Typed Event Structures and the π-Calculus
– Extended Abstract –

Daniele Varacca, Nobuko Yoshida [1]

Imperial College London, UK

Abstract

We propose a typing system for the true concurrent model of event structures that guarantees the interesting behavioural properties known as *conflict freeness* and *confusion freeness*. Conflict freeness is the true concurrent version of the notion of confluence. A system is confusion free if nondeterministic choices are localised and do not depend on the scheduling of independent components. Ours is the first typing system to control behaviour in a true concurrent model. To demonstrate its applicability, we show that typed event structures give a semantics of linearly typed version of the π-calculi with internal mobility. The semantics we provide is the first event structure semantics of the π-calculus and generalises Winskel's original event structure semantics of CCS.

Key words: Event structures, types, linearity, π-calculus.

1 Introduction

Models for concurrency can be classified according to different criteria. One possible classification distinguishes between *interleaving* models and *causal* models (also known as *true concurrent* models). In interleaving models, concurrency is reduced to the nondeterministic choice between all possible sequential schedulings of the concurrent actions. Instances of such models are *traces* and *labelled transition systems* [35]. Interleaving models are very successful in defining observational equivalence, by means of bisimulation [20]. In causal models, causality and concurrency are explicitly represented. Instances of such models are *Petri nets* [25], *Mazurkiewicz traces* [18] and *event structures* [23]. True concurrent models can easily represent interesting behavioural properties such as absence of conflict, independence of the choices and sequentiality [25].

In this paper we address a particular true concurrent model: the model of *event structures* [23,32]. Event structures have been used to give semantics to

[1] Email: `varacca@doc.ic.ac.uk`, `yoshida@doc.ic.ac.uk`

concurrent process languages. The earliest and possibly the most intuitive is Winskel's semantics of Milner's CCS [31].

The first contribution of this paper is to present a compositional typing system for event structures that ensures two important behavioural properties: *conflict freeness* and *confusion freeness*.

Conflict freeness is the true concurrent version of confluence. In a conflict free system, the only nondeterminism allowed is due to the scheduling of independent components. To illustrate the less familiar notion of confusion freeness, let us suppose that a system is composed of two processes P and Q. Suppose the system can reach a state where P has a choice between two different actions a_1, a_2, and where Q, independently, can perform action b. We say that such a state is *confused* if the occurrence of b changes the choices available to P (for instance by disabling a_2, or by enabling a third action a_3). Intuitively the choice of process P is not local to that process in that it can be influenced by an independent action. We say that the system is confusion free if none of its reachable states is confused.

Confusion freeness was first identified in the context the theory of Petri nets [25]. It has been studied in that context, in the form of free choice nets [11]. Confusion free event structures are also known as *concrete data structures* [4], and their domain-theoretic counterpart are the *concrete domains* [17]. Finally, confusion freeness has been recognised as an important property in the context of probabilistic models [27,1].

The typing system we present guarantees that all typable event structures are confusion free. A restricted form of typing guarantees the stronger property of conflict freeness.

The second contribution of this paper is to give the first direct event structure semantics of a fragment of the π-calculus [21]. Various causal semantics of the π-calculus exist [16,7,12,5,10,8], but none is given in terms of event structures. The technical difficulty in extending CCS semantics to the π-calculus lies in the handling of α-conversion, which is the main ingredient to represent dynamic creation of names. We are able to solve this problem for a restricted version of the π-calculus, a linearly typed version of Sangiorgi's πI-calculus [26,37]. This fragment is expressive enough to encode the typed λ-calculus (in fact, to encode it *fully abstractly* [37]). We argue that in this fragment, α-conversion need not be performed dynamically (at "run time"), but can be done during the typing (at "compile time"), by choosing in advance all the names that will be created during the computation. This is possible because the typing system guarantees that, in a sense, every process knows in advance which processes it will communicate with.

In addition, the derived semantics for the π-calculus preserves the intuitive notions of Winskel's original semantics of CCS: syntactic nondeterministic choice is modelled by *conflict*, prefix is modelled using *causality*, and parallel composition generates *concurrent* events. Moreover, since our semantics is given in terms of typed event structures, we obtain that all processes of this fragment are confusion free. Our typing system generalises an early idea by Milner, who devised a syntactic restriction of CCS (a kind of a typing system) that guarantees confluence of the interleaving semantics [20]. As a corollary of our work we show that a similar restriction applied to the π-calculus guarantees the property of conflict freeness.

The tight correspondence between the linear π-calculus and programming language semantics opens the door for event structure semantics to the λ-calculus and other functional and imperative languages.

Structure of the paper

Section 2 presents a linearly typed version of the πI-calculus. This section is inspired from [37], but our fragment is extended to allow nondeterministic choice. Section 3 introduces the basic definitions of event structures and defines formally the notion of confusion freeness. Section 4 presents our new typing system and an event structure semantics of the types. We then define a notion of typing of event structures by means of the morphisms of the category of event structures. Typed event structures are confusion free by definition. The main theorem of this section is that the parallel composition of typed event structures is again typed, and thus confusion free. Section 5 provides a sound event structure semantics of the typed πI-calculus. The main result of the section is that the semantic of a πI-calculus term is a typed event structure, and thus it is confusion free. Section 6 concludes with related and future works. Due to the space limitation, materials of an intermediate CCS-like language which is used to translate the π-calculus into the typed event structures are left to the full version [28]. Also the detailed definitions and all proofs can be found in the full version [28].

2 A linear version of the π-calculus

This section briefly summarises an extension of linear version of the π-calculus in [3] to non-determinism [36]. The reader may refer to [3,36] for a more detailed description and more examples.

2.1 Syntax and reduction

We assume the reader is familiar with the basic definitions of the π-calculus [21]. As anticipated, we consider a restricted version of the π-calculus, where only bound names are passed in interaction. The resulting calculus is called the πI-calculus in the literature [26] and has the same expressive power as the version with free name passing [37]. Syntactically we restrict an output to the form $(\boldsymbol{\nu}\,\tilde{y})\overline{x}(\tilde{y}).P$ (where $\tilde{y}$ represents a tuple of pairwise distinct names), which we henceforth write $\overline{x}(\tilde{y}).P$.

We consider a version of the calculus more general than the one presented in [3], in that both input and output are nondeterminstic. Nondeterministic input is called *branching*, and it is already present in [3], while nondeterministic linear output, called *selection*, is a novelty of this work. Branching is similar to the "case" construct and selection is "injection" in the typed λ-calculi; these constructs have been studied in other typed π-calculi [29].

The formal grammar of the calculus is defined below.

$$\begin{aligned} P ::= x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y_i}).P_i \;\mid\; \overline{x} \bigoplus_{i\in I} \mathtt{in}_i(\tilde{y_i}).P_i \\ \mid\; P \,|\, Q \;\mid\; (\boldsymbol{\nu}\, x)P \;\mid\; \mathbf{0} \;\mid\; !x(\tilde{y}).P \end{aligned}$$

The process $x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y_i}).P_i$ (resp. $\overline{x} \bigoplus_{i\in I} \mathtt{in}_i(\tilde{y_i}).P_i$) is a branching input (resp. selecting output), where I denotes a finite or countably infinite indexing set. $P \,|\, Q$ is a parallel composition, $(\boldsymbol{\nu}\, x)P$ is a restriction and $!x(\tilde{y}).P$ is a replicated input. We omit the empty tuple: for example, $\overline{x}$ stands for $\overline{x}()$. When the index in the branching or selection indexing set is a singleton we use the notation $x(\tilde{y}).P$ or $\overline{x}(\tilde{y}).P$; when it is binary, we use $x((\tilde{y_1}).P_1 \& (\tilde{y_2}).P_2)$ or $\overline{x}((\tilde{y_1}).P_1 \oplus (\tilde{y_2}).P_2)$. The bound/free names are defined as usual. We use $\equiv_\alpha$ and $\equiv$ for the standard $\alpha-$ and structural equivalences [21,3,37,15].

