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

SAMSON ABRAMSKY
University of Edinburgh

SIMON GAY
Royal Holloway, University of London

RAJAGOPAL NAGARAJAN
University of California, Berkeley
&
Imperial College, London

Abstract

We present a type-based technique for the verification of deadlock-freedom in asynchronous concurrent systems. Our approach is to start with an interaction category such as $\mathcal{ASProc}$, where 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. The extra information in this case concerns deadlock-freedom, and in the resulting category $\mathcal{ASProc}_D$, 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.

Address for Correspondence

Dr S. J. Gay
Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey TW20 0EX, UK
Tel: +44 1784 443429
Fax: +44 1784 439786
Email: `S.Gay@dcs.rhbnc.ac.uk`

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.

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 [18] and verifying that it is deadlock-free.

2 The Interaction Category $\mathcal{ASProc}$

In this section we briefly review those parts of the definition of $\mathcal{ASProc}$ which are relevant to the present paper. This includes the $*$-autonomous structure [5], corresponding to the multiplicative connectives $\otimes$, $\mathbin{⅋}$ and $(-)^{\perp}$ of linear logic [12], but not the additive structure (products and coproducts) or the delay operator. Complete definitions can be found in previous publications [1, 2, 3, 10]. Our use of $\mathcal{ASProc}$ 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, [11]) which assumes universal synchronisation with respect to a global clock.

An object of $\mathcal{ASProc}$ 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 $\mathsf{ObAct}(A) \stackrel{\text{def}}{=} \Sigma_A - \{\tau_A\}$, $S_A \subseteq^{\mathit{nepref}} \mathsf{ObAct}(A)^*$ (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 an observation equivalence class of labelled transition systems [18] with label set Σ. 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

$$\begin{array}{rcl} \mathsf{allobtraces}(P) & \stackrel{\text{def}}{=} & \{\varepsilon\} \cup \{as \mid P \stackrel{a}{\Longrightarrow} Q, s \in \mathsf{allobtraces}(Q)\} \\ \mathsf{obtraces}(P) & \stackrel{\text{def}}{=} & \{s \in \mathsf{allobtraces}(P) \mid s \text{ is finite}\} \\ \mathsf{infobtraces}(P) & \stackrel{\text{def}}{=} & \{s \in \mathsf{allobtraces}(P) \mid s \text{ is infinite}\} \end{array}$$

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

$$\begin{array}{rclcrcl} \Sigma_{A\otimes B} & \stackrel{\text{def}}{=} & \Sigma_A \times \Sigma_B & \qquad & \tau_{A\otimes B} & \stackrel{\text{def}}{=} & (\tau_A, \tau_B) \\ S_{A\otimes B} & \stackrel{\text{def}}{=} & \multicolumn{5}{l}{\{s \in \mathsf{ObAct}(\Sigma_{A\otimes B})^* \mid s{\upharpoonright}A \in S_A, s{\upharpoonright}B \in S_B\}} \end{array}$$

where, for $\alpha \in \mathsf{ObAct}(\Sigma_{A\otimes B})$, $\alpha{\upharpoonright}A \stackrel{\text{def}}{=} \begin{cases} \mathsf{fst}(\alpha) & \text{if } \mathsf{fst}(\alpha) \neq \tau_A \\ \varepsilon & \text{otherwise} \end{cases}$
and for $s \in \mathsf{ObAct}(\Sigma_{A\otimes B})^*$, $s{\upharpoonright}A$ is obtained by concatenating the individual $\alpha{\upharpoonright}A$. The projection $\sigma{\upharpoonright}B$ 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 $\mathcal{ASProc}$ do not distinguish between input and output; later in the paper, this distinction will be recovered.

A morphism $p : A \to B$ of $\mathcal{ASProc}$ is a process p such that $p : A \multimap B$. In general, $A \multimap B \stackrel{\text{def}}{=} A \otimes B^{\perp\perp}$. In $\mathcal{ASProc}$, 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.

