A Type-theoretic Approach to Deadlock-freedom of Asynchronous Systems

Samson Abramsky[1], Simon Gay[2], and Rajagopal Nagarajan[3]

[1] Department of Computer Science, University of Edinburgh, Mayfield Road, Edinburgh, UK, EH9 3JZ; samson@dcs.ed.ac.uk
[2] Department of Computer Science, Royal Holloway, University of London, Egham, Surrey, UK, TW20 0EX; S.Gay@dcs.rhbnc.ac.uk
[3] Department of Computing, Imperial College, 180 Queen's Gate, London, UK, SW7 2BZ
&
Electronics Research Laboratory, University of California, Berkeley, CA 94720, USA; R.Nagarajan@doc.ic.ac.uk

Abstract. We present a type-based technique for the verification of deadlock-freedom in asynchronous concurrent systems. Our general approach is to start with a simple interaction category, in which objects are types containing safety specifications and morphisms are processes. We then use a specification structure to add information to the types so that they specify stronger properties. In this paper the starting point is the category $\mathcal{ASP}roc$ and the extra type information concerns deadlock-freedom. In the resulting category $\mathcal{ASP}roc_D$, combining well-typed processes preserves deadlock-freedom. It is also possible to accommodate non-compositional methods within the same framework. The systems we consider are asynchronous, hence issues of divergence become significant; our approach incorporates an elegant treatment of both divergence and successful termination. As an example, we use our methods to verify the deadlock-freedom of an implementation of the alternating-bit protocol.

1 Introduction

In the realm of sequential programming, type systems allow the programmer to express constraints on the combinability of program modules. Compile-time type-checking, and particularly type-inference in the case of functional languages, is a valuable aid to the construction of correct programs. When distributed software is considered, the issue of ensuring compatible combination of subsystems becomes more complex. Interaction between components may take the form of a prolonged, dynamic pattern of communication rather than a simple procedure call, and both parties must agree about the expected nature of the dialogue. Our recent research on interaction categories [1, 2] has led to a framework for the design of sophisticated type systems which are able to specify communication protocols of this form. In this framework, type-checking and

verification are seen as different facets of a single activity and this opens up the possibility of developing a Propositions-as-Types approach to the verification of concurrent programs.

An interaction category is a semantic universe in which the objects are types, incorporating specifications of various kinds, and the morphisms are concurrent processes satisfying those specifications. A process interacts with its environment via an interface consisting of a number of typed ports. The structure of the category allows ports to be combined in various ways, and the shape of interfaces to be described. At the moment, only static connection topologies can be described; none of our work on interaction categories has yet tackled the issue of mobility. We hope to address this in future work.

Our approach to the construction of type systems for concurrency is to begin with a simple interaction category such as $\mathcal{ASP}roc$ [2, 3], in which the types correspond to very basic safety specifications, and use a *specification structure* [2] to add information to the types so that they specify stronger properties. This process can be iterated, leading to a "tower of categories" sharing a great deal of structure and possessing progressively more complex types.

In this paper we are interested in specifying deadlock-freedom. Deadlocks arise from communication failures, in which a message is not understood by its recipient, and from situations in which all of the subcomponents of a system are waiting for one another. The first of these possibilities may not be classically viewed as a deadlock, but it turns out that our theory naturally includes it. We construct a tower of three categories, beginning with $\mathcal{ASP}roc$ (whose definition is reviewed in Section 2). The next level up is the category $\mathcal{ASP}roc_V$, whose types incorporate a notion of *valid maximal behaviour* which is used to address the issues of divergence and successful termination. The third level is $\mathcal{ASP}roc_D$, in which well-typed processes can always be combined without fear of introducing deadlocks. However, the type structure of $\mathcal{ASP}roc_D$ does not support the formation of cyclic process configurations; to deal with such cases, we introduce an additional verification rule.

After presenting the necessary theory, we illustrate the use of our type system by analysing a version of the alternating-bit protocol [19] and verifying that it is deadlock-free. Communications protocols are a standard source of examples and case studies of verification techniques, so we see this as a significant choice of application.

The emphasis of the present paper is semantic; $\mathcal{ASP}roc_D$ provides a semantic framework for reasoning about communication behaviour. Although the categorical structure supports a number of inference rules for combining typed processes, we do not present a full type-checking system for syntactic processes. It is this semantic, denotational emphasis which distinguishes our approach from other recent work, for example Kobayashi's type system for a π-calculus-like language [15]. Elsewhere we have studied process calculi whose type systems correspond to the structure of various interaction categories: the synchronous

category $\mathcal{SP}roc$ whose types are safety specifications [12]; the synchronous category $\mathcal{SP}roc_D$ whose types specify deadlock-freedom [6]; and the asynchronous category $\mathcal{ASP}roc$ whose types are safety specifications [20]. Future work will combine the ideas of these calculi into a syntax to accompany the semantic type system of the present paper.

Much of our previous work has concentrated on deadlock-freedom of synchronous systems [1, 11]. In the asynchronous case, the theory is complicated by the need to consider issues of *divergence*. A divergent process is one which communicates internally forever, and under the process equivalence we use this is equated to a deadlocked process. Previous attempts to extend our work to the asynchronous case [2, 11] have used the idea of *fairness*, by assuming that processes are fair in the sense that every infinite behaviour has an infinite projection onto every port. There are several technical difficulties with this approach, which we describe in detail in Section 4.1, and the resulting theory is not completely satisfactory. The theory described in the present paper gives a much cleaner account of deadlock-freedom in asynchronous interaction categories.

2 The Interaction Category $\mathcal{ASP}roc$

In this section we briefly review those parts of the definition of $\mathcal{ASP}roc$ which are relevant to the present paper. This includes the $*$-autonomous structure [5], corresponding to the multiplicative connectives $\otimes$, $\parr$ and $(-)^{\perp}$ of linear logic [13], but not the additive structure (products and coproducts) or the delay operator. Complete definitions can be found in previous publications [1, 2, 3, 11]. Our use of $\mathcal{ASP}roc$ in the present paper allows us to analyse *asynchronous* systems, in which different components are able to evolve at their own rate. This is in contrast to much of our previous work on interaction categories (for example, [12]) which assumes universal synchronisation with respect to a global clock.

An object of $\mathcal{ASP}roc$ is a triple $A = (\Sigma_A, \tau_A, S_A)$, in which Σ_A is a set of actions and $\tau_A \in \Sigma_A$ is the *silent action*. Defining the observable actions by $\mathsf{ObAct}(A) \stackrel{\text{def}}{=} \Sigma_A - \{\tau_A\}$, $S_A \subseteq^{nepref} \mathsf{ObAct}(A)^*$ (a non-empty, prefix-closed set of observable traces) is a safety specification.

A *process* with *sort* Σ and *silent action* $\tau \in \Sigma$ is a labelled transition system (strictly, a distinguished state of a labelled transition system) with label set Σ. We identify processes up to observation equivalence [19]. Throughout the paper we will omit the verifications that the operations we define respect observation equivalence.

A *process* P *of type* A, written $P : A$, is a process P with sort Σ_A and silent action τ_A such that $\mathsf{obtraces}(P) \subseteq S_A$, where $\mathsf{obtraces}$ is given by the following coinductive definition.

$$\mathsf{allobtraces}(P) \stackrel{\text{def}}{=} \{\varepsilon\} \cup \{as \mid P \stackrel{a}{\Longrightarrow} Q, s \in \mathsf{allobtraces}(Q)\}$$

$$\mathsf{obtraces}(P) \stackrel{\text{def}}{=} \{s \in \mathsf{allobtraces}(P) \mid s \text{ is finite}\}$$

The morphisms of $\mathcal{ASP}roc$ are defined via the object part of the $*$-autonomous structure. Given objects A and B, the object $A \otimes B$ has

$$\Sigma_{A\otimes B} \stackrel{\text{def}}{=} \Sigma_A \times \Sigma_B \qquad \tau_{A\otimes B} \stackrel{\text{def}}{=} (\tau_A, \tau_B)$$
$$S_{A\otimes B} \stackrel{\text{def}}{=} \{s \in \mathsf{ObAct}(\Sigma_{A\otimes B})^* \mid s\restriction A \in S_A, s\restriction B \in S_B\}$$

where, for $\alpha \in \mathsf{ObAct}(\Sigma_{A\otimes B})$, $\alpha\restriction A \stackrel{\text{def}}{=} \begin{cases} \mathsf{fst}(\alpha) & \text{if } \mathsf{fst}(\alpha) \neq \tau_A \\ \varepsilon & \text{otherwise} \end{cases}$
and for $s \in \mathsf{ObAct}(\Sigma_{A\otimes B})^*$, $s\restriction A$ is obtained by concatenating the individual $\alpha\restriction A$. The projection $\sigma\restriction B$ is defined similarly. Notice that taking $\tau_{A\otimes B} = (\tau_A, \tau_B)$ means that a process with several ports delays by simultaneously delaying in its individual ports. Also, it is possible for a process to perform observable actions simultaneously in several ports; this is a contrast to traditional process calculi.

The duality is trivial on objects: $A^\perp \stackrel{\text{def}}{=} A$. This means that the types of $\mathcal{ASP}roc$ do not distinguish between input and output; later in the paper, this distinction will be recovered.