Processes where all selection indexing sets are singletons are called *deterministic*. Deterministic processes where also branching indexing sets are singletons are called *simple*.

The reduction semantics is as follows:

$$\begin{gathered} x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y_i}).P_i \mid \overline{x} \bigoplus_{j\in J} \mathtt{in}_j(\tilde{y_j}).Q_j \longrightarrow (\boldsymbol{\nu}\, \tilde{y}_h)(P_h \,|\, Q_h) \quad (h \in I \cap J) \\ !x(\tilde{y}).P \mid \overline{x}(\tilde{y}).Q \longrightarrow !x(\tilde{y}).P \mid (\boldsymbol{\nu}\, \tilde{y})(P \,|\, Q) \end{gathered}$$

closed under evaluation contexts and structural equivalence.

2.2 Types and typings

The linear type discipline restricts the behaviour of processes as follows.

(A) for each linear name there are a unique input and a unique output; and

(B) for each replicated name there is a unique stateless replicated input with zero or more dual outputs.

In the context of deterministic processes, the typing system guarantees confluence. We will see that in the presence of nondeterminism this typing system guarantees confusion freeness.

As an example for the first condition, let us consider:

$$Q_1 \stackrel{\text{def}}{=} \overline{a}.b \,|\, \overline{a}.c \,|\, a \qquad\qquad Q_2 \stackrel{\text{def}}{=} b.\overline{a} \,|\, c.\overline{b} \,|\, a.(\overline{c} \,|\, \overline{e})$$

Then Q_1 is not typable as a appears twice as output, while Q_2 is typable since each channel appears at most once as input and output. Typability of simple processes such as Q_2 offers only deterministic behaviour. However branching and selection can provide non-deterministic behaviour, preserving linearity:

$$Q_3 \stackrel{\text{def}}{=} \overline{a}.(b \oplus c) \mid a.(\overline{d} \mathbin{\&} \overline{e})$$

Q_3 is typable, and we have either $Q_3 \longrightarrow (b \mid \overline{d})$ or $Q_3 \longrightarrow (c \mid \overline{e})$. As an example of the second constraint, let us consider the following two processes:

$$Q_4 \stackrel{\text{def}}{=} !\, b.\overline{a} \mid !\, b.\overline{c} \qquad\qquad Q_5 \stackrel{\text{def}}{=} !\, b.\overline{a} \mid \overline{b} \mid !\, c.\overline{b}$$

Q_4 is untypable because b is associated with two replicators: but Q_5 is typable since, while output at b appears twice, a replicated input at b appears only once.

Channel types are inductively made up from type variables and action modes: the two *input modes* $\downarrow, \mathbf{!}$, and the two *output modes* $\uparrow, \mathbf{?}$. We let $p, p', \ldots$ denote modes. We define $\overline{p}$, the *dual* of p, by: $\overline{\downarrow} = \uparrow$, $\overline{\mathbf{!}} = \mathbf{?}$ and $\overline{\overline{p}} = p$. Then the syntax of types is given as follows:

$$\begin{array}{llllll} \sigma ::= & \&_{i\in I}\,(\tilde{\sigma}_i)^{\downarrow} & \mid \bigoplus_{i\in I}\,(\tilde{\sigma}_i)^{\uparrow} \mid & (\tilde{\sigma})^{\mathbf{!}} & \mid & (\tilde{\sigma})^{\mathbf{?}} \\ & \text{(branching)} & \text{(selection)} & \text{(offer)} & & \text{(request)} \\ \tau ::= & \sigma & \mid \quad \updownarrow & \text{(closed type)} & & \end{array}$$

where $\tilde{\sigma}$ is a tuple of types. We write $MD(\tau)$ for the outermost mode of τ. The *dual of* τ, written $\overline{\tau}$, is the result of dualising all action modes, with $\updownarrow$ being self-dual. A type environment Γ is a finite mapping from channels to channel types. Sometimes we will write $x \in \Gamma$ to mean $x \in Dom(\Gamma)$.

Types restrict the composability of processes: if P is typed under environment Γ_1, Q is typed under Γ_2 and if Γ_1, Γ_2 are "compatible", then a new environment $\Gamma_1 \odot \Gamma_2$ is defined, such that $P \mid Q$ is typed under $\Gamma_1 \odot \Gamma_2$. If the environments are not compatible, $\Gamma_1 \odot \Gamma_2$ is not defined and the parallel composition cannot be typed. Formally, we introduce a partial commutative operation $\odot$ on types, defined as follows:

$$\begin{array}{llll} (1) & \tau \odot \overline{\tau} = \updownarrow & & \text{with } MD(\tau) = \downarrow \\ (2) & \tau \odot \overline{\tau} = \overline{\tau}, & \tau \odot \tau = \tau & \text{with } MD(\tau) = ? \end{array}$$

Then, then environment $\Gamma_1 \odot \Gamma_2$ is defined homomorphically. Intuitively, the rules in (2) say that a server should be unique, but an arbitrary number of clients can request interactions. The rules in (1) say that once we compose input-output linear channels, the channel becomes uncomposable. Other compositions are undefined. The definitions (1) and (2) ensure the two constraints (A) and (B).

The rules defining typing judgments $P \triangleright \Gamma$ are identical to the affine π-calculus [3] except a straightforward modification to deal with the non-deterministic output. See Appendix A.

$$\overline{a}\bigoplus_{i\in I}(\tilde{y}_i).P_i \xrightarrow{\overline{a}\mathtt{in}_j(\tilde{y}_j)} P_j \qquad a\mathop{\&}_{i\in I}(\tilde{y}_i).P_i \xrightarrow{a\mathtt{in}_j(\tilde{y}_j)} P_j$$

$$!a(\tilde{y}).P \xrightarrow{a(\tilde{y})} P \mid !a(\tilde{y}).P \qquad \overline{a}(\tilde{y}).P \xrightarrow{\overline{a}(\tilde{y})} P$$

$$\frac{P \xrightarrow{\beta} P'}{P \mid Q \xrightarrow{\beta} P' \mid Q} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q' \quad obj(\alpha) = \tilde{y}}{P \mid Q \xrightarrow{\alpha\bullet\beta} (\boldsymbol{\nu}\,\tilde{y})(P' \mid Q')}$$

$$\frac{P \xrightarrow{\beta} P' \quad subj(\beta) \neq x}{(\boldsymbol{\nu}\,x)P \xrightarrow{\beta} (\boldsymbol{\nu}\,x)P'} \qquad \frac{P \equiv_\alpha P' \quad P \xrightarrow{\beta} Q}{P' \xrightarrow{\beta} Q}$$

Fig. 1. Labelled Transition System for the πI-Calculus

2.3 A typed labelled transition relation

Typed transitions describe the observations a typed observer can make of a typed process. The typed transition relation is a proper subset of the untyped transition relation, while not restricting $\boldsymbol{\tau}$-actions: hence typed transitions restrict observability, not computation.