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

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 $\mathcal{ASProc}$ 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 $\mathcal{ASProc}$; the two processes can make transitions either independently or simultaneously.

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

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

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 $\mathcal{ASProc}$ 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

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

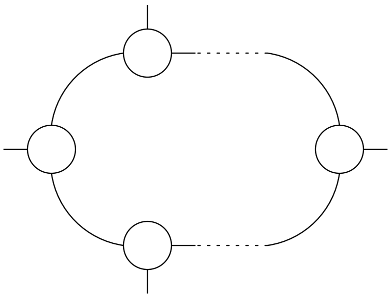
Currying is similarly defined by relabelling: if $f : A \otimes B \to C$ then $\Lambda(f) : A \to (B \multimap C)$ is defined by

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

Proposition 2.1 *$\mathcal{ASP}roc$ is a $*$-autonomous category.*

Because the linear negation $(-)^{\perp}$ is trivial in $\mathcal{ASP}roc$, $\otimes$ and $\invamp$ coincide. A $*$-autonomous category in which $\otimes$ and $\invamp$ 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 $\invamp$ coincide.



This point is discussed in more detail elsewhere [2]. Later in the present paper 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 [18]. A formal calculus of typed asynchronous processes (with a weaker type system which does not guarantee deadlock-freedom) has been developed [19], and this calculus will eventually be combined with 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 [17], Hoofman [13], O'Hearn and Tennent [20] and Pitts [22].

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

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

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" [8]. Given $\mathbb{C}$ and S as above, we can define a new category $\mathbb{C}_S$ as follows. The objects are pairs (A, φ) with $A \in \mathsf{ob}\,\mathbb{C}$ and $\varphi \in P_S A$. A morphism $f : (A, \varphi) \to (B, \psi)$ is a morphism $f : A \to B$ in $\mathbb{C}$ such that $\varphi\{f\}\psi$.

Composition and identities are inherited from $\mathbb{C}$; the axioms (1) and (2) ensure that $\mathbb{C}_S$ is a category. Moreover, there is an evident faithful functor $\mathbb{C}_S \rightarrowtail \mathbb{C}$ 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 $\mathbb{C}$, the set of properties $P_S X$ over each object X, and the Hoare triple relation.

1. $\mathbb{C} = \mathcal{S}et$, $P_S X = X$, $a\{f\}b \overset{\text{def}}{\Leftrightarrow} f(a) = b$.

   In this case, $\mathbb{C}_S$ is the category of pointed sets.

2. $\mathbb{C} = \mathcal{R}el$, $P_S X = \{*\}$, $*\{R\}* \overset{\text{def}}{\Leftrightarrow} \forall x \in A, y, z \in B.(xRy \wedge xRz \Rightarrow y = z)$.

   Then $\mathbb{C}_S$ is the category of sets and partial functions.

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

$$\begin{array}{rrcl} \mathsf{assoc}_{A,B,C}: & (A \otimes B) \otimes C & \cong & A \otimes (B \otimes C) \\ \mathsf{unitl}_A: & I \otimes A & \cong & A \\ \mathsf{unitr}_A: & A \otimes I & \cong & A \end{array}$$

satisfying the standard coherence equations [16]. A specification structure for $\mathbb{C}$ 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:

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

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

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

The axioms ensure that the conditions for a monoidal product are satisfied in $\mathbb{C}_S$. The action of $\otimes$ on morphisms in $\mathbb{C}_S$ is the same as in $\mathbb{C}$, 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 $\mathbb{C} \leftarrow \mathbb{C}_S$.

In the present paper we are interested in specification structures over $*$-autonomous categories, in particular $\mathcal{ASProc}$. If $\mathbb{C}$ 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 $\mathbb{C}$. This sequence will be used in the next Section when defining specification structures over $\mathcal{ASProc}$; it mirrors the sequence already used in the definition of $\mathcal{ASProc}$ 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 $⅋_{A,B}$ and $\multimap_{A,B}$.
5. Define the Hoare triple relation by $\theta\{f\}\varphi \overset{\text{def}}{\Leftrightarrow} f \models_{A \multimap B} \theta \multimap \varphi$.

