subsequence of $t'$ then we we say that $t$ is a prefix of $t'$ and that $t'$ is an extension of $t$. If $t' \in T$ and $t$ is a prefix of $t'$ then we require that $t \in T$.
- $\mathbf{expand}[\,t\,]\ =\ \mathbf{skip}; (\{t\}^\dagger * \mathbf{skip})$.

These operations are related to healthiness conditions detailed in the next section. The trace semantics of processes will be closed under prefixes and stuffering. In a model with finite traces only, it is necessary to consider non-completed traces in order, for example, to account for the parallel composition of terminating and non-terminating processes. Closure under stuttering is a technical device used to make the lengths of traces match up when using $*$ to define parallel composition. The initial **skip** in the definition of expansion is there to ensure that the resulting predicate contains all singleton traces, and we then close up under prefixes, stuttering, and the tacking on of additional alphabet via $*$**skip**. The closure by $*$**skip** extends $t$ in a way that leaves additional resources unchanged, and is related to the

the issue of avoiding explicit frame axioms (which say what doesn't change); see Sections 6.1 and 6.3.

We can use this expansion notion to give the semantics of a construct by presenting just a single trace. As an example, consider that with

$$\textbf{expand}[\ \{x, x?\}\ []\ \{\}\checkmark\ ]$$

we get what we expect for the semantics of disposal, a trace set consisting of

- singleton traces that consist of any initial alphabet,
- traces that stutter from any initial alphabet and fail to terminate, and
- traces that stutter from an initial alphabet containing $\{x!, x?\}$, then delete $x!$ and $x?$, then possibly stutter some more and then possibly terminate.

Here are the semantic clauses for the channel constructs.

$$\begin{aligned}
\mathsf{traces}(\texttt{new}\,x.P) &= \exists x.\,\textbf{expand}[\ \{\}[]\{x!, x?\}\checkmark\ ]\,;\,\mathsf{traces}(P)\\
\mathsf{traces}(\texttt{dispose}\,x) &= \textbf{expand}[\ \{x!, x?\}[]\{\}\checkmark\ ]\\
\mathsf{traces}(x!p\rho) &= \text{if } x? \in \mathsf{res}(p\rho) \text{ then } \ \mathsf{traces}(\texttt{STOP})\\
&\phantom{=}\ \text{else}\quad \textbf{expand}[\ (\mathsf{pre}(x!p\rho)[x!p\rho](\{x!\} - \mathsf{res}(p\rho))\checkmark\ ]\\
\mathsf{traces}(x?(y\rho).P) &= \exists y.\,x? \notin \mathsf{res}(y\rho) \wedge\\
&\phantom{=}\ \textbf{expand}[\ \{x?\}[x?(y\rho)](\{x?\} \uplus \mathsf{res}(y\rho)\checkmark\ ]\,;\,\mathsf{traces}(P)
\end{aligned}$$

For `new` the trace allocates the two ends of a channel, and then continues as $P$. (Note that $x$ might occur in $\mathsf{traces}(P)$.) For `dispose` the semantics just removes the two ends, when they are present. The clauses for message send and receive use res and pre, which were defined in Section 4.2, and "−" is used for set difference.

The clause for output of a channel end has one subtlety, concerning message sends like $x!(x?)$. In such a case, we know that there can never be any possible receiver, because the sending process owns both ends of the channel. So, the semantics treats this special case as being necessarily unrequited. Similarly, the meaning of input treats as a special case the receiving of the input permission on the same channel.

On the other hand, a message send $x!(x!)$, where one sends write permission for a channel in a message along that very channel, is perfectly sensible, and is used in an example in Section 5.2.