Labels are generated by the following grammar:

$$\begin{array}{lcccc} \alpha,\beta ::= & x\mathtt{in}_i(\tilde{y}) \mid & \overline{x}\mathtt{in}_i(\tilde{y}) \mid & x(\tilde{y}) \mid & \overline{x}(\tilde{y}) \\ & \text{(branching)} & \text{(selection)} & \text{(offer)} & \text{(request)} \\ \boldsymbol{\tau} ::= & (x,\overline{x})\mathtt{in}_i(\tilde{y}) \mid & (x,\overline{x})(\tilde{y}) & \text{(synchronisation)} & \end{array}$$

With the notation above, we say that x is the *subject* of the label β, denoted as $subj(\beta)$, while $\tilde{y} = y_1,\ldots,y_n$ are the *object* names, denoted as $obj(\beta)$. For branching/selection labels, the index i is the *branch* of the label. The notation "$\mathtt{in}_i$" comes from the injection of the typed λ-calculus. The partial operation $\alpha\bullet\beta$ is defined as follows: $x\mathtt{in}_i(\tilde{y}_i)\bullet\overline{x}\mathtt{in}_i(\tilde{y}_i) = (x,\overline{x})\mathtt{in}_i(\tilde{y}_i)$, $x(\tilde{y})\bullet\overline{x}(\tilde{y}) = (x,\overline{x})(\tilde{y})$, and undefined otherwise.

The standard untyped transition relation is defined in Figure 1. We define the predicate "Γ allows β" which represents how an environment restricts observability: for all Γ, Γ allows $\boldsymbol{\tau}$; if $MD(\Gamma(x)) = \downarrow$, then Γ allows $x\mathtt{in}_i(\tilde{y})$; and if $MD(\Gamma(x)) = \mathbf{!}$, then Γ allows $x(\tilde{y})$. The cases $MD(\Gamma(x)) = \uparrow, \mathbf{?}$ are defined dually. Intuitively, labels only allowed when the type environment is coherent with them.

Whenever Γ allows β, we define a new environment $\Gamma\setminus\beta$ to represent what remains after the transition labelled by β has happened. Linear channels are consumed, while replicated channels are not consumed. The new previously bound channels are released. For instance, if $\Gamma = \Delta, x : \mathop{\&}_{i\in I}(\tilde{\tau}_i)^{\downarrow}$, then $\Gamma\setminus x\mathtt{in}_i(\tilde{y}) = \Delta, \tilde{y} : \tilde{\tau}$. Then the typed transition, written $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma'$ is defined by adding the constraint:

$$\text{if } P\xrightarrow{\beta}Q \text{ and } \Gamma \text{ allows } \beta \quad \text{then} \quad P \triangleright \Gamma\xrightarrow{\beta}Q \triangleright \Gamma \setminus \beta$$

The above rule does not allow a linear input action and an output action when there is a complementary channel in the process. For example, if a process has $x:(\tilde{\tau})^{\mathsf{!}}$ in its action type, then output at x is excluded since such actions can never be observed in a typed context – cf. [3]. For a concrete example, consider the process $\overline{a}.b \mid \overline{b}.a$ which is typed in the environment $a:\updownarrow, b:\updownarrow$. Although the process has some untyped transitions, none of them is allowed by the environment.

By induction on the rules in Figure 1, we can obtain:

Proposition 2.1 (i) *If $P \triangleright \Gamma$, $P\xrightarrow{\beta}Q$ and Γ allows β, then $Q \triangleright \Gamma \setminus \beta$.*

(ii) (Subject reduction) *If $P \triangleright \Gamma$ and $P\xrightarrow{\tau}Q$, then $Q \triangleright \Gamma$.*

(iii) (Church Rosser for deterministic processes) *Suppose $P \triangleright \Gamma$ and P is deterministic. Assume $P\xrightarrow{\tau}Q_1$, and $P\xrightarrow{\tau}Q_2$. Then $Q_1 \equiv_\alpha Q_2$ or there exists R such that $Q_1\xrightarrow{\tau}R$ and $Q_2\xrightarrow{\tau}R$.*

Finally we define the notion of typed bisimulation. Let $\mathscr{R}$ be a symmetric relation between judgments such that if $(P \triangleright \Gamma) \mathscr{R} (P' \triangleright \Gamma')$, then $\Gamma = \Gamma'$. We say that $\mathscr{R}$ is a bisimulation if the following is satisfied:

- whenever $(P \triangleright \Gamma) \mathscr{R} (P' \triangleright \Gamma)$, $P \triangleright \Gamma\xrightarrow{\beta}Q \triangleright \Gamma \setminus \beta$, then there exists Q' such that $P' \triangleright \Gamma\xrightarrow{\beta}Q' \triangleright \Gamma \setminus \beta$, and $(Q \triangleright \Gamma \setminus \beta) \mathscr{R} (Q' \triangleright \Gamma \setminus \beta)$.

If there exists a bisimulation between two judgments, we say that they are bisimilar $(P \triangleright \Gamma) \approx (P' \triangleright \Gamma)$. Note that $\approx$ is a congruent relation [28].

3 Event structures

Event structures were introduced by Nielsen, Plotkin and Winskel [23,30], and have been subject of several studies since. They appear in different forms. The one we introduce in this work is sometimes referred to as *prime event structures* [32]. For the relations of event structures with other models for concurrency, the standard reference is [35].

3.1 Basic definitions

An *event structure* is a triple $\mathscr{E} = \langle E, \leq, \smile \rangle$ such that

- E is a countable set of *events*;
- $\langle E, \leq \rangle$ is a partial order, called the *causal order*;
- for every $e \in E$, the set $[e) := \{e' \mid e' < e\}$, called the *enabling set* of e, is finite;
- $\smile$ is an irreflexive and symmetric relation, called the *conflict relation*, satisfying the following: for every $e_1, e_2, e_3 \in E$ if $e_1 \leq e_2$ and $e_1 \smile e_3$ then $e_2 \smile e_3$.

The reflexive closure of conflict is denoted by $\asymp$. We say that the conflict $e_2 \smile e_3$ is *inherited* from the conflict $e_1 \smile e_3$, when $e_1 < e_2$. If a conflict $e_1 \smile e_2$ is not inherited from any other conflict we say that it is *immediate*, denoted by $e_1 \smile_\mu e_2$. The reflexive closure of immediate conflict is denoted by $\asymp_\mu$. If two events are not causally related nor in conflict they are said to be *concurrent*. A *configuration* x of an event structure $\mathscr{E}$ is a conflict free downward closed subset of E, i.e. a subset x of E satisfying: (1) if $e \in x$ then $[e) \subseteq x$ and (2) for every $e, e' \in x$, it is not the case that $e \smile e'$. Therefore, two events of a configuration are either causally dependent or concurrent, i.e., a configuration represents a run of an event structure where events are partially ordered.

A *labelled event structure* is an event structure $\mathscr{E}$ together with a labelling function $\lambda : E \to L$, where L is a set of labels. Events should be thought of as occurrences of actions. Labels allow us to identify events which represent different occurrences of the same action. Labels are also essential in defining the parallel composition, and play a major role in the typed setting. A labelled event structure generates a labelled transition system as follows.

Definition 3.1 Let $\mathscr{E} = \langle E, \leq, \smile, \lambda \rangle$ be a labelled event structure and let e be one of its minimal events. The event structure $\mathscr{E}\lfloor e = \langle E', \leq', \smile', \lambda' \rangle$ is defined by: $E' = \{e' \in E \mid e' \not\asymp e\}$, $\leq' = \leq_{|E'}$, $\smile' = \smile_{|E'}$, and $\lambda' = \lambda_{E'}$.

Roughly speaking, $\mathscr{E}\lfloor e$ is $\mathscr{E}$ minus the event e, and minus all events that are in conflict with e. We can then generate a labelled transition system on event structures as follows: if $\lambda(e) = \beta$, then

$$\mathscr{E} \xrightarrow{\beta} \mathscr{E}\lfloor e\ .$$

The reachable transition system with initial state $\mathscr{E}$ is denoted as $TS(\mathscr{E})$.

3.2 Conflict free and confusion free event structures

Definition 3.2 An event structure is *conflict free* if its conflict relation is empty.

Conflict freeness is the true concurrent version of confluence. Indeed it is easy to verify that if $\mathscr{E}$ is conflict free, then $TS(\mathscr{E})$ is confluent.