6. Verify that the desired structure of $\mathbb{C}$, including the $*$-autonomous structure, lifts to $\mathbb{C}_{\mathcal{S}}$.

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

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

Such a tower arises by progressively refining $\mathbb{C}_0$ by specification structures $S_1, \ldots, S_k$ so that $\mathbb{C}_{i+1} = (\mathbb{C}_i)_{S_{i+1}}$. Each such step adds propositional information to the underlying "raw" computational entities (morphisms of $\mathbb{C}_0$). The aim of verification in this framework is to "promote" a morphism from $\mathbb{C}_i$ to $\mathbb{C}_j$, where $i < j$. That is, to promote a $\mathbb{C}_0$ morphism $f : A \to B$ to a $\mathbb{C}_k$ 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 $\mathbb{C}_k$ 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 $\mathcal{ASProc}_V$

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

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

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

The linear type structure of $\mathcal{ASProc}_V$ is defined as for $\mathcal{ASProc}$, with the addition of

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

As suggested by the name, $\mathcal{ASProc}_V$ can be defined by means of a specification structure V over $\mathcal{ASProc}$. However, the notation is easier to control if $\mathcal{ASProc}_V$ 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 $S_I = \{\epsilon\}$, 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, 10]), 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 $\epsilon \in V_A$ and $\epsilon \in V_C$, hence $\epsilon \in V_{A \multimap C}$ and $P\,;Q$ is interpreted as a successfully terminated process.

4.2 The Specification Structure 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.

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

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

$U^{\perp}$ 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 $\mathcal{ASProc}_V$ is obtained from the following sequence of definitions.

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

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 $f \models U^{\perp} \invamp V$ and $g \models V^{\perp} \invamp W$. 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 $\mathcal{ASProc}_V$ by means of the specification structure D is our category of deadlock-free processes. For consistency it should be called $\mathcal{ASProc}_{VD}$, but we will refer to it as $\mathcal{ASProc}_D$. The multiplicative structure of $\mathcal{ASProc}_V$ can be lifted to the new category, and $\mathcal{ASProc}_D$ is $*$-autonomous.

It is useful to have a collection of standard deadlock-free types over each $\mathcal{ASProc}_V$ type. Two obvious types, which we will now define, correspond to *input* and *output*. For each type A of $\mathcal{ASProc}_V$, 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 $\mathcal{ASProc}_D$ 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

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

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

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

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

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

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).

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:

$$\frac{P : (A_1, \theta_1) \parr \cdots \parr (A_n, \theta_n) \parr (X, \varphi) \parr (X^\perp, \varphi^\perp)}{\bar{P} : (A_1, \theta_1) \parr \cdots \parr (A_n, \theta_n)}\ \mathsf{cycle}(P)$$

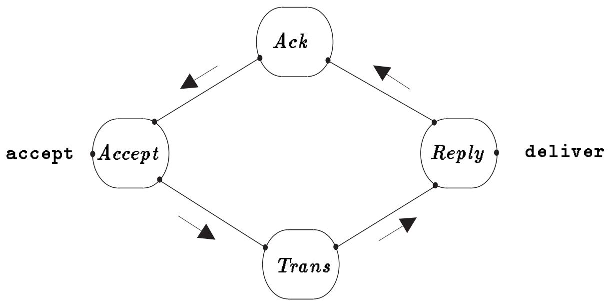
Here P is a process representing the acyclic portion of the desired configuration; it has two ports, of types (X, φ) and $(X^\perp, \varphi^\perp)$, which could potentially be connected together. The process $\bar{P}$ results from making that connection, but the condition $\mathsf{cycle}(P)$ must be satisfied if $\bar{P}$ is to be deadlock-free. The condition $\mathsf{cycle}(P)$ is defined by: whenever $P \stackrel{s}{\Longrightarrow} Q$ with $s \lceil X = s \lceil X^\perp$, and $Q \xrightarrow{(a_1,\ldots,a_n,x,y)} R$, there is a (possibly empty) sequence $Q_0 = Q, Q_1, \ldots, Q_m$ and actions $z_1, \ldots, z_n$ such that for each i, $Q_i \xrightarrow{(\tau_{A_1},\ldots,\tau_{A_n},z_i,z_i)} Q_{i+1}$ and then $Q_m \xrightarrow{(a_1,\ldots,a_n,z,z)} R$ with $(a_1, \ldots, a_n, z, z) \neq \tau$.