Technical Note. The semantics just given induces a function

$$\mathsf{traces}(P) : \mathit{Environments} \to \mathcal{P}(L)$$

where an environment $\eta \in \mathit{Environments}$ is a function mapping ordinary variables $x$ to values $\eta(x)$, and $\mathcal{P}(L)$ is the set of sets of legal traces. We used logical notation above instead of passing environments around explicitly. For example, instead of writing $\lambda\eta.\,\mathsf{traces}(P)\eta * \mathsf{traces}(Q)\eta$ we wrote $\mathsf{traces}(P) * \mathsf{traces}(Q)$.

The most significant simplification in this form of presentation comes from the use of quantifiers. For instance, we wrote

$$\mathsf{traces}(\mathtt{new}\,x.P) = \exists x.\,\mathbf{expand}[\ \{\}[\,]\{x!, x?\}\checkmark\ ]\,;\,\mathsf{traces}(P)$$

in the semantics of `new`, rather than the more literal (and cumbersome)

$$\mathsf{traces}(\mathtt{new}\,x.P)\eta = \bigvee_c \mathbf{expand}[\ \{\}[\,]\{c!, c?\}\checkmark\ ]\,;\,\mathsf{traces}(P)(\eta \mid x{\longmapsto}c)$$

which is just a standard semantics of the predicate calculus notation. That is, the environment-free presentation of the semantics for `new` is relying on the ability of a complete boolean algebra to model quantification in the standard way. With these remarks, we expect that no confusion is likely to arise from our use of logical notation in this way.

### *5.1 Example: Forever Unrequited Communication*

The idea of the Synchronization Property from Section 4.3 is that when a process has a resource it will have it exclusively. So, when a process has both ends of a channel, and it offers a communication on that channel, it is impossible for any other process to reciprocate. The following example illustrates this point further:

$\mathtt{prog6}\ =\ \mathtt{new}\,x.x!3.$

This program is doubly-bad: it deadlocks and it leaks a channel. As in pi-calculus, the deadlock happens because the offered communication will be forever unrequited, but, unlike in pi-calculus, our semantics also says that it leaks. In pi-calculus `prog6` is bisimilar to 0 (here, `STOP`), the basic deadlocking process, an identification justified by the garbage-collecting nature of pi's channel management. Here, because we have explicit deallocation we do not depend on a garbage collector, and so distinguish the process from `STOP`.

In more detail, in our model `prog6` has a trace

$\{\}\,[\,]\,\{x!, x?\}$

which shows the allocation explicitly, but no trace extending this one that does anything other than stutter. In particular, if we try to extend the trace with the indicated communication

$\{\}\,[\,]\,\{x!, x?\}\,[x!3]\,\{x!, x?\}$

then the resulting extended trace is not legal: it violates the Synchronization Property.

In contrast, the traces of `STOP` never change the alphabet.

### *5.2 Example: Buffer with Explicit Channel Management*

A more advanced example shows a buffer that passes messages from a sender to a receiver. The processes observe the convention that end of transmission is indicated by sending the output permission for a channel along the channel itself; it is then the responsibility of the recipient of an "end transmission" message to dispose the channel (or not). We give the sender and buffer processes, but not the receiver.

$$\mathtt{new}(\mathit{left}).\ \mathtt{sender} \parallel \mathtt{buffer}$$

where

$$\begin{array}{rcl} \mathtt{buffer} & = & \mathit{left}?x.\ \mathit{right}!x.\ \mathtt{buffer} \\ & & +\ \mathit{left}?(y!).(\mathtt{dispose}\,(\mathit{left}) \parallel \mathit{right}!(\mathit{right}!)) \end{array}$$

and

$$\begin{array}{rcl} \mathtt{sender} & = & \text{“produce an } m\text{”};\ \mathit{left}!m.\ \mathtt{sender} \\ & & +\ \mathit{left}!(\mathit{left}!).\ \mathtt{SKIP} \end{array}$$

The sender sends a stream of messages on the *left* channel to the buffer, which copies them out to a *right* channel. The sender indicates that it is done by sending the write permission for *left* to the buffer, which then sends the *left* channel back to the channel manager. The buffer assumes responsibility for disposing the *left* channel, but hands off all responsibility for managing the *right* channel.