A morphism $p : A \to B$ of $\mathcal{ASP}roc$ is a process p such that $p : A \multimap B$. In general, $A \multimap B \stackrel{\text{def}}{=} (A \otimes B^\perp)^\perp$. In $\mathcal{ASP}roc$, the fact that $(-)^\perp$ is trivial means that $A \multimap B = A \otimes B$.

If $p : A \to B$ and $q : B \to C$, then the composite $p\,;q : A \to C$ is defined by labelled transitions.

$$\frac{p \xrightarrow{(a,\tau_B)} p'}{p\,;q \xrightarrow{(a,\tau_C)} p'\,;q} \qquad \frac{q \xrightarrow{(\tau_B,c)} q'}{p\,;q \xrightarrow{(\tau_A,c)} p\,;q'} \qquad \frac{p \xrightarrow{(a,b)} p' \qquad q \xrightarrow{(b,c)} q'}{p\,;q \xrightarrow{(a,c)} p'\,;q'}$$

The first two rules allow either process to make a transition independently, if no communication is required. The third rule allows the processes to communicate by performing the same action in the port B. Any of the actions a, b, c can be τ; if $b = \tau_B$ in the third rule, then two simultaneous independent transitions are made.

It is easy to see that if $f : A \to B$ and $g : B \to C$, then $f\,;g$ satisfies the safety specification necessary to be a morphism $A \to C$.

Although $\mathcal{ASP}roc$ is a category of asynchronous processes, the identity morphisms are *synchronous* buffers. $\mathsf{id}_A : A \to A$ instantaneously transmits any data received in its left port to its right port, and vice versa. It is defined by

$$\frac{a \in S_A}{\mathsf{id}_A \xrightarrow{(a,a)} \mathsf{id}_{A/a}.}$$

where the type A/a is defined by $\Sigma_{A/a} = \Sigma_A$, $\tau_{A/a} = \tau_A$ and $S_{A/a} = \{s \mid as \in S_A\}$.

If $p : A \to C$ and $q : B \to D$ then $p \otimes q : A \otimes B \to C \otimes D$ and $p^\perp : C^\perp \to A^\perp$ are defined by the transition rules below. The rules for $\otimes$ illustrate the asynchronous nature of $\mathcal{ASP}roc$; the two processes can make transitions either independently or simultaneously.

$$\frac{p \xrightarrow{(a,c)} p'}{p \otimes q \xrightarrow{((a,\tau_B),(c,\tau_D))} p' \otimes q} \qquad \frac{q \xrightarrow{(b,d)} q'}{p \otimes q \xrightarrow{((\tau_A,b),(\tau_C,d))} p \otimes q'}$$

$$\frac{p \xrightarrow{(a,c)} p' \qquad q \xrightarrow{(b,d)} q'}{p \otimes q \xrightarrow{((a,b),(c,d))} p' \otimes q'} \qquad \frac{p \xrightarrow{(a,c)} p'}{p^\perp \xrightarrow{(c,a)} p'^\perp}$$

The tensor unit I is defined by

$$\Sigma_I \stackrel{\text{def}}{=} \{\tau_I\} \qquad\qquad S_I \stackrel{\text{def}}{=} \{\varepsilon\}.$$

The morphisms expressing the symmetric monoidal closed structure of $\mathcal{ASP}roc$ are formed from identity morphisms by applying suitable relabellings. For example, $\mathsf{symm}_{A,B} : A \otimes B \cong B \otimes A$ is defined by

$$\frac{\mathsf{id}_{A \otimes B} \xrightarrow{((a,b),(a,b))} \mathsf{id}_{(A/a)\otimes(B/b)}}{\mathsf{symm}_{A,B} \xrightarrow{((a,b),(b,a))} \mathsf{symm}_{A/a,B/b}.}$$

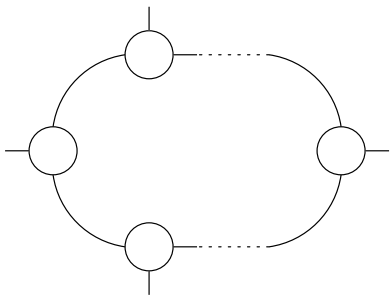
Currying is similarly defined by relabelling: if $f : A \otimes B \to C$ then $\Lambda(f) : A \to (B \multimap C)$ is defined by

$$\frac{f \xrightarrow{((a,b),c)} f'}{\Lambda(f) \xrightarrow{(a,(b,c))} \Lambda(f').}$$

Proposition 1 *$\mathcal{ASP}roc$ is a $*$-autonomous category.*

Because the linear negation $(-)^\perp$ is trivial in $\mathcal{ASP}roc$, $\otimes$ and $\parr$ coincide. A $*$-autonomous category in which $\otimes$ and $\parr$ are the same (note that in general it is possible for this to happen even if linear negation is non-trivial) is called a *compact closed* category.

When dealing with interaction categories, compact closure is significant because it means that a wider class of process constructions is supported by the categorical structure. In a $*$-autonomous category, processes can be connected in arbitrary *acyclic* configurations, by means of composition and appropriate use of currying and application. However, the final step in the construction of a cyclic network can only be carried out if $\otimes$ and $\parr$ coincide.



This point is discussed in more detail elsewhere [2], and in Section 5 of this paper. In Section 4 we will construct a $*$-autonomous category of deadlock-free processes; this category turns out not to be compact closed, which means that additional verification rules are required in order to form cyclic networks.

Notation In this paper we will use a CCS-like notation to describe processes in $\mathcal{ASP}roc$. The main constructions used will be prefixing, non-deterministic sum and guarded recursion, with their usual interpretations in terms of labelled transition systems [19]. A formal calculus of typed asynchronous processes (with a weaker type system which does not guarantee deadlock-freedom) has been developed [20], and this calculus will eventually be combined with the ideas of the present paper, but for now we will concentrate on the semantic aspects of the type system for deadlock-freedom. A significant difference between our notation and that of CCS is that we do not need complementary actions such as a and $\bar{a}$. Instead, communication between two processes (on a particular port) consists of them both performing the same action a, and the distinction between input and output is indicated by the types.

3 Specification Structures

The notion of specification structure, at least in its most basic form, is no more than a variation on standard notions from category theory. Nevertheless, it provides an alternative view of these standard notions which is highly suggestive, particularly from a Computer Science point of view. Similar notions have been studied, for a variety of purposes, by Burstall and McKinna [18], Hoofman [14], O'Hearn and Tennent [21] and Pitts [23].

Let $\mathcal{C}$ be a category. A *specification structure* S over $\mathcal{C}$ is defined by the following data:

- for each object A of $\mathcal{C}$, a set $P_S A$ of "properties over A".
- for each pair of objects A, B of $\mathcal{C}$, a relation $S_{A,B} \subseteq P_S A \times \mathcal{C}(A,B) \times P_S B$.

We write $\varphi\{f\}\psi$ for $S_{A,B}(\varphi, f, \psi)$, borrowing the notation of Hoare triples. This relation is required to satisfy the following axioms, for $f : A \to B$, $g : B \to C$,

$\varphi \in P_S A$, $\psi \in P_S B$ and $\theta \in P_S C$:

$$\varphi\{\mathsf{id}_A\}\varphi \tag{1}$$

$$\varphi\{f\}\psi, \psi\{g\}\theta \Longrightarrow \varphi\{f\,;g\}\theta \tag{2}$$

The axioms (1) and (2) are typed versions of the standard Hoare logic axioms for "skip" and "sequential composition" [9]. Given $\mathcal{C}$ and S as above, we can define a new category $\mathcal{C}_S$ as follows. The objects are pairs (A, φ) with $A \in \mathsf{ob}\,\mathcal{C}$ and $\varphi \in P_S A$. A morphism $f : (A, \varphi) \to (B, \psi)$ is a morphism $f : A \to B$ in $\mathcal{C}$ such that $\varphi\{f\}\psi$.

Composition and identities are inherited from $\mathcal{C}$; the axioms (1) and (2) ensure that $\mathcal{C}_S$ is a category. Moreover, there is an evident faithful functor $\mathcal{C}_S \rightarrowtail \mathcal{C}$ given by $(A, \varphi) \mapsto A$.

Two simple examples of specification structures are useful at this point. In each case we specify the base category $\mathcal{C}$, the set of properties $P_S X$ over each object X, and the Hoare triple relation.

1. $\mathcal{C} = \mathcal{S}et$, $P_S X = X$, $a\{f\}b \overset{\text{def}}{\Leftrightarrow} f(a) = b$.
   In this case, $\mathcal{C}_S$ is the category of pointed sets.
2. $\mathcal{C} = \mathcal{R}el$, $P_S X = \{*\}$, $*\{R\}* \overset{\text{def}}{\Leftrightarrow} \forall x \in A, y, z \in B.(xRy \wedge xRz \Rightarrow y = z)$.
   Then $\mathcal{C}_S$ is the category of sets and partial functions.