In any particular case, specific reasoning is needed to establish $\mathsf{cycle}(P)$ for the system in question. Once this has been done, the process $\bar{P}$ has a type in $\mathcal{ASP}roc_D$ and can be combined with other processes on the basis of that type.

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 following diagram *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 [18], 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.

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

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, and we have analysed similar specifications of other systems [1, 10, 19], but it is not the purpose of the present paper to illustrate this aspect of the theory.

5.2 Types in $\mathcal{ASP}roc_V$

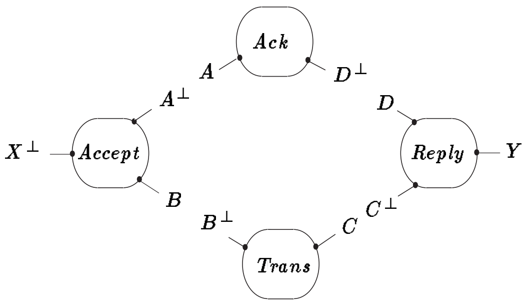
In order to type the system at the most elementary level, without taking account of deadlock-freedom, we need the following $\mathcal{ASP}roc_V$ types. In each case, the safety specification consists of all finite traces over the alphabet.

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

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 $Ack = (\tau_A, \mathtt{rec}_0).Ack_0 + (\tau_A, \mathtt{rec}_1).Ack_1$.

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



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 fact that $\mathcal{ASP}roc_V$ is compact closed allows the entire system to be constructed by means of categorical operations, despite the fact that it has a cyclic structure, yielding the process $\mathit{Protocol} : X^\perp \parr Y$. Because $X \cong Y$, we can also write $\mathit{Protocol} : X^\perp \parr X$ to emphasise the fact that the receiving and delivering ports are of compatible types.

5.3 Types in $\mathcal{ASProc}_D$

Because all of our types specify that every infinite trace is valid, and none of the processes can terminate, every process is convergent. 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 $\mathcal{ASProc}_D$ 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.

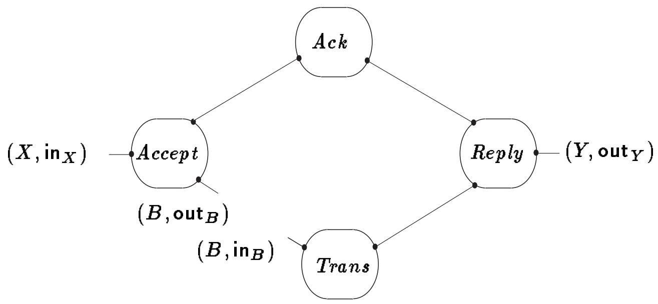
$$\begin{aligned} \mathit{Accept}: &\quad (X, \mathsf{in}_X) \parr (B, \mathsf{out}_B) \parr (A, \mathsf{in}_A) \\ \mathit{Trans}: &\quad (B, \mathsf{in}_B) \parr (C, \mathsf{out}_C) \\ \mathit{Reply}: &\quad (Y, \mathsf{out}_Y) \parr (C, \mathsf{in}_C) \parr (D, \mathsf{out}_D) \\ \mathit{Ack}: &\quad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

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

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

This requires $\mathit{Trans} \perp P \otimes \mathsf{max}_C$ for any process P of type B. This means that whenever Trans is capable of performing an action in its B port, it must offer a choice of all possible actions; however, there is no restriction on the possible actions in the C port. We can easily see that this is true, by examining the definition of *Trans*. Similar arguments apply to the other components of the system.