(The processes just given are recursive, and we have not given a semantics to recursion yet. We expect in any case that the reader can follow the example prior to such a semantics, which is discussed in Section 6.2.)

Obviously there are several possible alternative arrangements for who (buffer or sender or receiver) has final responsibility for the various channels, and this example has displayed just one of them. In different practical situations different choices can be and are made.

To show the semantics working, here are sample traces for the sender and buffer processes after the allocaton of *left* has been done.

$$\begin{array}{rcccccccccc} t_S & = & \{l!\} & [l!3] & \{l!\} & [\,] & \{l!\} & [l!(l!)] & \{\} & [\,] & \{\} \\ t_B & = & \{l?, r!, t!\} & [l?3] & \{l?, r!\} & [r!3] & \{l?, r!\} & [l?(l!)] & \{l!, l?, r!\} & [r!(r!)] & \{\} \end{array}$$

In showing the traces we have written $r$ for *right* and $l$ for *left*. These traces show when the ownership transfer happens, and their composition $t_S * t_B$ is a trace of the behaviour of the parallel composition of the sender and the buffer. In trace $t_B$, the final transition

$$\{l!, l?, r!\}\ [r!(r!)]\ \{\} \ = \ \{l!, l?\}[\,]\{\} \ * \ \{r!\}[r!(r!)]\{\}$$

is itself a $*$-composition, obtained from the parallel composition in the *buffer* process.

Finally, it is worth remarking that the buffer is programmed in a dangerous style. If the sender process happens to send the *left*! channel end, then everything will be alright. But, if the sender mistakenly sent another channel end, then the buffer would mistakenly attempt to dispose that channel. A safer, but less efficient way to program the buffer is

$$\begin{aligned}\texttt{buffer2} \;=\;& \mathit{left}?x.\,\mathit{right}!x.\,\texttt{buffer2}\\ &+\; \mathit{left}?(y!).\mathbf{if}\;(y=\mathit{left})\;\mathbf{then}\;(\texttt{dispose}\,\mathit{left}\;\|\;\mathit{right}!(\mathit{right}!))\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\;\texttt{SKIP}\end{aligned}$$

Here the buffer explicitly checks that the correct permission has been obtained before disposing. This would help protect from incorrectly disposing. But, with the given sender process, the unsafe buffer is safe enough.

### *5.3 Discusson: Channel Faults*

This discussion of safety of the buffer program brings up a limitation of the trace model in this paper. In the model we are using STOP as the receptacle of several kinds of error, including deadlock, divergence, and channel faults. The former two are standard for trace models; we comment on the third.

Informally, a channel fault occurs when a program attempts an operation not in its alphabet. The double disposal of a channel, $\texttt{dispose}(x);\texttt{dispose}(x)$, is the classic example (the fault is in the second step). It may be understood by analogy with a double `free()` in the C language, which is regarded as resulting in "undefined behaviour" and (depending on the compiler) may result in a segmentation fault. Another example is the parallel composition $x!3 \parallel x!4$: if $\parallel$ splits alphabets, then one or the other of the parallel processes would attempt a communication outside its alphabet.

In the traces model faulting is represented by STOP, as illustrated by the equivalence

$$\texttt{dispose}(x);\texttt{dispose}(x) \;=\; \texttt{dispose}(x);\texttt{STOP}.$$

Also, we have the equivalence

$$x!3 \parallel x!4 \;=\; (x!3 + x!4);\ \texttt{STOP}$$

where we would like to say that the left process is has a fault (a race condition; recall our point-to-point assumption) where the right does not.

The basic problem is that, since $\mathsf{traces}(P + \texttt{STOP}) = \mathsf{traces}(P)$, it is impossible to use the trace model to say when errors represented by STOP will be avoided. In CSP, refined models have been used to distinguish the different kinds of error represented together by STOP in the basic trace semantics. The divergences model separates out infinite looping with no external communication, and the failures model separates out deadlock. Similarly, we would like a model that lets us prove that channel faults are avoided.