The notion of specification structure acquires more substance when there is additional structure on $\mathcal{C}$ which should be lifted to $\mathcal{C}_S$. Suppose for example that $\mathcal{C}$ is a monoidal category, i.e. there are a bifunctor $\otimes : \mathcal{C}^2 \to \mathcal{C}$, an object I, and natural isomorphisms

$$\begin{aligned} \mathsf{assoc}_{A,B,C} &: (A \otimes B) \otimes C \cong A \otimes (B \otimes C) \\ \mathsf{unit}|_A &: \quad I \otimes A \cong A \\ \mathsf{unitr}_A &: \quad A \otimes I \cong A \end{aligned}$$

satisfying the standard coherence equations [17]. A specification structure for $\mathcal{C}$ must then correspondingly be extended with an action

$$\otimes_{A,B} : P_S A \times P_S B \to P_S(A \otimes B)$$

and an element $\mathsf{u} \in P_S I$ satisfying, for $f : A \to B$, $f' : A' \to B'$ and properties φ, φ', ψ, ψ', θ over suitable objects:

$$\begin{gathered} \varphi\{f\}\psi, \varphi'\{f'\}\psi' \Longrightarrow (\varphi \otimes \varphi')\{f \otimes f'\}(\psi \otimes \psi') \\ ((\varphi \otimes \psi) \otimes \theta)\{\mathsf{assoc}_{A,B,C}\}(\varphi \otimes (\psi \otimes \theta)) \\ (\mathsf{u} \otimes \varphi)\{\mathsf{unit}|_A\}\varphi \\ (\varphi \otimes \mathsf{u})\{\mathsf{unitr}_A\}\varphi. \end{gathered}$$

The monoidal structure on $\mathcal{C}$ can be lifted to $\mathcal{C}_S$, by defining

$$(A, \varphi) \otimes (B, \psi) = (A \otimes B, \varphi \otimes_{A,B} \psi).$$

The axioms ensure that the conditions for a monoidal product are satisfied in $\mathcal{C}_S$. The action of $\otimes$ on morphisms in $\mathcal{C}_S$ is the same as in $\mathcal{C}$, and the axioms guarantee that this is correct: if $f : (A, \varphi) \to (B, \psi)$ and $g : (A', \varphi') \to (B', \psi')$, then

$$f \otimes g : (A \otimes A', \varphi \otimes \varphi') \to (B \otimes B', \psi \otimes \psi').$$

Moreover, the monoidal structure is preserved by the faithful functor $\mathcal{C}_S \rightarrowtail \mathcal{C}$.

In the present paper we are interested in specification structures over $*$-autonomous categories, in particular $\mathcal{ASP}roc$. If $\mathcal{C}$ is a $*$-autonomous category with a notion of a set of *processes* of each type, written $\mathsf{Proc}(A)$ (a process P of type A may be identified with a morphism $P : I \to A$), then the following sequence of steps provides a convenient way to define a specification structure S over $\mathcal{C}$. This sequence will be used in the next Section when defining specification structures over $\mathcal{ASP}roc$; it mirrors the sequence already used in the definition of $\mathcal{ASP}roc$ itself.

1. Define $P_S A$ for each A.
2. For each A, define a relation of *satisfaction*: $\models_A \subseteq \mathsf{Proc}(A) \times P_S A$.
3. Define $(-)^{\perp}_A$.
4. Define $\otimes_{A,B}$ and hence $\parr_{A,B}$ and $\multimap_{A,B}$.
5. Define the Hoare triple relation by $\theta\{f\}\varphi \stackrel{\text{def}}{\Leftrightarrow} f \models_{A \multimap B} \theta \multimap \varphi$.
6. Verify that the desired structure of $\mathcal{C}$, including the $*$-autonomous structure, lifts to $\mathcal{C}_S$.

Consider again the idea of a tower of categories, mentioned in the introduction:

$$\mathcal{C}_0 \leftarrowtail \mathcal{C}_1 \leftarrowtail \mathcal{C}_2 \leftarrowtail \cdots \leftarrowtail \mathcal{C}_k.$$

Such a tower arises by progressively refining $\mathcal{C}_0$ by specification structures

$$S_1, \ldots, S_k$$

so that $\mathcal{C}_{i+1} = (\mathcal{C}_i)_{S_{i+1}}$. Each such step adds propositional information to the underlying "raw" computational entities (morphisms of $\mathcal{C}_0$). The aim of verification in this framework is to "promote" a morphism from $\mathcal{C}_i$ to $\mathcal{C}_j$, where $i < j$. That is, to promote a $\mathcal{C}_0$ morphism $f : A \to B$ to a $\mathcal{C}_k$ morphism $f : (A, \varphi_1, \ldots, \varphi_k) \to (B, \psi_1, \ldots, \psi_k)$ is precisely to establish the "verification conditions" $\bigwedge_{i=1}^{k} \varphi_i\{f\}\psi_i$. Once this has been done, by whatever means—model checking, theorem proving, manual verification, etc.—the morphism is now available in $\mathcal{C}_k$ to participate in typing judgements there. In this way, a coherent framework for combining methods, including both compositional and non-compositional approaches, begins to open up.

4 A Specification Structure for Deadlock-Freedom

4.1 The Category $\mathcal{ASP}roc_V$

We need to define a category $\mathcal{ASP}roc_V$, which is like $\mathcal{ASP}roc$ except that each type is augmented with a notion of *valid maximal trace*. For any $\mathcal{ASP}roc$ type A, we can define

$$\begin{aligned}\overline{S_A} &= \{s \in \mathsf{ObAct}(A)^\omega \mid \forall t \in \mathsf{ObAct}(A)^*.t \sqsubseteq s \Rightarrow t \in S_A\}\\ \mathsf{maxfintraces}(A) &= \{s \in S_A \mid \forall t \in S_A.s \sqsubseteq t \Rightarrow s = t\}\\ \mathsf{maxtraces}(A) &= \mathsf{maxfintraces}(A) \cup \overline{S_A},\end{aligned}$$

where $\sqsubseteq$ is the prefix ordering on traces.

An object A of $\mathcal{ASP}roc_V$ is a tuple $(\Sigma_A, \tau_A, S_A, V_A)$ such that (Σ_A, τ_A, S_A) is an object of $\mathcal{ASP}roc$ and $V_A \subseteq \mathsf{maxtraces}(A)$. A process of type A in $\mathcal{ASP}roc_V$ is a process of type (Σ_A, τ_A, S_A) in $\mathcal{ASP}roc$.

The linear type structure of $\mathcal{ASP}roc_V$ is defined as for $\mathcal{ASP}roc$, with the addition of

$$\begin{aligned}V_{A^\perp} &= V_A\\ V_{A\otimes B} &= \{s \in \mathsf{maxtraces}(A \otimes B) \mid s\restriction A \in V_A \wedge s\restriction B \in V_B\}\\ V_I &= \{\varepsilon\}.\end{aligned}$$

As suggested by the name, $\mathcal{ASP}roc_V$ can be defined by means of a specification structure V over $\mathcal{ASP}roc$. However, the notation is easier to control if $\mathcal{ASP}roc_V$ is defined directly.

When we define the specification structure for deadlock-freedom, the component V_A of a type A will be used to define what we mean by deadlock. Finite traces in V_A are interpreted as successful terminations. Infinite traces in V_A are constructed from infinite traces or successfully terminated traces in simpler types. Notice that because $S_I = \{\varepsilon\}$, ε is a maximal finite trace in I and is interpreted as a successful termination.

At this point we can say a few words about divergence. Consider the processes $P : A \to B$ and $Q : B \to C$, defined by $P = (\tau_A, b).P$ and $Q = (b, \tau_C).Q$. It does not matter what the types A, B and C are, as long as $b \in \mathsf{ObAct}(B)$. If P and Q are forced to communicate, in the composite $P\,;Q$, the result is divergence: they can communicate internally forever, and never produce any observable actions. If deadlock is simply taken to be termination (this approach was used in our earlier work [2, 11]), then $P\,;Q$ is deadlocked even though successful communication takes place. The introduction of the sets V of valid infinite traces provides a nice resolution of this conflict. As we will see, for P and Q to be individually acceptable means that the traces $(\tau_A, b)^\omega$ and $(b, \tau_C)^\omega$ must be valid infinite traces in the types $A \multimap B$ and $B \multimap C$ respectively. This in turn means that

$\varepsilon \in V_A$ and $\varepsilon \in V_C$, hence $\varepsilon \in V_{A \multimap C}$ and $P\,;Q$ is interpreted as a successfully terminated process.

Compared to our previous work [2, 11], the present theory offers a much more satisfactory treatment of deadlock-freedom in interaction categories of asynchronous processes. The improvements stem from the use of the sets V of valid maximal traces; although this is a simple addition, it makes the theory much more elegant. In the earlier version, all processes were required to run forever; the fact that the type I of $\mathcal{ASP}roc$ does not permit infinite behaviour meant that the category of deadlock-free processes had no tensor unit. This defect in the categorical structure has now been rectified. Before the introduction of the sets V, the only way of avoiding problems such as the divergent composition of P and Q above was to require every process to be *fair*. Fairness of a process meant equal use of all ports, in the sense that every infinite behaviour had to perform an infinite sequence of observable actions in every port. To preserve this property it was necessary to make fairness assumptions about the way in which $\otimes$ (parallel composition) interleaved process behaviours. These fairness assumptions were incorporated into the theory by attaching to each process a specification of which of its infinite behaviours were to be ignored. This rather awkward feature has now been eliminated. Finally, the introduction of successful termination into the theory allows a wider range of process behaviour to be analysed.