As informally explained, in a confusion free event structure every conflict is *localised*. To specify what "local" means in this context, we need the notion of *cell*, a set of pairwise conflicting events with the same causal predecessors.

Definition 3.3 A *partial cell* is a set c of events such that $e, e' \in c$ implies $e \asymp_\mu e'$ and $[e) = [e')$. A maximal partial cell is called a *cell*.

In general, two events in immediate conflicts need not belong to the same cell. If a cell is thought of as a location, this means that not all conflicts are localised. This leads us to the following definition.

Definition 3.4 An event structure is *confusion free* if its cells are closed under immediate conflict.

Equivalently, in a confusion free event structure reflexive immediate conflict is an equivalence relation with cells as its equivalence classes [27].

3.3 Morphisms of event structures

Event structures form the class of objects of a category [35]. The morphisms are defined as follows. Let $\mathscr{E}_1 = \langle E_1, \leq_1, \smile_1 \rangle$, $\mathscr{E}_2 = \langle E_2, \leq_2, \smile_2 \rangle$ be two event structures. A *morphism* $f : \mathscr{E}_1 \to \mathscr{E}_2$ is a partial function $f : E_1 \to E_2$ such that

- f reflects causality: if $f(e_1)$ is defined, then $\big[f(e_1)\big) \subseteq f\big([e_1)\big)$;
- f reflects reflexive conflict: if $f(e_1), f(e_2)$ are defined, and if $f(e_1) \asymp f(e_2)$, then $e_1 \asymp e_2$.

Given two labelled event structures $\mathscr{E}_1 = \langle E_1, \leq_1, \smile_1, \lambda_1 \rangle$, $\mathscr{E}_2 = \langle E_2, \leq_2, \smile_2, \lambda_2 \rangle$ on the same set of labels L, a morphism $f : \mathscr{E}_1 \to \mathscr{E}_2$ is said to be *label preserving* if, whenever $f(e_1)$ is defined, $\lambda_2(f(e_1)) = \lambda_1(e_1)$.

3.4 Operators on event structures

We can define several operations on labelled event structures. We provide here an informal description of some of them. See [32] for more details.

- *Prefixing* $a.\mathscr{E}$. This is obtained by adding a new minimum event, labelled by a. Conflict, order, and labels remain the same on the old events.
- *Prefixed sum* $\sum_{i\in I} a_i.\mathscr{E}_i$. This is obtained by disjoint union of copies of the event structures $a_i.\mathscr{E}_i$, where the order relation is the disjoint union of the orders, the labelling function is the disjoint union of the labelling functions, and the conflict is the disjoint union of the conflicts extended by putting in conflict every two events in two different copies. It is a generalisation of prefixing, where we add an initial *cell*, instead of an initial event.
- *Restriction* $\mathscr{E} \setminus X$ where $X \subseteq A$ is a set of labels. This is obtained by removing from E all events with label in X and all events that are above one of those. On the remaining events, order, conflict and labelling are unchanged.
- *Relabelling* $\mathscr{E}[f]$. This is just composing the labelling function λ with a function $f : L \to L$. The new event structure has thus labelling function $f \circ \lambda$.

All these constructions preserve the class of confusion free event structures. Also, with the exception of the prefixed sum, they preserve the class of conflict free event structures

3.5 The parallel composition

The parallel composition of event structures is defined in [35] as the categorical product followed by restriction and relabelling. Although the categorical product of two event structures is unique up to isomorphism, it can be explicitly constructed in different ways. We provide here a brief outline of one such construction [28,9]. Let $\mathscr{E}_1 := \langle E_1, \leq_1, \smile_1 \rangle$ and $\mathscr{E}_2 := \langle E_2, \leq_2, \smile_2 \rangle$ be two event structures. Let $E_i^* := E_i \uplus \{*\}$. Consider the set $\tilde{E}$ obtained as the initial solution of the equation $X = \mathscr{P}_{fin}(X) \times E_1^* \times E_2^*$. Its elements have the form (x, e_1, e_2) for x finite, $x \subseteq \tilde{E}$. We define a set $E \subseteq \tilde{E}$, an order $\leq$ and a conflict relation $\smile$ on E, such that $\mathscr{E} = \langle E, \leq, \smile \rangle$ is an event structure. This is the categorical product of $\mathscr{E}_1, \mathscr{E}_2$, with the projections defined as $\pi_1(x, e_1, e_2) = e_1$ and $\pi_2(x, e_1, e_2) = e_2$.

For event structures with labels in L, the labelling function of the product takes on the set $L_* \times L_*$, where $L_* := L \uplus \{*\}$. We define $\lambda(x, e_1, e_2) = (\lambda_1^*(e_1), \lambda_2^*(e_2))$, where $\lambda_i^*(e_i) = \lambda_i(e_i)$ if $e_i \neq *$, and $\lambda_i^*(*) = *$. A *synchronisation algebra* S is given by a partial binary operation $\bullet_S$ defined on L_* [35]. Given two labelled event structures $\mathscr{E}_1, \mathscr{E}_2$, the parallel composition $\mathscr{E}_1 \|_S \mathscr{E}_2$ is defined as the categorical product followed by restriction and relabelling: $(\mathscr{E}_1 \times \mathscr{E}_2 \setminus X)[f]$ where X is the set of pairs $(l_1, l_2) \in L_* \times L_*$ for which $l_1 \bullet_S l_2$ is undefined, while the function f : is defined as $f(l_1, l_2) = l_1 \bullet_S l_2$. The subscripts S are omitted when the synchronisation algebra is clear from the context.

The simplest possible synchronisation algebra is defined as $l \bullet * = * \bullet l = l$, and undefined in all other cases. In this particular case, the induced parallel composition can be represented as the disjoint union of the sets of events, of the causal orders, and of the conflict. This can be also generalised to an arbitrary family of event structures $(\mathscr{E}_i)_{i \in I}$. In such a case we denote the parallel composition as $\prod_{i \in I} \mathscr{E}_i$.

Parallel composition does not preserve in general the classes of conflict free and confusion free event structures. New conflicts can be created through synchronisation. One of the main reasons to devise a typing system for event structures is to guarantee the preservation of these two important behavioural properties.

3.6 Examples

We collect in this section a series of examples, with graphical representation.

Example 3.5 Consider the following event structures $\mathscr{E}_1, \mathscr{E}_2, \mathscr{E}_3$, defined on the same set of events $E := \{a, b, c, d, e\}$. In $\mathscr{E}_1$, we have $a \leq b, c, d, e$ and $b \smile_\mu c$, $c \smile_\mu d$, $b \smile_\mu d$. In $\mathscr{E}_2$, we do not have $a \leq d$, while in $\mathscr{E}_3$, we do not have $b \smile_\mu d$. The three event structures are represented in Figure 2, where curly lines represent immediate conflict, while the causal order proceeds upwards along the straight lines.

The event structure $\mathscr{E}_1$ is confusion free, with three cells: $\{a\}, \{b, c, d\}, \{e\}$. In $\mathscr{E}_2$, there are four cells: $\{a\}, \{b, c\}, \{d\}, \{e\}$. $\mathscr{E}_2$ is not confusion free, because immediate conflict is not cellular. This is an example of *asymmetric* confusion [24]. In $\mathscr{E}_3$ there are four cells: $\{a\}, \{b, c\}, \{c, d\}, \{e\}$. $\mathscr{E}_3$ is not confusion free, because immediate conflict is not transitive. This is an example of *symmetric* confusion.