The $*$-autonomous structure of $\mathcal{ASProc}_D$ allows the components to be connected together, essentially by using categorical composition, up to the point at which the cycle is about to be completed. This gives the following process, *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 $\mathcal{ASProc}_D$ 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 $\mathcal{ASProc}_D$.

The verification is the same as we might carry out even without types, to check that the system does not deadlock. 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.

Once we know that *Protocol* does not deadlock, the actual condition $\mathsf{cycle}(PreProtocol)$ can be checked rather easily. We need to be sure that completing the cycle does not change the actions which are available in the remaining part of the interface, i.e. the X and Y ports. Because X has only one observable action, $\mathsf{in}_X = \mathsf{out}_X$, which means that $\mathsf{in}_X \parr \mathsf{out}_X = \mathsf{out}_X \parr \mathsf{out}_X = \mathsf{out}_{X \parr X}$. *Protocol* must satisfy this property, for general reasons.

In other examples, more work may be required to check the cycle condition. However, once a process has a type in $\mathcal{ASP}roc_D$, it 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.

6 Related Work

The analysis of deadlock-freedom in concurrency has been investigated by a number of authors [4, 6, 7, 9, 21, 23, 24]. The work of Roscoe and Daithi [23] 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 [25] have recently 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 [15] have developed a linear type system for the π-calculus. Recently Kobayashi [14] 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. Since our semantic types can also describe the order in which channels are used, there may be an interesting connection; however, we use a uniform and stronger definition of deadlock-freedom.

Besides conveying more information, our type system is flexible and modular, allowing for a range of concurrent program properties to be expressed. Also our treatment is more semantic, with a close connection via the Curry-Howard isomorphism to an underlying categorical logic. In previous work [2, 10], we have used techniques similar to those used in this paper to address deadlock-freedom. Most success has been obtained with synchronous systems; the theory described in the present paper provides for the first time a satisfactory treatment of deadlock-freedom in asynchronous systems.

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 syntax for deadlock freedom using the ideas from the typed process calculi that have been already developed [10, 11, 19]. 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 [23] may provide some clues. Another possibility is the cycle sum test of Wadge [26].

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 fairness and 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.

Acknowledgements

This research was partly supported by the EPSRC project "Foundational Structures in Computer Science", and the ESPRIT BRAs 6454 (CONFER) and 9102 (Coordination). 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] 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.

[7] K. M. Chandy and J. Misra. Deadlock absence proofs for networks of communicating processes. *Information Processing Letters*, 9(4), November 1979.

[8] J. W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice Hall International, 1980.

[9] E. W. Dijkstra and C. S. Scholten. A class of simple communication patterns. In *Selected Writings on Computing. EWD643.* Springer-Verlag, 1982.

[10] S. J. Gay. *Linear Types for Communicating Processes.* PhD thesis, University of London, 1995.

[11] 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.

[12] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[13] R. Hoofman. *Non-Standard Models of Linear Logic*. PhD thesis, Universiteit Utrecht, Netherlands, 1992.

[14] N. Kobayashi. A partially deadlock-free typed process calculus. Draft paper, 1996.

[15] 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.

[16] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, Berlin, 1971.

[17] J. McKinna and R. Burstall. Deliverables: A categorical approach to program development in type theory. In *Proceedings of Mathematical Foundation of Computer Science*, 1993.

[18] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[19] R. Nagarajan. *Typed Concurrent Programs: Specification & Verification*. PhD thesis, University of London, 1996. To appear.

[20] 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.

[21] S. S. Owicki and D. Gries. Verifying properties of parallel programs. *Communications of the ACM*, 19(5):279–285, May 1976.

[22] 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.

[23] A. W. Roscoe and N. Daithi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, December 1987.

[24] 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.

[25] 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.

[26] W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.