We should also point out that the treatment of deadlock-freedom in the asynchronous case is significantly more complex than in the synchronous case. The problem of divergence does not arise in categories based on $\mathcal{SP}roc$, because a process performs actions in all of its ports at every step (this is a consequence of the synchrony assumption). Synchrony also means that all parts of a system must terminate simultaneously, if at all, and so successful termination is a much less useful concept; it does not make sense to consider successful termination of the behaviour in an individual port.

4.2 The Specification Structure $\boldsymbol{D}$

In order to discuss correct communication, we need the operation $\sqcap$ which selects the behaviours in common to two processes—i.e. the behaviours which correspond to sequences of communication. If $P, Q : A$ in $\mathcal{ASP}roc_V$, then the process $P \sqcap Q : A$ is defined by the following transition rules.

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \sqcap Q \xrightarrow{a} P' \sqcap Q'} \qquad \frac{P \xrightarrow{\tau_A} P'}{P \sqcap Q \xrightarrow{\tau_A} P' \sqcap Q} \qquad \frac{Q \xrightarrow{\tau_A} Q'}{P \sqcap Q \xrightarrow{\tau_A} P \sqcap Q'}$$

Before looking at ways of combining processes without introducing deadlocks, we need to restrict attention to processes which, in isolation, do not terminate unsuccessfully or diverge. Let $P : A$ in $\mathcal{ASP}roc_V$. Then $P\!\downarrow$ (P converges) if and only if whenever $P \stackrel{s}{\Longrightarrow} Q$ there is $t \in \mathsf{alltraces}(Q)$ such that $st \in V_A$. *The notion of*

deadlock-freedom used in this paper is convergence. Hence the distinction between deadlock and successful termination is determined by the safety specification, rather than by distinguishing the final states in the execution of the process.

Note that convergence of P does not mean that every maximal trace of P is valid; rather, it means that P never reaches a state from which only invalid maximal traces are possible. For example, $P \otimes Q$ has infinite traces which contain no contribution from Q; these traces ignore certain ports, and could lead to divergence when composition is used. But if P and Q are convergent, so is $P \otimes Q$, because from any state, the valid maximal behaviours of P and Q can be combined to yield a valid maximal behaviour of $P \otimes Q$. In a system containing $P \otimes Q$, we would like to consider only the valid traces of $P \otimes Q$; in practice, some form of fair scheduler would be needed. We interpret the definition of convergence as a way of allowing us to avoid considering the scheduler explicitly. This definition of convergence, and the way it allows us to type more processes without introducing fairness constraints, is another difference between the present theory and our previous work.

For each object A of $\mathcal{ASP}roc_V$, $\mathsf{Proc}(A) \stackrel{\text{def}}{=} \{P : A \mid P\!\downarrow\}$. There is an *orthogonality* relation on $\mathsf{Proc}(A)$, defined by $P \perp Q \iff (P \sqcap Q)\!\downarrow$. If P and Q are orthogonal, then any sequence of communications between them can be extended to either a valid infinite behaviour or a successfully terminated behaviour.

Orthogonality can be lifted to sets of processes, and used to define the notion of linear negation on sets of processes. If P is a process and U a set of processes, define

$$\begin{aligned} P \perp U &\iff \forall Q \in U. P \perp Q \\ U^{\perp} &= \{P \mid P \perp U\}. \end{aligned}$$

$U^{\perp}$ is the set of processes which can successfully communicate with processes in the set U. If U describes the possible behaviours in a particular port, then $U^{\perp}$ is the type of ports which can be connected to that port without introducing deadlocks.

The specification structure D over $\mathcal{ASP}roc_V$ is obtained from the following sequence of definitions.

$$\begin{aligned} P_D A &= \{U \subseteq \mathsf{Proc}(A) \mid U \neq \varnothing, U = U^{\perp\perp}\} \\ U \otimes V &\stackrel{\text{def}}{=} \{P \otimes Q \mid P \in U, Q \in V\}^{\perp\perp} \\ U \parr V &\stackrel{\text{def}}{=} (U^{\perp} \otimes V^{\perp})^{\perp} \\ U \multimap V &\stackrel{\text{def}}{=} (U \otimes V^{\perp})^{\perp} \end{aligned}$$

If $P : A$ and $U \in P_D A$ then $P \models U \iff P \in U$. Finally, $U\{f\}V \iff f \models U \multimap V$.

The composition axiom is the key property which needs to be checked. Suppose that $f : A \to B$ and $g : B \to C$ with $f \models U \multimap V$ and $g \models V \multimap W$. The

types can be rewritten as $f \models U^\perp \parr V$ and $g \models V^\perp \parr W$. The behaviour of f in its second port is described by the processes in the set V, and the behaviour of g in its first port is described by the processes in the set $V^\perp$. The definition of orthogonality means that when f and g are connected together along those ports (which is what $f\,;g$ means), communication is successful and no deadlocks arise. This is not the complete proof, but gives the general idea of the argument.

The category constructed from $\mathcal{ASP}roc_V$ by means of the specification structure D is our category of deadlock-free processes. For consistency it should be called $\mathcal{ASP}roc_{V\,D}$, but we will refer to it as $\mathcal{ASP}roc_D$. The multiplicative structure of $\mathcal{ASP}roc_V$ can be lifted to the new category, and $\mathcal{ASP}roc_D$ is $*$-autonomous.

4.3 Standard Deadlock-Free Types

It is useful to have a collection of standard deadlock-free types over each $\mathcal{ASP}roc_V$ type. Two obvious types, which we will now define, correspond to *input* and *output*. For each type A of $\mathcal{ASP}roc_V$, we have $\mathsf{Proc}(A) \in P_D A$. $\mathsf{Proc}(A)$ is the maximal property over A; it is satisfied by every convergent process of type A. Using $\mathsf{Proc}(A)$ as a specification imposes no constraint on a process, and so a port with the $\mathcal{ASP}roc_D$ type $(A, \mathsf{Proc}(A))$ corresponds to a possibly non-deterministic output. We will therefore refer to the property $\mathsf{Proc}(A)$ as out_A; every convergent process of type A satisfies out_A. Dually we expect to use a property in_A to represent an input; naturally the definition is $\mathsf{in}_A = \mathsf{out}_A^\perp$. It can be shown that $\mathsf{in}_A = \{\mathsf{max}_A\}$, where max_A is defined by

$$\frac{a \in S_A}{\mathsf{max}_A \xrightarrow{a} \mathsf{max}_{A/a}}.$$

The process max_A is always prepared to engage in any action allowed by the underlying safety specification; this is precisely the property required of an input port.

The most useful fact about in and out properties is that for any A and B, $\mathsf{in}_A \otimes \mathsf{in}_B = \mathsf{in}_{A\otimes B}$. We will not prove this fact here, but it expresses the intuitive idea that a pair of input ports can be viewed as a single input of a compound type. Dually, $\mathsf{out}_A \parr \mathsf{out}_B = \mathsf{out}_{A\parr B}$. This fact allows certain $\mathcal{ASP}roc_V$ processes to be immediately assigned types in $\mathcal{ASP}roc_D$, as follows. If $P : A_1 \parr \cdots \parr A_n$ in $\mathcal{ASP}roc_V$, and $P\!\downarrow$, then $P \models \mathsf{out}_{A_1 \parr \cdots \parr A_n} = \mathsf{out}_{A_1} \parr \cdots \parr \mathsf{out}_{A_n}$. This means that P has type $(A_1, \mathsf{out}_{A_1}) \parr \cdots \parr (A_n, \mathsf{out}_{A_n})$ in $\mathcal{ASP}roc_D$.

The properties in and out only allow us to describe very simple patterns of interaction, namely those in which each port is committed to being either an input or an output for all time. For the examples which we consider in the present paper, these simple patterns are sufficient. However, the structure of $\mathcal{ASP}roc_D$ is able to support much richer interaction patterns, and future work will address the question of using $\mathcal{ASP}roc_D$ to provide a semantics for a more realistic language of types.