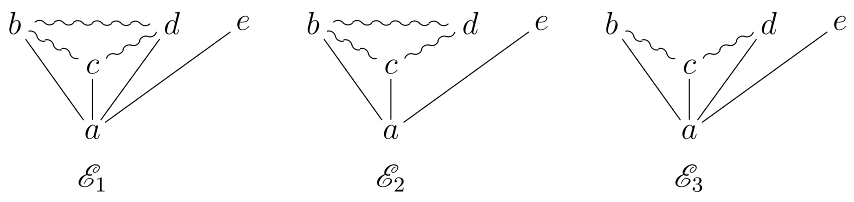

Fig. 2. Event structures

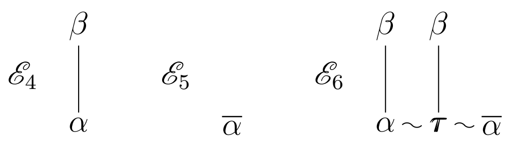

Fig. 3. Parallel composition of event structures

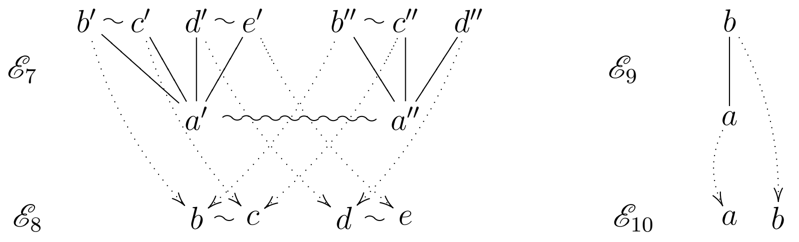

Fig. 4. Morphisms of event structures

Example 3.6 Next we show an example of parallel composition, see Figure 3. Consider the two labelled event structures $\mathscr{E}_4, \mathscr{E}_5$, where $E_4 = \{a, b\}, E_5 = \{a'\}$, conflict and order being trivial, and $\lambda(a) = \alpha, \lambda(b) = \beta, \lambda(a') = \overline{\alpha}$. Consider the symmetric synchronisation algebra $\alpha \bullet \overline{\alpha} = \boldsymbol{\tau}$, $\alpha \bullet * = \alpha$, $\overline{\alpha} \bullet * = \overline{\alpha}$, $\beta \bullet * = \beta$ and undefined otherwise. Then $\mathscr{E}_6 := \mathscr{E}_4 \| \mathscr{E}_5$ is as follows: $E_6 = \{e := (\emptyset, a, *), e' := (\emptyset, *, a'), e'' := (\emptyset, a, a'), d := (\{e\}, a', *), d'' := (\{e''\}, a', *)\}$, with the ordering defined as $e \leq d, e'' \leq d''$, while the conflict is defined as $e \smile e''$, $e' \smile e''$, $e \smile d''$, $e' \smile d''$, $e'' \smile d$, $d \smile d''$. The labelling function is $\lambda(e) = \alpha, \lambda(e') = \overline{\alpha}, \lambda(e'') = \boldsymbol{\tau}, \lambda(d) = \lambda(d'') = \beta$. Note that, while $\mathscr{E}_4, \mathscr{E}_5$ are confusion free, $\mathscr{E}_6$ is not, since reflexive immediate conflict is not transitive.

Example 3.7 Finally we show an example of morphism. Consider the two event structures $\mathscr{E}_7, \mathscr{E}_8$ defined as follows:

- $E_7 = \{a', b', c', d', e', a'', b'', c'', d''\}$ with $a' \smile_\mu a''$ $b' \smile_\mu c', d' \smile_\mu e', b'' \smile_\mu c''$ and $a' \leq b', c', d', e'$ and $a'' \leq b'', c'', d''$.
- $E_8 = \{b, c, d, e\}$ with $b \smile_\mu c, d \smile_\mu e$, and trivial ordering.

Note that both $\mathscr{E}_7$ and $\mathscr{E}_8$ are confusion free.

We define a morphism $f : \mathscr{E}_7 \to \mathscr{E}_8$ by putting $f(x') = f(x'') = x$ for $x = b, c, d, e$ while f is undefined on a', a''. Note that b' and b'' are mapped to the same element b, and they are indeed in conflict, because they inherit the conflict $a' \smile a''$.

For another example consider the two event structures $\mathscr{E}_9, \mathscr{E}_{10}$, where $E_9 = E_{10} = \{a, b\}$, both have empty conflict, and in $\mathscr{E}_9$ we have $a \leq b$. The identity function on $\{a, b\}$ is a morphism $\mathscr{E}_9 \to \mathscr{E}_{10}$ but not vice versa. We can say that the causal order of $\mathscr{E}_9$ refines the causal order of $\mathscr{E}_{10}$.

4 Typed event structures

In this section we present a notion of types for an event structure, which are inspired from the types for the linear π-calculus. Every such type is represented by an event structure which interprets the causality between the names contained in the type. We then assign types to event structures by allowing a more general notion of causality.

4.1 Types and environments

Types, type environmentsare generated by the following grammar

$$\begin{array}{rcl} \Gamma, \Delta & ::= & y_1 : \sigma_1, \ldots, y_n : \sigma_n \qquad \text{(type environment)} \\ \tau, \sigma & ::= & \&_{i\in I}\,\Gamma_i \quad | \quad \bigoplus_{i\in I}\Gamma_i \quad | \bigotimes_{i\in I}\Gamma_i \mid \biguplus_{i\in I}\Gamma_i \quad | \qquad \updownarrow \\ & & \text{(branching)} \quad \text{(selection)} \quad \text{(offer)} \quad \text{(request)} \quad \text{(closed type)} \end{array}$$

A type environment Γ is *well formed* if any name appears at most once. Only well formed environments are considered for typing event structures. An environment can also be thought of as a partial function from names to types. In this view we can talk of *domain* and *range* of an environment.

Event structures types and environments are similar to those of the π-calculus, but they recursively keep track of the object names. Moreover server and client types explicitly represent each copy of the resource.

A notion of composition of environments is defined in a similar way to the π-calculus. The difference is that we not only match modes, but we recursively match the object names. As in the π-calculus, the composition is only partially defined. Given two type environments Γ_1, Γ_2 we denote their composition (when defined) as $\Gamma_1 \odot \Gamma_2$. See Appendix B for the formal definition.

$$[\![y_1 : \sigma_1, \ldots, y_n : \sigma_n]\!] = [\![y_1 : \sigma_1]\!] \| \ldots \| [\![y_n : \sigma_n]\!]$$

$$[\![x : \&_{i\in I} \Gamma_i]\!] = \textstyle\sum_{i\in I} x\mathtt{in}_i(\tilde{y}_i).[\![\Gamma_i]\!] \qquad [\![x : \bigoplus_{i\in I} \Gamma_i]\!] = \textstyle\sum_{i\in I} \overline{x}\mathtt{in}_i(\tilde{y}_i).[\![\Gamma_i]\!]$$

$$[\![x : \bigotimes_{i\in I} \Gamma_i]\!] = \textstyle\prod_{i\in I} x(\tilde{y}_i).[\![\Gamma_i]\!] \qquad [\![x : \biguplus_{i\in I} \Gamma_i]\!] = \textstyle\prod_{i\in I} \overline{x}(\tilde{y}_i).[\![\Gamma_i]\!]$$

$$[\![x : \updownarrow]\!] = \emptyset$$

Fig. 5. Denotational semantics of types

4.2 Semantic of types

Type environments are given a semantics in terms of labelled confusion free event structures.

The labels are the ones described in the Section 2. Labels can be *allowed* or *disallowed* by a type environments, similarly to the π-calculus case, but recursively considering the object names [28]. We denote by $\mathit{Dis}(\Gamma)$ the set of labels that are disallowed by the environment Γ.

The semantics is presented in Figure 5, where we assume that $\tilde{y}_i$ represents the sequence of names in the domain of Γ_i. A name used for branching/selection identifies a cell. A name used for offer/request identifies a "cluster" of parallel events.