# 6 Foundational Properties of the Model

In this section we describe some basic theoretical properties enjoyed by the semantics. Generally, these properties will correspond to *healthiness conditions*, which identify conditions on predicates preserved by the semantics of process terms. We also formulate a result on *footprints*, which formalizes some of our intuitions about resources.

## *6.1 Healthiness*

The trace semantics obeys the following three healthiness conditions: for a set $A \in \mathcal{P}(L)$,

$$\begin{aligned}\textbf{Unity}:&\quad \textbf{skip}; A; \textbf{skip} = \textbf{skip} * A \ = \ A\\ \textbf{StutPref}:&\quad A = A^{\dagger}\\ \textbf{Consistency}:&\quad A \neq \emptyset\ .\end{aligned}$$

**Proposition 6.1** $\mathsf{traces}(P)$ *satisfies* **Unity***,* **StutPref** *and* **Consistency***.*

In fact, there is a stronger result than what is stated. Each construct in the language preserves all three conditions, and so this result is robust under addition of new constructs to the language of process terms.

We give the validation of **Unity** in a few cases of the trace semantics. For $\|$ we calculate

$$\mathsf{traces}(P) * \mathsf{traces}(Q) \ = \ \mathsf{traces}(P) * \mathsf{traces}(Q) * \textbf{skip} \ = \ \mathsf{traces}(P \parallel Q) * \textbf{skip}$$

The proof for ";" is similar, and the one for +, use that $*$ preserves joins.

$$(\mathsf{traces}(P) * \textbf{skip}) \vee (\mathsf{traces}(Q) * \textbf{skip}) \ = \ \mathsf{traces}(P) \vee \mathsf{traces}(Q).$$

The reason for condition **StutPref** was discussed in the previous section. **Consistency** is a basic condition which (together with **Unity**) implies a property reminiscent of the CSP trace model, where it is required that that the empty trace is included in the denotation of any process. Here, the singleton traces consisting of alphabets $\alpha$ play the role of the empty trace in CSP. We could have given inclusion of all singleton traces as this healthiness condition, but in the presence of **Unity** that is equivalent to the simpler condition **Consistency**.

We give a fuller discussion of **Unity**. It is necessary for the expected equivalences

$$(U) \qquad \mathsf{traces}(P) \ = \ \mathsf{traces}(P \parallel \mathtt{SKIP}) \ = \ \mathsf{traces}(\mathtt{SKIP}; P) \ = \ \mathsf{traces}(P; \mathtt{SKIP})$$

The reason is that **skip** is not the unit of ";" or "$*$" for arbitrary predicates. For counterexamples, consider a set $\{t\}$ consisting of any single trace (say, $t = \{c!\}[\,]\{c!\}\checkmark$). Then, $\{t\} * \textbf{skip}$ has infinitely many traces, not only one, and so is not equal to $\{t\}$. Similarly, $\{t\} \neq \textbf{skip}; \{t\}$ and $\{t\} \neq \{t\}; \textbf{skip}$.

The separating conjunctions do have units for arbitrary (non-healthy) predicates. The unit of $*$ is the set $\mathsf{emp}$ consisting of all traces whose alphabet and event components are everywhere $\{\}$ and $[\,]$. The unit of ; is set $\mathsf{step}$ of completed traces $\alpha\checkmark$ of length two. Neither of these predicates is denotable by a process in our illustrative language.

The **Unity** condition is obtained from a standard method for building a monoid from a semigroup by choosing an idempotent element. Specifically, we know that $\mathbf{skip} \in \mathcal{P}(L)$ is idempotent wrt $*$ and ;

$$\mathbf{skip}; \mathbf{skip} \;=\; \mathbf{skip} * \mathbf{skip} \;= \mathbf{skip}$$

It follows that if $A$ satisfies **Unity** we have that **skip** functions as a left and right unit for both "$*$" and ";", so we immediately obtain our equivalences $(U)$.