As indicated earlier, the category $\mathcal{ASP}roc_D$ is not compact closed: $\otimes$ and $\parr$ are in general different. The simplest example of this loss of compact closure is as follows. Consider the $\mathcal{ASP}roc_V$ types A and B, defined by

$$\begin{array}{ll} \Sigma_A = \{a, \tau_A\} & \Sigma_B = \{b, c, \tau_B\} \\ S_A = \{a\}^* & S_B = \{b, c\}^* \\ V_A = \{a^\omega\} & V_B = \{b, c\}^\omega. \end{array}$$

We will outline the proof that $\mathsf{in}_A \otimes \mathsf{in}_B \neq \mathsf{in}_A \parr \mathsf{in}_B$. First, we have $\mathsf{in}_A \otimes \mathsf{in}_B = \mathsf{in}_{A\otimes B} = \{\mathsf{max}_{A\otimes B}\}$. Also, because max_A is the only convergent process of type A, $\mathsf{in}_A = \mathsf{out}_A$. The definitions of $\parr$ and $\otimes$ give

$$\begin{aligned} \mathsf{in}_A \parr \mathsf{in}_B &= (\mathsf{out}_A \otimes \mathsf{out}_B)^\perp \\ &= \{\mathsf{max}_A \otimes P \mid P \in \mathsf{Proc}(B)\}^\perp \end{aligned}$$

To prove that $\mathsf{in}_A \otimes \mathsf{in}_B \neq \mathsf{in}_A \parr \mathsf{in}_B$ we just need to find a process $Q : A \otimes B$ such that $\forall P \in \mathsf{Proc}(B).Q \perp (\mathsf{max}_A \otimes P)$ but $Q \neq \mathsf{max}_{A\otimes B}$. If Q is defined by $Q = (a, b).Q + (a, c).Q$ then this condition is satisfied. Clearly $Q \neq \mathsf{max}_{A\otimes B}$ because Q never performs an action (a, τ_B). However, we do have $(Q \sqcap (\mathsf{max}_A \otimes P))\downarrow$ for every $P \in \mathsf{Proc}(B)$; indeed, the behaviour of $Q \sqcap (\mathsf{max}_A \otimes P)$ is simply that of P with actions b replaced by (a, b) and actions c replaced by (a, c).

4.4 Constructing Cyclic Networks in $\mathcal{ASP}roc_D$

Because $\mathcal{ASP}roc_D$ is not compact closed, we are in general unable to construct cyclic configurations of processes. This is to be expected, as it is the presence of cycles which can lead to deadlock. However, in a particular system there may be other reasons why specific cycles can be formed without introducing deadlock. We can formulate the following proof rule for the construction of deadlock-free cycles:

$$\frac{P : (\Gamma, U) \parr (X, W) \parr (X^\perp, W^\perp)}{\overline{P} : (\Gamma, U)}\ \mathsf{cycle}(P)$$

Here P is a process representing the acyclic portion of the desired configuration; it has two ports, of types (X, W) and $(X^\perp, W^\perp)$, which could potentially be connected together. The port of type (Γ, U) may in general be composed of several ports: $(\Gamma, U) = (A_1, U_1) \parr \cdots \parr (A_n, U_n)$. The process $\overline{P}$ results from connecting the X and $X^\perp$ ports, and forcing the actions in those ports to match. Thus the behaviour of $\overline{P}$ is defined by:

$$\frac{P \xrightarrow{(a,x,x)} Q}{\overline{P} \xrightarrow{a} \overline{Q}}$$

$\mathsf{cycle}(P)$ is a sufficient condition for $\overline{P}$ to be deadlock-free. It is defined by:

For any $Q \in U^\perp$, if $P \stackrel{s}{\Longrightarrow} P'$ and $Q \stackrel{s\restriction\Gamma}{\Longrightarrow} Q'$ with $\pi_X(s) = \pi_{X^\perp}(s)$, $\exists t \in \mathsf{allobtraces}(P')$ such that $t\restriction\Gamma \in \mathsf{allobtraces}(Q')$, $\pi_X(t) = \pi_{X^\perp}(t)$, and $st \in V_{\Gamma \parr X \parr X^\perp}$.

$\pi_X(s)$ and $\pi_{X^\perp}(s)$ are the projections of the trace s in the X and $X^\perp$ ports respectively; τ actions are preserved by these projections.

The cycle condition has two functions. The first is to ensure that $\overline{P}$ is convergent. The second is to ensure that $\overline{P}$ is able to interact with its environment as required by the type (Γ, U). It is the second part which requires processes $Q \in U^\perp$ to be considered; they are the processes which $\overline{P}$ must be able to interact with.

Proposition 2 *The proof rule for cycle formation is semantically sound.*

Proof We need to prove that $\overline{P} \in U$, i.e. that $\overline{P} \perp U^\perp$. Let $Q \in U^\perp$. In order to prove that $(\overline{P} \parr Q)\downarrow$, suppose that $\overline{P} \parr Q \stackrel{u}{\Longrightarrow} \overline{P'} \parr Q'$. This means that $P \stackrel{s}{\Longrightarrow} P'$ and $Q \stackrel{u}{\Longrightarrow} Q'$ with $u = s\restriction\Gamma$ and $\pi_X(s) = \pi_{X^\perp}(s)$.

The condition $\mathsf{cycle}(P)$ gives $t \in \mathsf{allobtraces}(P')$ such that $t\restriction\Gamma \in \mathsf{allobtraces}(Q')$, $\pi_X(t) = \pi_{X^\perp}(t)$ and $st \in V_{\Gamma \parr X \parr X^\perp}$.

Let $v = t\restriction\Gamma$. Then $v \in \mathsf{allobtraces}(\overline{P'} \parr Q')$ and $uv \in V_\Gamma$. Thus v is the required extension of u, and we have established $(P \parr Q)\downarrow$. □

The proof rule for cycle formation is not compositional; in order to check the condition $\mathsf{cycle}(P)$ we may need to examine the internal behaviour of P. The details of this check will be specific to particular examples. Once $\mathsf{cycle}(P)$ has been established, the process $\overline{P}$ has a type in $\mathcal{ASP}roc_D$ and can be combined with other processes on the basis of that type. We have separated the compositional (acyclic) and non-compositional (cyclic) verification steps, and identified the condition which must be checked when cycles are formed.

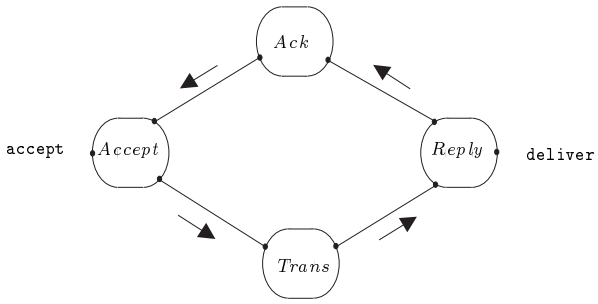
5 Deadlock-Freedom of the Alternating-Bit Protocol

5.1 The Protocol

The Alternating-Bit Protocol is a communications protocol which is used for the transmission of messages under adverse conditions. In the diagram below, *Accept* receives a message and sends it across the transmission medium; *Reply* delivers the message to its destination and also sends an acknowledgement signal back to *Accept*; and *Trans* and *Ack* are the two parts of the transmission medium, one for each direction.

There are several versions of the protocol, with different assumptions about the nature of the transmission medium. In the version which we will consider, corresponding to Exercise 15 in Chapter 6 of Milner's book [19], the transmission

lines *Trans* and *Ack* each have capacity for at most one message, and may lose messages. Thus any message sent by *Accept*, and any acknowledgement sent by *Reply*, may disappear unpredictably. Furthermore, the bounded capacity of the transmission lines introduces the possibility of deadlock because they may not be prepared to accept input.



The basic idea of the protocol is that a single bit, either 0 or 1, is attached to each message or acknowledgement which is sent. An acknowledgement of a message always carries the same bit as the original message, and successive messages carry opposite bits. If *Accept* receives an acknowledgement with the same bit as the most recent message, then the next message can be sent; otherwise, the previous message must be resent. Similarly, if *Reply* receives a message with the opposite bit to the most recent acknowledgment, then it is a new message which can be acknowledged; otherwise, the previous acknowledgement is judged to have been lost and must be sent again.

Using a CCS-style syntax, the processes can be defined as follows. Note that the subscript b can be either 0 or 1, parameterising the processes and the actions on the value of the current bit. As mentioned in Section 2, the interaction category approach does not use complementary actions, and this is reflected in the definitions.

$$
\begin{aligned}
Accept_b &= \mathtt{sen}_b.Accept'_b \\
Accept'_b &= \tau.Accept_b + \mathtt{ack}_b.Accept''_{\neg b} + \mathtt{ack}_{\neg b}.Accept'_b \\
Accept''_b &= \mathtt{accept}.Accept_b \\
Reply_b &= \mathtt{rec}_b.Reply'_b \\
Reply'_b &= \tau.Reply_b + \mathtt{tra}_{\neg b}.Reply''_{\neg b} + \mathtt{tra}_b.Reply'_b \\
Reply''_b &= \mathtt{deliver}.Reply_b \\
Trans &= \mathtt{sen}_0.Trans_0 + \mathtt{sen}_1.Trans_1 \\
Trans_b &= \mathtt{tra}_b.Trans + \tau.Trans \\
Ack &= \mathtt{rec}_0.Ack_0 + \mathtt{rec}_1.Ack_1 \\
Ack_b &= \mathtt{ack}_b.Ack + \tau.Ack
\end{aligned}
$$

It is important to appreciate the various roles played by the silent actions τ in these definitions. In *Accept*, τ is used to model the receipt of a timeout signal

from a timer (which is not explicitly represented in our system). If timeout occurs before the acknowledgement of a message is received, the message is retransmitted. In *Reply*, τ also models a timeout, this time limiting the waiting period for the next message to arrive. If timeout occurs, the previous acknowledgement is repeated. However, the τ actions in *Trans* and *Ack* represent spontaneous loss of messages.