The following result is a sanity check for our definitions. It shows that matching of types corresponds to parallel composition with synchronisation. The synchronisation algebra we use is the one defined in Section 2 extended with $\alpha \bullet * = * \bullet \alpha = \alpha$.

Proposition 4.1 *Take two environments Γ_1, Γ_2, and suppose $\Gamma_1 \odot \Gamma_2$ is defined. Then $([\![\Gamma_1]\!] \| [\![\Gamma_2]\!]) \setminus (\mathit{Dis}(\Gamma_1 \odot \Gamma_2) \cup \boldsymbol{\tau}) = [\![\Gamma_1 \odot \Gamma_2]\!]$.*

4.3 Typing event structures

Given a labelled confusion free event structure $\mathscr{E}$ on the same set of labels as above, we define when $\mathscr{E}$ is typed in the environment Γ, written as $\mathscr{E} \rhd \Gamma$. A type environment Γ defines a general behavioural pattern via its semantics $[\![\Gamma]\!]$. The intuition is that for an event structure $\mathscr{E}$ to have type Γ, $\mathscr{E}$ should follow the pattern of $[\![\Gamma]\!]$, possibly "refining" the causal structure of $[\![\Gamma]\!]$ and possibly omitting some of its actions.

Definition 4.2 We say that $\mathscr{E} \rhd \Gamma$, if the following conditions are satisfied:

- each cell in $\mathscr{E}$ is labelled by x, $\overline{x}$ or $(x, \overline{x})$, and labels of the events correspond to the label of their cell in the obvious way;
- there exists a label-preserving morphism of labelled event structures $f : \mathscr{E} \to [\![\Gamma]\!]$ such that $f(e)$ is undefined if and only if $\lambda(e) \in \boldsymbol{\tau}$.

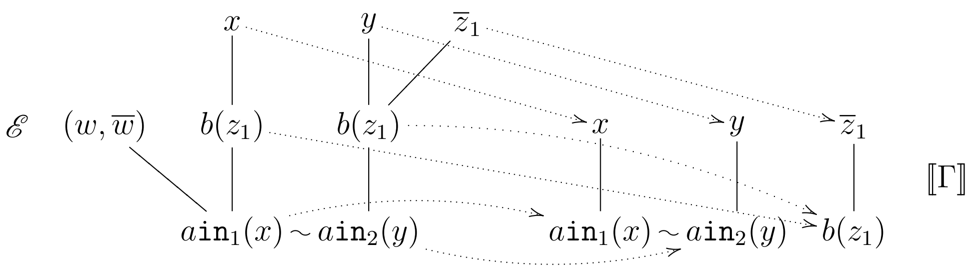

Fig. 6. Typed event structure

Roughly speaking a confusion free event structure $\mathscr{E}$ has type Γ if cells are partitioned into branching, selection, request, offer and synchronisation cells, all the non-synchronisation events of $\mathscr{E}$ are represented in Γ and causality in $\mathscr{E}$ refines causality in $[\![\Gamma]\!]$.

As we said, the parallel composition of confusion free event structures is not confusion free in general. The main result of this section shows that the parallel composition of typed event structures is still confusion free, and moreover is typed.

Theorem 4.3 *Take two labelled confusion free event structures $\mathscr{E}_1, \mathscr{E}_2$. Suppose $\mathscr{E}_1 \triangleright \Gamma_1$ and $\mathscr{E}_2 \triangleright \Gamma_2$. Assume $\Gamma_1 \odot \Gamma_2$ is defined. Then $(\mathscr{E}_1 \| \mathscr{E}_2) \setminus (Dis(\Gamma_1 \odot \Gamma_2))$ is confusion free and $(\mathscr{E}_1 \| \mathscr{E}_2) \setminus (Dis(\Gamma_1 \odot \Gamma_2)) \triangleright \Gamma_1 \odot \Gamma_2$.*

The proof relies on the fact that the typing system, in particular the uniqueness condition on well formed environments, guarantees that no new conflict is introduced through synchronisation.

Special cases are obtained when some or all cells are singletons. We call a typed event structure *deterministic* if its selection cells and its $\boldsymbol{\tau}$ cells are singletons. We call a typed event structure *simple* if all its cells are singletons. In particular, a simple event structure is conflict free.

Theorem 4.4 *Take two labelled deterministic (resp. simple) event structures $\mathscr{E}_1 \triangleright \Gamma_1$ and $\mathscr{E}_2 \triangleright \Gamma_2$. Suppose $\Gamma_1 \odot \Gamma_2$ is defined. Then $(\mathscr{E}_1 \| \mathscr{E}_2) \setminus Dis(\Gamma_1 \odot \Gamma_2)$ is deterministic (resp. simple).*

4.4 Examples

In the following, when the indexing set of a branching type is a singleton, we use the abbreviation $(\Gamma)^{\downarrow}$. Similarly, for a singleton selection type we write $(\Gamma)^{\uparrow}$. Also, when the indexing set of a type is $\{1,2\}$, we write $(\Gamma_1 \& \Gamma_2)$ or $(\Gamma_1 \otimes \Gamma_2)$.

Example 4.5 Consider the types $\tau_1 = (x : ()^{\downarrow} \ \& \ y : ()^{\downarrow}), \sigma_1 = \biguplus_{i\in\{2\}}(z_i : \updownarrow)$ $\tau_2 = (x : ()^{\uparrow} \oplus y : ()^{\uparrow}), \sigma_2 = \bigotimes_{i\in\{1,2,3\}}(z_i : \updownarrow)$. If we put $\Gamma_1 = a : \tau_1, b : \sigma_1$, and $\Gamma_2 = a : \tau_2, b : \sigma_2$, we have that $\Gamma_1 \odot \Gamma_2 = a : \updownarrow, b : \bigotimes_{i\in\{1,3\}}(z_i : \updownarrow)$.

Example 4.6 As an example of typed event structures, consider the environment $\Gamma = a : (x : ()^{\downarrow} \,\&\, y : ()^{\downarrow}), b : \biguplus_{i \in \{1\}} (z_i : ()^{\uparrow})$. Figure 6 shows an event structure $\mathscr{E}$, such that $\mathscr{E} \triangleright \Gamma$, together with a morphism $\mathscr{E} \to [\![\Gamma]\!]$. Note that the two events in $\mathscr{E}$ labelled with $b(z_1)$ are mapped to the same event and indeed they are in conflict.

5 Event structure semantics of the typed π-calculus

In this section we provide the event structure semantics of the π-calculus, and study some of its properties.

5.1 *Definition of the semantics*

The semantics is given by a family of partial functions $[\![-]\!]^{\Delta}$, parametrised by an event structure type environment Δ, that take a judgment of the π-calculus and return an event structure. The parameter is essentially providing a fixed choice for the object names. This parametrisation is necessary because π-calculus terms are identified up to α-conversion, and so the identity of (bound) object names is irrelevant, while in the typed event structures, the identity of object names is important.

The semantics is defined in Figure 7, by induction on the derivation of the typing judgment. In the semantics of the replicated input, we also need the following definitions. For any set K, let $K(x) := \{x^k \mid k \in K\}$ be a set of names such that for distinct x, y, $K(x) \cap K(y) = \emptyset$. Given a type τ, and an index $k \in K$, we write τ^k for the type obtained from τ by substituting y^k for every name y. Given an environment Γ, we define Γ^k to be such that for every $x \in \mathit{Dom}(\Gamma)$, $\Gamma^k(x) = \Gamma(x)^k$. Let Γ be an environment such that for every name $x \in \mathit{Dom}(\Gamma)$, $\Gamma(x) = \biguplus_{h \in H} \Delta_h$ The environment $\Gamma[K]$ is defined as follows: for every $x \in \mathit{Dom}(\Gamma)$, $\Gamma[K](x) = \biguplus_{(k,h) \in K \times H} \Delta_h^k$. If we assume that all names in $K(x)$ are fresh for Γ, we have that $\Gamma[K]$ is well formed.