We should argue that the **Unity** restriction is reasonable. First, consider $\mathbf{skip}; A = A = A; \mathbf{skip}$. If processes are allowed to stutter and be closed under prefixes, as we want, then it makes no difference to add stuttering on either end.

Second, consider $A = A * \mathbf{skip}$. The construction $A * \mathbf{skip}$ in a sense "expands the reach" of $A$, where any trace

$$\alpha_0 ... E_n \alpha_n \in A \quad \text{or} \quad \alpha_0 ... E_n \alpha_n \checkmark \in A$$

is extended so that any completely separate alphabet $\alpha$ comes along for the ride:

$$(\alpha_0 \uplus \alpha) ... E_n (\alpha_n \uplus \alpha) \in A \quad \text{or} \quad (\alpha_0 \uplus \alpha) ... E_n (\alpha_n \uplus \alpha)\checkmark \in A.$$

All of the events $E_i$ remain the same in the extended trace, because the events in **skip** are the unit $[\,]$ of the event composition $\uplus$. Thus, from healthiness condition **Unity** we automatically obtain a sense in which processes behave locally: computation on a 'small" amount of resource can be automatically extended to larger portions of resource: as a consequence, we do not have to think about the entire global state of a system to understand a computation. This is related to the intuition behind the frame rule of [17].

Technical Note. When you "extend with a separate alphabet" you must be sure to satisfy the Synchronization Property. For example, $\{c!\}[c!3]\{c!\} \;*\; \{c?\}[\,]\{c?\}$ is undefined, because the trace $\{c!, c?\}[c!3]\{c!, c?\}$ is not legal. Thus, this illegal trace is *not* in the trace set of $\{\{c!\}[c!3]\{c!\}\} * \mathbf{skip}$. We state this because, if we were more literally following the frame rule from [17], we might have expected the closure property (written as a rule)

$$\frac{\{c!\}[c!3]\{c!\} \in \mathsf{traces}(P)}{\{c!, c?\}[c!3]\{c!, c?\} \in \mathsf{traces}(P)}$$

but this inference is blocked by the Synchronization Property.

### *6.2 Recursion and CPO Structure*

We will not give an explicit semantics of recursion in this paper, but we note that the semantics has properties sufficient for it to be defined via fixed-points.

**Proposition 6.2** *The set of trace sets satisfying the three healthiness conditions is a complete lattice under the subset order, with lubs being calculated by union, with* traces(STOP) *being the least element and the set $L$ of legal traces the greatest. The trace semantics of each construct is Scott-continuous in its process arguments.*

Note that Scott-continuity of $\|$ and ; is immediate from the join-preservation property of separating conjunction quoted in Section 2.1.

This result gives us enough information to calculate the semantics of recursion using either least or greatest fixed-points. Both have their uses. The least fixed-point corresponds to a partial-correctness semantics, where the greatest fixed-point corresponds to a specification-oriented semantics in which the order of reverse subset is regarded as an ordering of refinement.

Finally, the **expand**$[\,t\,]$ construction used in the previous section is connected to healthiness as follows.

**Proposition 6.3** **expand**$[\,t\,]$ *is the smallest set containing $t$ that satisfies the three healthiness conditions.*

### *6.3 Footprints*

Above, we remarked how the $A * \textbf{skip}$ part of condition **Unity** is related to the frame rule of Separation Logic. We can take this locality idea one step further, by revisiting the footprint idea [17]. Say that the footprint of a program execution is the set of resources touched by it, including those that are allocated. Extrapolating from this, we can talk about a footprint execution or trace, which mentions only those elements in its alphabets that are strictly needed for the execution to take place. For example,

$$t \;=\; \{z!, x!, x?, y!, y?\}\,[\,]\,\{z!, x!, x?\}\,[\,]\,\{z!\}$$

is a trace of `dispose` $y$; `dispose` $x$ but not a footprint trace because of the redundant $z!$, which is not accessed. Deleting $z!$ at every step gives us a footprint trace

$$t_f \;=\; \{x!, x?, y!, y?\}\,[\,]\,\{x!, x?\}\,[\,]\,\{\}$$

of `dispose` $y$; `dispose` $x$.