The complete system is defined as follows. The current state assumes that a message has just been delivered and another is ready to be accepted.

$$\begin{aligned} Protocol = (Accept_1'' \mid Trans \mid Ack \mid Reply_0')\backslash \\ \{\mathtt{sen}_0, \mathtt{sen}_1, \mathtt{tra}_0, \mathtt{tra}_1, \mathtt{rec}_0, \mathtt{rec}_1, \mathtt{ack}_0, \mathtt{ack}_1\} \end{aligned}$$

The remainder of this section will demonstrate the assignment of types to the components of the system, in both $\mathcal{ASP}roc_V$ and $\mathcal{ASP}roc_D$. It is important to note that we are only considering one aspect of the system's correctness, namely deadlock-freedom. The types do not express the specification that the sequence of delivered messages is the same as the sequence of received messages. This specification could be expressed in our typed framework [20], and we have analysed similar specifications of other systems [1, 11], but it is not the purpose of the present paper to illustrate this aspect of the theory.

5.2 Types in $\mathcal{ASP}roc_V$

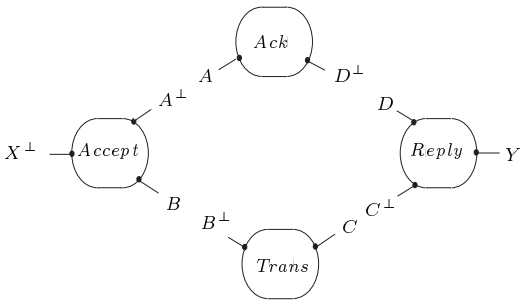
In order to type the system at the most elementary level, without taking account of deadlock-freedom, we need the following $\mathcal{ASP}roc_V$ types. In each case, the safety specification consists of all finite traces over the alphabet.

$$\begin{array}{ll} \Sigma_X = \{\tau_X, \mathtt{accept}\} & V_X = \{\mathtt{accept}^\omega\} \\ \Sigma_Y = \{\tau_Y, \mathtt{deliver}\} & V_Y = \{\mathtt{deliver}^\omega\} \\ \Sigma_A = \{\tau_A, \mathtt{ack}_0, \mathtt{ack}_1\} & V_A = \{(\mathtt{ack}_0|\mathtt{ack}_1)^\omega\} \\ \Sigma_B = \{\tau_B, \mathtt{sen}_0, \mathtt{sen}_1\} & V_B = \{(\mathtt{sen}_0|\mathtt{sen}_1)^\omega\} \\ \Sigma_C = \{\tau_C, \mathtt{tra}_0, \mathtt{tra}_1\} & V_C = \{(\mathtt{tra}_0|\mathtt{tra}_1)^\omega\} \\ \Sigma_D = \{\tau_D, \mathtt{rec}_0, \mathtt{rec}_1\} & V_D = \{(\mathtt{rec}_0|\mathtt{rec}_1)^\omega\} \end{array}$$

There is a certain amount of redundancy in these definitions: the types A, B, C and D are isomorphic, as are X and Y. However, using different names makes it easier to keep track of the steps in the construction of the system.

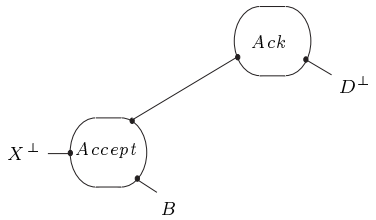
With these definitions, the types of the components of the system are as follows. Some changes are necessary to the equations defining the processes, to take account of the fact that an observable action in one port must be accompanied by silent actions in the other ports. For example, the definition of Ack becomes $Ack = (\tau_A, \mathtt{rec}_0).Ack_0 + (\tau_A, \mathtt{rec}_1).Ack_1$.

$$
\begin{aligned}
\mathit{Accept} &: X^{\perp} \parr B \parr A^{\perp} \\
\mathit{Trans} &: B^{\perp} \parr C \\
\mathit{Reply} &: Y \parr C^{\perp} \parr D \\
\mathit{Ack} &: A \parr D^{\perp}
\end{aligned}
$$



The types of the ports of each process are combined using $\parr$. In $\mathcal{ASP}roc_V$ this could just as well be $\otimes$, but again it is useful to stick to notation which reflects the logical distinctions between the connectives. In general, $\parr$ relates to connected concurrency and $\otimes$ to disjoint concurrency; $\parr$ is the appropriate connective for combining ports of a single process. Again to maintain the logical distinction between various types, we have distinguished between, for example, A and $A^{\perp}$ despite the fact that at the $\mathcal{ASP}roc_V$ level these types are identical.

The categorical structure of $\mathcal{ASP}roc_V$ allows the components to be connected in the desired configuration. The first step is to connect Accept and Ack.



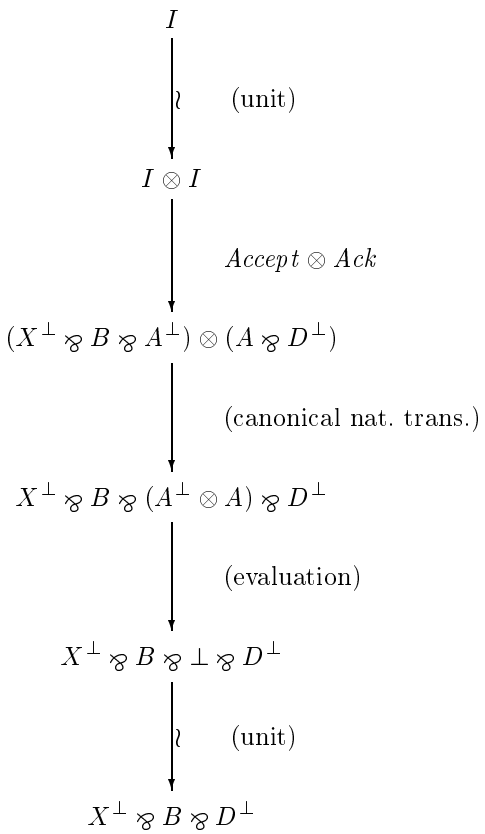
The typed processes

$$
\begin{aligned}
\mathit{Accept} &: X^{\perp} \parr B \parr A^{\perp} \\
\mathit{Ack} &: A \parr D^{\perp}
\end{aligned}
$$

can be viewed as morphisms

$$
\begin{aligned}
\mathit{Accept} &: I \to X^{\perp} \parr B \parr A^{\perp} \\
\mathit{Ack} &: I \to A \parr D^{\perp}
\end{aligned}
$$

and the following categorical calculation produces a morphism $I \to X^{\perp} \parr B \parr D^{\perp}$.



This morphism, viewed as a process of type $X^\perp \parr B \parr D^\perp$, corresponds to the process which, in CCS notation, is written $(Accept \mid Ack)\backslash\{\mathtt{ack}_0, \mathtt{ack}_1\}$.

Another view of this construction is as follows. The structure of a compact closed category allows us to construct morphisms (once again, denoted here by *Accept* and *Ack* for convenience)

$$\begin{array}{c} Accept : (X^\perp \parr B)^\perp \to A^\perp \\ Ack : A^\perp \to D^\perp \end{array}$$

which have the same process behaviour as before, but differently organised interfaces. (For more details of the way in which a single process can be viewed as many different morphisms, see [1, 11]). Now, connecting *Accept* and *Ack* is simply an application of categorical composition.

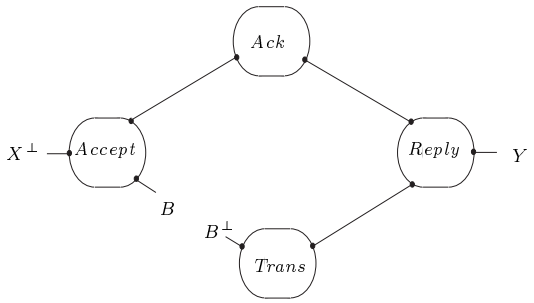
However we use the categorical structure to form process connections, the effect is to justify the following typing rule:

$$\frac{P : A \parr B \quad Q : B^\perp \parr C}{(P \mid Q)\backslash\Sigma_B : A \parr C}$$

Again we are using an informal CCS-like notation here. In our typed calculus of synchronous processes [12], based on the structure of $\mathcal{SP}roc$, the corresponding operation (treated formally) is *cut*.

Using the same typing rule we can connect *Reply* and then *Trans*, to obtain the following process.

$$PreProtocol : X^{\perp} \parr B \parr Y \parr B^{\perp}.$$



The following categorical calculation, justified by the compact closed structure of $\mathcal{ASP}roc_V$, allows the free ends to be connected.

$$I$$

$\downarrow$ *PreProtocol*

$$X^{\perp} \parr B \parr Y \parr B^{\perp}$$

(permutation)

$$X^{\perp} \parr Y \parr B^{\perp} \parr B$$

(iso in $\mathcal{ASP}roc_V$)

$$X^{\perp} \parr Y \parr B^{\perp} \otimes B$$

(evaluation)

$$X^{\perp} \parr Y \parr \perp$$

$\downarrow$ (unit)

$$X^{\perp} \parr Y$$

The final process is

$$Protocol : X^{\perp} \parr Y.$$

Because $X \cong Y$, we can also write $Protocol : X^{\perp} \parr X$ to emphasise the fact that the receiving and delivering ports are of compatible types.