Note in particular that in the parallel composition we restrict all names that are subject of communication, by restricting all names that are not allowed by the new type environment.

The interpretation functions are indeed partial functions: for the wrong choice of Δ_1, Δ_2, the interpretation of the parallel composition could be undefined, because $\Delta_1 \odot \Delta_2$ may be undefined. However it is always possible to find suitable Δ_1, Δ_2. Intuitively we can say that in interpreting the typed π-calculus into event structures, we perform α-conversion "at compile time".

Theorem 5.1 *For every judgment $P \triangleright \Gamma$ in the π-calculus, there exists an environment Δ such that $[\![P \triangleright \Gamma]\!]^{\Delta}$ is defined.*

Example 5.2 We demonstrate how the process which generates an infinite behaviour with infinite new name creation is interpreted into the event structures. Consider the process $\mathtt{Fw}(ab) = !a(x).\overline{b}(y).y.\overline{x}$. This agent links two lo-

$$[\![\overline{a} \bigoplus_{i\in I} \mathtt{in}_i(\tilde{y}_i).P_i \rhd \Gamma, a : \bigoplus_{i\in I}(\tilde{\tau}_i)]\!]^{\Delta, a:\bigoplus_{i\in I} \tilde{z}_i:\tilde{\sigma}_i}$$
$$= \sum_{i\in I} \overline{a}\mathtt{in}_i(\tilde{z}_i).[\![P_i[\tilde{z}_i/\tilde{y}_i] \rhd \Gamma, \tilde{z}_i : \tilde{\tau}_i]\!]^{\Delta, \tilde{z}_i:\tilde{\sigma}_i}$$

$$[\![a \&_{i\in I} \mathtt{in}_i(\tilde{y}_i).P_i \rhd \Gamma, a : \&_{i\in I} \tilde{\tau}_i]\!]^{\Delta, a:\&_{i\in I} \tilde{z}_i:\tilde{\sigma}_i}$$
$$= \sum_{i\in I} a\mathtt{in}_i(\tilde{z}_i).[\![P_i[\tilde{z}_i/\tilde{y}_i] \rhd \Gamma, \tilde{z}_i : \tilde{\tau}_i]\!]^{\Delta, \tilde{z}_i:\tilde{\sigma}_i}$$

$$[\![!a(\tilde{y}).P \rhd \Gamma, a : (\tilde{\tau})^{!}]\!]^{\Delta[K], a:\bigotimes_{k\in K}(\tilde{y}^k:\tilde{\sigma}^k)} = \prod_{k\in K} a(\tilde{y}^k).[\![P[\tilde{y}^k/\tilde{y}] \rhd \Gamma[\tilde{y}^k/\tilde{y}]]\!]^{\Delta^k, \tilde{y}^k:\tilde{\sigma}^k}$$

$$[\![\overline{a}(\tilde{y}).P \rhd \Gamma, a : (\tilde{\tau})^{?}]\!]^{\Delta, a:\biguplus_{h\in H\uplus\{*\}}(\tilde{w}_h:\tilde{\sigma}_h)} = \overline{a}(\tilde{w}_*).[\![P[\tilde{w}_*/\tilde{y}] \rhd \Gamma, \tilde{w}_* : \tilde{\tau}_*]\!]^{\Delta, \tilde{w}_h:\tilde{\sigma}_h}$$

$$[\![P_1 \mid P_2 \rhd \Gamma_1 \odot \Gamma_2]\!]^{\Delta_1\odot\Delta_2} = ([\![P_1 \rhd \Gamma_1]\!]^{\Delta_1} \| [\![P_2 \rhd \Gamma_2]\!]^{\Delta_2}) \setminus Dis(\Delta_1 \odot \Delta_2)$$

$$[\![\mathbf{0} \rhd x_i : (\tau_i)^{?}, y_j : \updownarrow]\!]^{x_i:\biguplus_{h\in H}\Gamma_h, y_j:\updownarrow} = \emptyset$$

$$[\![(\boldsymbol{\nu}\, a)P \rhd \Gamma]\!]^{\Delta} = [\![P \rhd \Gamma, a : \tau]\!]^{\Delta, a:\sigma} \setminus \{a\}$$

Fig. 7. Event Structure Semantics of the πI-Calculus

cations a and b and it is called a *forwarder*. It can be derived that $\mathtt{Fw}(ab) \rhd a : \tau, b : \overline{\tau}$ with $\tau = (()^{\uparrow})^{!}$. Consider the process $P_\omega = \mathtt{Fw}(ab) \mid \mathtt{Fw}(ba)$ so that $P_\omega \rhd a, b : \tau$. The interpretation $[\![\mathtt{Fw}(ab) \rhd a : \tau, b : \overline{\tau}]\!]^{\Delta_1}$ is defined for

$$\Delta_1 = a : \bigotimes_{k\in K}(x^k : ()^{\uparrow}), b : \biguplus_{k\in K}(y^k : ()^{\downarrow}).$$

Similarly the semantics $[\![\mathtt{Fw}(ba) \rhd b : \tau, a : \overline{\tau}]\!]^{\Delta_2}$ is defined for

$$\Delta_2 = b : \bigotimes_{h\in H}(z^h : ()^{\uparrow}), a : \biguplus_{h\in H}(w^h : ()^{\downarrow}) \ .$$

Assuming there are two "synchronising" injective functions $f : K \to H, g : H \to K$, such that $y^k = z^{f(k)}, w^h = x^{g(h)}$ (if not, we can independently perform a fresh injective renaming on both environments), we obtain that the corresponding types for a, b match, so that $\Delta_1 \odot \Delta_2$ is defined. Therefore the semantics of $[\![P_\omega \rhd (\Gamma_1 \odot \Gamma_2)]\!]^{\Delta}$ is defined for

$$\Delta = a : \bigotimes_{k\in K\setminus g(H)}(x^k : ()^{\downarrow}), b : \bigotimes_{h\in H\setminus f(K)}(z^h : ()^{\downarrow}).$$

5.2 Properties of the semantics

The main property of the typed semantics is that all denoted event structures are confusion free. More specifically, the semantics of a typed process is a typed event structure.

Theorem 5.3 *Let P be a process and Γ an environment such that $P \rhd \Gamma$. Suppose that $[\![P \rhd \Gamma]\!]^{\Delta}$ is defined. Then $[\![P \rhd \Gamma]\!]^{\Delta} \rhd \Delta$.*

The syntax introduces the conflict explicitly, therefore we cannot obtain conflict free event structures. The result above shows that no new conflict is introduced through synchronisation. In the deterministic fragment, synchronisation does indeed resolve the conflicts. Firstly, the semantics of the deterministic πI-calculus is in term of deterministic event structures:

Proposition 5.4 *Suppose P is a deterministic process, and that $[\![P \triangleright \Gamma]\!]^\Delta$ is defined. Then $[\![P \triangleright \Gamma]\!]^\Delta$ is deterministic.*

Secondly, once all choices have been matched with selections, or cancelled out, what remains is a conflict free event structure.

Proposition 5.5 *Let X be the set of names in a deterministic process P. Then $[\![P \triangleright \Gamma]\!]^\Delta \setminus X$ is a conflict free event structure. In particular if $[\![\Gamma]\!] = \emptyset$, then $[\![P \triangleright \Gamma]\!]^\Delta$ is conflict free.*

Finally we have the simple fragment. In this case the syntax does not introduce directly any conflict and the typing guarantees that no conflict is introduced by the parallel composition.