The relationship between these two traces is that $t \in \{t_f\} * \textbf{skip}$. We can generalize from this to formalize the idea is that we only need be concerned with traces that mention the resources that are accessed as computation progresses; there should be a smallest amount of relevant resource.

> Footprint of a predicate: *The footprint* foot$(A)$ *of a predicate $A$ is the smallest set $X$, if such a smallest set exists, where $X * \textbf{skip} \;=\; A$.*

A predicate is thus obviously completely determined by its footprint, when it exists.

We can show that footprints always exist by giving a characterization in terms minimal resource required. Consider the order where $t \sqsubseteq t'$ means that $t' \in \{t\} * \textbf{skip}$.

**Proposition 6.4** *If $A = A * \mathbf{skip}$, then the footprint of $A$ its subset of $\sqsubseteq$-minimal traces:*

$$\mathsf{foot}(A) \;=\; \{t_f \in A \mid t \sqsubseteq t_f \wedge t \in A \textit{ implies } t_f \sqsubseteq t\}.$$

The proof of the Proposition uses the fact that $\sqsubseteq$ has no infinite descending sequences, from which it follows that any trace $t \in A$ must have at least one minimal trace $t_f \in A$ below it. Since $\mathsf{traces}(P)$ satisfies **Unity** the antecedent in the Proposition is satisfied, and we conclude that the footprint of $\mathsf{traces}(P)$ exists.

Technical Note. For arbitrary partial commutative monoids in place of alphabets, $\sqsubseteq$ might have infinite descending sequences, in which case Proposition 6.4 would fail. Fractional permissions [2] provide one such model. See [22] for further information on the theory of footprints (where the term "footprint" is used in a related, but inequivalent, way).

## 7 Related Work

In the Introduction we acknowledged the influence of prior work on pi-calculus, occam and occam-pi, as well as CSL and CSP. In this section we discuss two bodies of closely related work on substructural typing and logic.

There has been a significant body of work on type disciplines that capture constraints on channel usage in pi-calculus. Linear type systems [13] ensure point-to-point channels and more: that each channel is used at most once. Session types are closer to the approach here, in that they ensure point-to-point channels but allow a channel to be reused multiple times while maintaining that there is at most one sender and receiver [12,25]. The type systems have been used to characterize special classes of behaviour, such as sequential behaviour [1], and are now being used in work on web services [7].

Our use of partial alphabet composition is similar to how substructural typing limits the number of processes that can access a channel. Beyond that basic similarity, the techniques developed here and in the work on types for pi-calculus appear to be complementary. Indeed, it would be conceivable to use ideas like in session typing to underapproximate safe states for our illustrative language (cf, Section 5.3). Conversely, it might be possible to employ techniques like those developed here to provide denotational models of session typing systems, where the changing alphabet is an explicit part of the semantics.

A number of authors have used substructural logics to reason about process calculi [9,8,20]. The approach has been to first set down an operational semantics of a process calculus, and then use the parallel composition to define a separating conjunction connective as described in Section 2. The approach in this paper is in a sense inverse. We first set down a ternary relation model or models, and then use the induced separation connectives in the description of the denotational semantics of processes. So, we use separation conjunctions to provide the semantics of process terms, where [9,8,20] use process terms to provide the semantics of separating conjunctions (in the generalized sense of Section 2).

Although our approach is inverse, we share a long-term aim with these works:

We would like to obtain tractable specification and proof methods for processes. Here we have set down a model, but we have not yet formulated explicit proof rules that could be used in a verification system for processes.