This calculation also justifies a general typing rule (valid in $\mathcal{ASP}roc_V$ but not in $\mathcal{ASP}roc_D$), again using an informal process notation:

$$\frac{P : A \parr B \parr B^{\perp}}{P \backslash \Sigma_B : A}$$

5.3 Types in $\mathcal{ASP}roc_D$

Each process in the system is convergent. We will explain why this is true for *Accept*; the other cases are similar. In each of the types A, B and X, all infinite traces are valid. Hence $V_{X^{\perp} \parr B \parr A^{\perp}}$ consists of the infinite traces which include infinite sequences of actions in the X, B and A ports. Whenever $Accept \stackrel{s}{\Longrightarrow} P$, we can see from the definition of *Accept* that there are infinite behaviours of P which never take the τ branch of $Accept'$. Any such behaviour generates a trace t which includes `sen` actions, `ack` actions and the `accept` action infinitely often. Hence $st \in V_{X^{\perp} \parr B \parr A^{\perp}}$.

The previous description of the protocol makes it clear whether each port is used as an input or as an output, and by using the properties in and out accordingly we expect the components of the system to have the following $\mathcal{ASP}roc_D$ types. Here we can drop the notational distinction between A and $A^{\perp}$, as the real distinction is now indicated by the presence of in or out.

$$\begin{aligned} Accept &: (X, \mathsf{in}_X) \parr (B, \mathsf{out}_B) \parr (A, \mathsf{in}_A) \\ Trans &: (B, \mathsf{in}_B) \parr (C, \mathsf{out}_C) \\ Reply &: (Y, \mathsf{out}_Y) \parr (C, \mathsf{in}_C) \parr (D, \mathsf{out}_D) \\ Ack &: (A, \mathsf{out}_A) \parr (D, \mathsf{in}_D) \end{aligned}$$

Verifying that the processes do indeed have these types requires some work. For example, to check $Trans : (B, \mathsf{in}_B) \parr (C, \mathsf{out}_C)$ we need to show that

$$\begin{aligned} Trans &\in \mathsf{in}_B \parr \mathsf{out}_C \\ &= (\mathsf{out}_B \otimes \mathsf{in}_C)^{\perp} \\ &= \{P \otimes \mathsf{max}_C \mid P \in \mathsf{out}_B\}^{\perp}. \end{aligned}$$

We do this semantically; we do not have a formal syntax and typing system which allows the individual components to be constructed with their $\mathcal{ASP}roc_D$ types. Such a typing system is being developed for the synchronous case [6] and will be adapted for the asynchronous case.

Proposition 3 $Trans \in \mathsf{in}_B \parr \mathsf{out}_C$.

Proof We need to show that for any convergent process P of type B,

$$\mathit{Trans} \perp P \otimes \mathsf{max}_C,$$

i.e.

$$(\mathit{Trans} \sqcap P \otimes \mathsf{max}_C)\downarrow .$$

Suppose

$$\mathit{Trans} \sqcap (P \otimes \mathsf{max}_C) \stackrel{s}{\Longrightarrow} Q \sqcap (P' \otimes \mathsf{max}_{C/s}).$$

This means that $P \stackrel{s\restriction B}{\Longrightarrow} P'$. Because $P \downarrow$ there is $t \in \mathsf{allobtraces}(P')$ such that $(s\restriction B)t \in V_B$. The definition of V_B means that t is infinite.

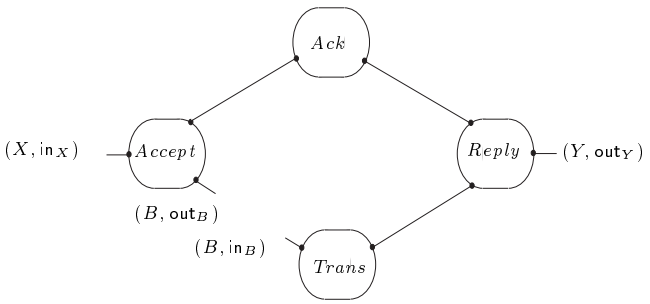
The process Q is either Trans, Trans_0 or Trans_1. We can find $u \in \mathsf{allobtraces}(Q)$ such that $u\restriction B = t$, by considering the behaviour of Q which alternately does a $\mathtt{sen}_0$ or $\mathtt{sen}_1$ action (whichever is needed to match t), and a $\mathtt{tra}$ action.

The definition of max processes means that $u\restriction C \in \mathsf{allobtraces}(\mathsf{max}_{C/s})$. Now $u\restriction C$ is infinite, because u contains a $\mathtt{tra}$ action for every $\mathtt{sen}$ action.

We now have $u \in \mathsf{allobtraces}(Q \sqcap (P' \otimes \mathsf{max}_{C/s}))$. We know that $(su)\restriction B$ is infinite, hence is in V_B, and $(su)\restriction C$ is infinite, hence is in V_C. Therefore $su \in V_{B \parr C}$, as required. □

Intuitively, we have made use of the fact that whenever Trans is capable of performing an action in its B port, it must offer a choice of all possible actions. This is consistent with our description of the deadlock-free type over B as in_B. Similar informal reasoning applies to the other components of the system; the arguments can also be formalised as above.

The $*$-autonomous structure of $\mathcal{ASP}\mathit{roc}_D$ allows the components to be connected together, using the same typing rule as in $\mathcal{ASP}\mathit{roc}_V$, up to the point at which the cycle is about to be completed. This gives the following process, $\mathit{PreProtocol}$.



Because we have already established that the individual components have compatible types, no extra work is needed to take the construction to this point. However, the type system of $\mathcal{ASP}\mathit{roc}_D$ does not allow the cycle to be closed. Thus we need to verify that the condition $\mathsf{cycle}(\mathit{PreProtocol})$ is satisfied, in order to establish that the process $\overline{\mathit{PreProtocol}} = \mathit{Protocol}$ has the type $(X, \mathsf{in}_X) \parr (Y, \mathsf{out}_Y)$ in $\mathcal{ASP}\mathit{roc}_D$.

To check the condition $\mathsf{cycle}(\mathit{PreProtocol})$, observe that $(\Gamma, U) = (X, \mathsf{in}_X) \parr (Y, \mathsf{out}_Y)$. We need to consider any $Q \in (\mathsf{in}_X \parr \mathsf{out}_Y)^\perp$. We have

$$\begin{aligned}(\mathsf{in}_X \parr \mathsf{out}_Y)^\perp &= \mathsf{out}_X \otimes \mathsf{in}_Y \\ &= \mathsf{in}_X \otimes \mathsf{in}_Y \\ &= \mathsf{in}_{X \otimes Y}\end{aligned}$$

by the duality of $\otimes$ and $\parr$, the fact that $\mathsf{out}_X = \mathsf{in}_X$ (because X has only one observable action), and the properties of in types. Hence we only need to consider $Q = \mathsf{max}_{X \otimes Y}$, since in general $\mathsf{in}_A = \{\mathsf{max}_A\}$.

Suppose $\mathit{PreProtocol} \stackrel{s}{\Longrightarrow} P'$, $\mathsf{max}_{X \otimes Y} \stackrel{s \upharpoonright X,Y}{\Longrightarrow} \mathsf{max}_{X \otimes Y}$ and $\pi_B(s) = \pi_{B^\perp}(s)$. We need to find $t \in \mathsf{allobtraces}(P')$ such that

- $t \upharpoonright X, Y \in \mathsf{allobtraces}(\mathsf{max}_{X \otimes Y})$ (this condition is vacuous, since max has all traces)
- $\pi_B(t) = \pi_{B^\perp}(t)$
- $st \in V_{X \parr B \parr Y \parr B}$ (this condition holds, provided that st is infinite).

Checking the condition $\mathsf{cycle}(\mathit{PreProtocol})$ reduces to checking that any behaviour of *PreProtocol* which has matching actions in the B ports, can be extended to an infinite behaviour of *PreProtocol* which has matching actions in the B ports. This is equivalent to checking that any behaviour of *Protocol* can be extended to an infinite behaviour, i.e. that *Protocol* does not deadlock. We carry out this check by using a traditional argument. The only potential deadlock arises from the use of a one-place buffer in the transmission lines *Trans* and *Ack*—if the buffer is full then a new message could be rejected. However, the τ actions in the defining equations for Trans_b and Ack_b play a dual role: as well as representing an undesirable message loss, they can also be interpreted as a deliberate message discard when a new message is received before the previous message has been passed on. This is a rather peculiar feature of this particular implementation of the alternating-bit protocol; the protocol could be modified, for example by introducing additional actions to represent fullness of the buffers, but such modifications would merely increase the complexity without changing the essential points of our argument.

The other purpose of the condition $\mathsf{cycle}(\mathit{PreProtocol})$ is to ensure that *Protocol* can interact with its environment, on the X and Y ports, as required by the type $(X, \mathsf{in}_X) \parr (Y, \mathsf{out}_Y)$. In this example, we have

$$\begin{aligned}\mathsf{in}_X \parr \mathsf{out}_Y &= \mathsf{out}_X \parr \mathsf{out}_Y \\ &= \mathsf{out}_{X \parr Y}\end{aligned}$$

and if *Protocol* is convergent then trivially $\mathit{Protocol} \in \mathsf{out}_{X \parr Y}$ (any convergent process satisfies an out type). Hence this part of the condition becomes trivial.

In other examples, it may be less trivial to verify that the process can continue to interact with its environment in the same way after completion of the cycle.

But once $\mathsf{cycle}(P)$ has been established, $\overline{P}$ has a type in $\mathcal{ASP}roc_D$ and can be combined with other processes on the basis of that type, without further verification. Any process with a port of type (X, out_X) can be connected to the (X, in_X) port of *Protocol* without introducing deadlock; similarly, any process with a port of type (Y, in_Y) can be connected to the (Y, out_Y) port of *Protocol*. No further checking is needed in order to establish the correctness of these connections.

In this example the types of the individual ports are very simple—just input or output. However, the overall type of, say, *Accept* is fairly complex: $\mathsf{in}_X \parr \mathsf{out}_B \parr \mathsf{in}_A$. The treatment of complex combinations of inputs and outputs in a uniform semantic theory is one of the significant features of our approach. It is also possible to describe more complex types for individual ports. For example, given a type A of $\mathcal{ASP}roc_V$, let

$$U = \{P \in \mathsf{Proc}(A) \mid \forall a, Q. P \stackrel{a}{\Longrightarrow} Q \Rightarrow Q = \mathsf{max}_{A/a}\}.$$

Then (A, U) is a type of $\mathcal{ASP}roc_D$ which specifies one step of output followed by repeated inputs. In general, the choice of input or output can be made independently at each step. The structure of $\mathcal{ASP}roc_D$ allows a rich variety of input/output behaviour to be specified.

6 Related Work

The analysis of deadlock-freedom in concurrency has been investigated by a number of authors [4, 7, 8, 10, 22, 24, 25]. The work of Roscoe and Daithi [24] is one of the most recent, and also the most relevant to our own. Their approach is to define a *variant function* which assigns a value to each state of a process. A network of processes, satisfying certain conditions, is deadlock-free if for each process P that is waiting for another process Q, the value of the state of P is greater than that of the state of Q. This enables local analysis of deadlock-freedom, and hence offers the possibility of constructing deadlock-free networks from deadlock-free subcomponents.

None of the above-mentioned approaches is based on types. However, Takeuchi, Honda and Kubo [26] have developed a typed language for interaction, in which the type system guarantees avoidance of a class of communication errors; these errors can be viewed as weak forms of deadlock, but do not include the possibility of cyclic dependencies. Based on this work, Kobayashi, Pierce and Turner [16] have developed a linear type system for the π-calculus. Recently Kobayashi [15] has proposed a process calculus with a type system which captures information about order of channel usage, and uses this information to guarantee deadlock-freedom. In this calculus, a distinction is made between reliable and unreliable channels, and it is the reliable channels whose use is guaranteed not to cause deadlock.

The main difference between our work and other work on type systems for deadlock avoidance is that we take a more semantic view; the structure of the category

$\mathcal{ASP}roc_D$ provides a uniform framework in which a range of communication behaviours can be described. The link with category theory and linear logic also means that our type system follows the tradition of the Curry-Howard isomorphism, well established in functional programming. Another distinguishing feature of the interaction categories approach is the use of specification structures to organise the process of successively adding information to a type system, so that a range of program properties can be discussed within a common framework. The key advance of the present paper over our previous work is a satisfactory treatment of an interaction category of *asynchronous* deadlock-free processes, achieved by adding the notion of valid maximal traces to the types.

7 Conclusions and Future Work

We have presented a typed framework for the verification of deadlock-freedom. The main novelty of this paper is the new category $\mathcal{ASP}roc_D$, which is a category of deadlock-free processes constructed by means of a specification structure over the category $\mathcal{ASP}roc_V$ ($\mathcal{ASP}roc$ with a notion of valid maximal traces). This is a simple but effective treatment of asynchronous deadlock-freedom incorporating notions of deadlock, divergence, and termination. We have also applied the techniques presented to verify the deadlock-freedom of the alternating-bit protocol. We see this work as a significant step in the development of type-theoretic methods for compositional verification of concurrent systems.

Work is in already progress to develop a formal syntax for deadlock-free processes, using ideas from the typed process calculi that we have already developed [11, 12, 20] and the typed language of Takeuchi *et al.* [26]. In addition, it is highly desirable to develop techniques whereby forming cyclic connections can be automated to some extent, perhaps under certain conditions. The use of variants as in Roscoe and Daithi's work [24] may provide some clues. Another possibility is the cycle sum test of Wadge [27], which captures the idea that the presence of non-trivial delays in feedback loops is a necessary condition for deadlock-freedom. In any case, extra information about processes would be needed in order to automate checking of cyclic connections; we would hope to be able to define an additional specification structure, on top of D, whose types incorporate the necessary data. This approach would further demonstrate the scope of specification structures for organising a range of type systems.

Future work will include applications of these ideas to more substantial examples, perhaps illustrating the use of successful termination. Although we have focused on deadlock-freedom in this paper, specification structures allow other properties such as liveness to be added in the same framework; we also aim to investigate this in the future. A specification structure for liveness, built on top of that for deadlock-freedom, would allow specification of more detailed properties of infinite executions. Finally, we would like to extend interaction categories to handle mobility, which would enable us to establish a more direct connection with other work based on the π-calculus.

Acknowledgements

We would like to thank the referees for their valuable comments. This research was partly supported by the EPSRC projects "Foundational Structures in Computer Science", "Typed Concurrent Object-Oriented Languages: foundations, methods and tools" and the ESPRIT BRA 6454 (CONFER). The second author was also supported by the ESPRIT BRA 9102 (Coordination), and a grant from the Nuffield Foundation. The third author was also funded by the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Dolby Laboratories, Hewlett Packard, Hitachi, Hughes Space and Communications, LG Electronics, Lockheed Martin ATL, NEC, Philips, and Rockwell. Paul Taylor's commutative diagrams package was used in the production of the paper.

References

1. S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and foundations of typed concurrent programming. In M. Broy, editor, *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School*, NATO ASI Series F: Computer and Systems Sciences. Springer-Verlag, 1995.
2. S. Abramsky, S. J. Gay, and R. Nagarajan. Specification structures and propositions-as-types for concurrency. In G. Birtwistle and F. Moller, editors, *Logics for Concurrency: Structure vs. Automata—Proceedings of the VIIIth Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
3. S. Abramsky. Interaction Categories and communicating sequential processes. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 1–15. Prentice Hall International, 1994.
4. L. Aceto and M. Hennessy. Termination, deadlock and divergence. *Journal of the ACM*, 39:147–187, January 1992.
5. M. Barr. $*$-autonomous categories and linear logic. *Mathematical Structures in Computer Science*, 1(2):159–178, July 1991.
6. M. Berger, S. Gay, and R. Nagarajan. A typed calculus of deadlock-free processes. Paper in preparation, 1997.
7. S. D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13, pages 305–324. NATO Advanced Study Institutes, Series F, Springer-Verlag, Berlin, 1985.
8. K. M. Chandy and J. Misra. Deadlock absence proofs for networks of communicating processes. *Information Processing Letters*, 9(4), November 1979.
9. J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice Hall International, 1980.
10. E. W. Dijkstra and C. S. Scholten. A class of simple communication patterns. In *Selected Writings on Computing. EWD643*. Springer-Verlag, 1982.
11. S. J. Gay. *Linear Types for Communicating Processes*. PhD thesis, University of London, 1995.

12. S. J. Gay and R. Nagarajan. A typed calculus of synchronous processes. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1995.
13. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
14. R. Hoofman. *Non-Standard Models of Linear Logic*. PhD thesis, Universiteit Utrecht, Netherlands, 1992.
15. N. Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1997.
16. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings, 23rd ACM Symposium on Principles of Programming Languages*, 1996.
17. S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, Berlin, 1971.
18. J. McKinna and R. Burstall. Deliverables: A categorical approach to program development in type theory. In *Proceedings of Mathematical Foundation of Computer Science*, 1993.
19. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
20. R. Nagarajan. *Typed Concurrent Programs: Specification & Verification*. PhD thesis, University of London, 1997. To appear.
21. P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Proceedings, 20th ACM Symposium on Principles of Programming Languages*. ACM Press, 1993.
22. S. S. Owicki and D. Gries. Verifying properties of parallel programs. *Communications of the ACM*, 19(5):279–285, May 1976.
23. A. M. Pitts. Relational properties of recursively defined domains. In *8th Annual Symposium on Logic in Computer Science*, pages 86–97. IEEE Computer Society Press, Washington, 1993.
24. A. W. Roscoe and N. Daithi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, December 1987.
25. J. Sifakis. Deadlocks and livelocks in transition systems. In *Mathematical Foundations of Computer Science*, volume 88 of *Lecture Notes in Computer Science*, pages 587–599. Springer-Verlag, Berlin, 1980.
26. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th European Conference on Parallel Languages and Architectures*, number 817 in Lecture Notes in Computer Science. Springer-Verlag, 1994.
27. W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.