# 8 Concluding Remarks

This paper has been an experiment in model construction, where we are aiming at models of communicating processes where all possible uses of resources are explicitly circumscribed. The general hope is that models of circumscribed resources can lead to modular and tractable methods of reasoning about concurrent processes.

We carried out our study by marrying some of the ideas in CSL and CSP, two formalisms which have led to modular reasoning methods in different arenas. We described the semantics of a message-passing language with dynamic allocation and deallocation of channels, where the trace semantics of parallel composition uses a composition operation on traces that partitions channel ends between processes. Results were given on the footprint of a process, expressing a sense in which the model accounts for resources locally.

Although some steps have been taken, the ideas reported here should be considered preliminary in nature. There are two particular limitations that we highlight.

First is the basic problem that the plain trace model does not address certain kinds of error, such as deadlock and divergence. For our illustrative language there is a further kind of error, channel faults (Section 5.3), which again are not addressed by the trace model. The treatment of channel faults is perhaps the most important immediate problem.

Second, the process language we used is restrictive, and there is need to model a fuller range of concurrency mechanisms (based on shared memory, on many-to-many channels, etc). In fact, we have used the term "illustrative language" to emphasize that the language itself is not the important target of your study; rather, the model is. We have described the model in a general way that allows for variations, where one could swap different monoids for the alphabets and the events, and we have made suggestive remarks on particular variations in Section 4.6. The study of such alternatives is a topic for future work.

Acknowledgements. We thank Philippa Gardner, Kohei Honda, Alan Mycroft, Robin Milner, John Reynolds and Hongseok Yang for comments and discussions. O'Hearn acknowledges the support of the EPSRC and a Royal Society Wolfson Research Merit Award.

# References

[1] M. Berger, K. Honda, and N. Yoshida. Sequentiality and the pi-calculus. In *TLCA*, pages 29–45, 2001.

[2] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *32nd POPL*, pp59–70, 2005.

[3] S. D. Brookes. Traces, pomsets, fairness and full abstraction for communicating processes. In *13th CONCUR*, pages466-482, 2002.

[4] S. D. Brookes. A semantics of concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp16-34).

[5] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In *34th POPL*, pages 123–134, 2007.

[6] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd LICS*, pp366-378, 2007.

[7] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *16th ESOP*, pages 2–17, 2007.

[8] L. Cardelli and L Caires. A spatial logic for concurrency. In *4th International Symposium on Theoretical Aspects of Computer Science*, LNCS 2255:1–37, Springer, 2001.

[9] L. Cardelli and A. D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In *27th POPL*, 2000.

[10] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Inform. Proc. Letters*, 40(5):269-276, 1991.

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

[12] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th ESOP*, pages 122–138, 1998.

[13] N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. *ACM TOPLAS 21(5)* pp914-947, 1999.

[14] Inmos Limited. occam 2.1 reference manual. Technical Report, Inmos Limited, May 1995.

[15] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[16] B. Moszkowski and Z. Manna. Reasoning in interval temporal logic. Proceedings of the Workshop on Logics of Programs, volume 164 of LNCS, pages 371382, 1983.

[17] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*, pp1-19, 2001.

[18] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp49-67).

[19] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.

[20] D. Pym and C. Tofts. A calculus and logic of resources and processes. *Formal Asp. of Comput.*, 18(4):495–517, 2006.

[21] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications.* Applied Logic Series. Kluwer Academic Publishers, 2002.

[22] M. Raza and P. Gardner. Footprints in local reasoning. In 11th FOSSACS, 2008.

[23] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[24] R. Routley and R.K. Meyer. The semantics of entailment. pp. 194–243 in Truth, Syntax, and Modality, H. Leblanc, ed., Amsterdam: North-Holland., 1973.

[25] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *6th PARLE*, pages 398–413, 1994. Springer LNCS 817.

[26] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *LNCS*, pages 175–210. Springer Verlag, April 2005.