Proposition 5.6 *Suppose P is a simple process such that $[\![P \triangleright \Gamma]\!]^\Delta$ is defined. Then $[\![P \triangleright \Gamma]\!]^\Delta$ is simple (and thus conflict free).*

There is a correspondence between the event structure semantics and the operational semantics. The basic result is that the semantics is sound with respect to bisimulation.

Proposition 5.7 (Soundness) *Suppose that for some Δ, $[\![P \triangleright \Gamma]\!]^\Delta = [\![P' \triangleright \Gamma]\!]^\Delta$. Then $P \triangleright \Gamma \approx P' \triangleright \Gamma$.*

Note that the event structure semantics of CCS is already not fully abstract with respect to bisimulation [31], hence the other direction does not hold in our case either. However, as in the event structure semantics of CCS, there is another kind of correspondence between the labelled transition systems and the event structures.

Theorem 5.8 *Let $\cong$ denote isomorphism of labelled event structures.*

Suppose $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ in the π-calculus. Then there exists Δ such that $[\![P \triangleright \Gamma]\!]^\Delta$ is defined and $[\![P \triangleright \Gamma]\!]^\Delta \xrightarrow{\beta} \cong [\![P' \triangleright \Gamma \setminus \beta]\!]^{\Delta \setminus \beta}$.

Conversely, suppose $[\![P \triangleright \Gamma]\!]^\Delta \xrightarrow{\beta} \mathscr{E}'$. Then there exists P' such that $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ and $[\![P' \triangleright \Gamma \setminus \beta]\!]^{\Delta \setminus \beta} \cong \mathscr{E}'$.

6 Conclusions and related work

This paper has provided a typing system for event structures and exploited it to give an event structure semantics of the π-calculus. As far as we know, this work offers the first formalisation of a notion of types in event structures, and the first direct event structure semantics of the π-calculus.

The work is quite technical and it requires a little effort to be read. The readers may ask themselves what they gain from this effort. We think the contribution of this paper are as follows.

- It is a standard intuition that confluence means absence of conflict, determinism. In this work we have *formalised* this intuition. In the process of this formalisation some conflict situations that are *hidden* by the interleaving semantics were discovered. This fact can be underlined by noting that the standard event structure semantics of the so called *confluent* CCS [20] is *not* conflict free.
- It is well known how to compose event structures in order to obtain event structures. However it was not known how to compose confusion free event structures in order to obtain confusion free event structures. Our work offers a solution to this problem. Concrete data structures, a fundamental concept in various fields of semantics, can be seen as confusion free event structures. Therefore our work also shows how to compose concrete data structures.
- Although several causal semantics of the π-calculus exist (see related work below), no one ever gave a *direct* event structure semantics, that could be seen as an extension of Winskel's semantics of CCS. We believe the main difficulty of an event structures semantics of the π-calculus lies in the handling of name generation. Name generation is a inherently *dynamic* operation, while event structures have a more *static*, denotational flavour. We have shown that, by restricting the amount of concurrency to that permitted by the linear type discipline, we can deal with name generation statically, and thus we can extend Winskel's semantics. This restricted π-calculus is still very expressive, in that it can encode fully abstractly functional programming languages.
- Finally, this work is an important preliminary step of several research directions that we believe to be fruitful and interesting, as shown in the next paragraph.

Future work

Future works include extending this approach to a probabilistic framework, for instance the probabilistic π-calculus [14], by using a typed version of probabilistic event structures [27]. The typed λ-calculus can be encoded into the typed π-calculus. This provides an event structure semantics of the λ-calculus, that we want to study in detail. Also the types of the λ-calculus are given an event structure semantics. We aim at comparing this "true concurrent" semantics of the λ-types with concurrent games [19], and with ludics nets [13].

Related work

There are several causal models for the π-calculus, that use different techniques. In [5,10], the causal relations between transitions are represented by "proofs" of the transitions which identify different occurrences of the same transition. In our case a similar role is played by names in types. In [8], a more abstract approach is followed, which involves indexed transition systems. In [16], a semantics of the π-calculus in terms of pomsets is given, following ideas from dataflow theory. The two papers [7,12] present Petri nets semantics of the π-calculus. Since we can unfold Petri nets into event structures, these could indirectly provide event structure semantics of the π-calculus. In [2], an event structure unfolding of double push-out rewriting systems is studied, and this could also indirectly provide an event structure semantics of the π-calculus, via the double push-out semantics of the π-calculus presented in [22]. In [6], Petri Nets are used to provide a type theory for the Join-calculus, a language with several features in common with the π-calculus. None of the above semantics directly uses event structures and no notion of compositional typing systems in true concurrent models is presented. In addition, none of them is used to study a correspondence between semantics and behavioural properties of the π-calculus in our sense.

In [34], event structures are used in a different way to give semantics to a process language, a kind of value passing CCS. That technique does not apply yet to the π-calculus where we need to model creation of new names, although recent work [33] is moving in that direction.

Acknowledgements

We thank the anonymous referees for their comments. We acknowlegde the support of the EPSRC grant GR/T04724/01 "Program Analysis and the Typed Pi-Calculus" .

References

[1] S. Abbes and A. Benveniste. Branching cells as local states for event structures and nets: Probabilistic applications. In *Proceedings of 8th FoSSaCS*, volume 3441 of *LNCS*, pages 95–109. Springer, 2005.

[2] P. Baldan, A. Corradini, and U. Montanari. Unfolding and event structure semantics for graph grammars. In *Proceedings of 2nd FoSSaCS*, volume 1578 of *LNCS*, pages 73–89. Springer, 1999.

[3] M. Berger, K. Honda, and N. Yoshida. Sequentiality and the π-calculus. In *Procceedingss of TLCA'01*, volume 2044 of *LNCS*, pages 29–45, 2001.

[4] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20(265–321), 1982.

[5] M. Boreale and D. Sangiorgi. A fully abstract semantics for causality in the π-calculus. *Acta Inf.*, 35(5):353–400, 1998.

[6] M. G. Buscemi and V. Sassone. High-level petri nets as type theories in the join calculus. In *Proceedings of 4th FOSSACS*, volume 2030 of *LNCS*, pages 104–120. Springer, 2001.

[7] N. Busi and R. Gorrieri. A petri net semantics for pi-calculus. In *Proceedings of 6th CONCUR*, volume 962 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 1995.

[8] G. L. Cattani and P. Sewell. Models for name-passing processes: Interleaving and causal. In *Proceedings of 15th LICS*, pages 322–332, 2000.

[9] P. Degano, R. De Nicola, and U. Montanari. On the consistency of "truly concurrent" operational and denotational semantics (extended abstract). In *Proceedings of 3rd LICS*, pages 133–141, 1988.

[10] P. Degano and C. Priami. Non-interleaving semantics for mobile processes. *Theoretical Computer Science*, 216(1-2):237–270, 1999.

[11] J. Desel and J. Esparza. *Free Choice Petri Nets.* Cambridge University Press, 1995.

[12] J. Engelfriet. A multiset semantics for the pi-calculus with replication. *Theoretical Computer Science*, 153(1&2):65–94, 1996.

[13] C. Faggian and F. Maurel. Ludics nets, a game model of concurrent interaction. In *Proceedings of 20th LICS*, pages 376–385, 2005.

[14] M. Herescu and C. Palamidessi. Probabilistic asynchronous π-calculus. In *Proceedings of 3rd FoSSaCS*, volume 1784 of *LNCS*, pages 146–160. Springer, 2000.

[15] K. Honda and N. Yoshida. On reduction-based process semantics. *TCS*, 151(2):385–435, 1995.

[16] L. J. Jagadeesan and R. Jagadeesan. Causality and true concurrency: A data-flow analysis of the pi-calculus (extended abstract). In *Proceedings of 4th AMAST*, volume 936 of *LNCS*, pages 277–291. Springer, 1995.

[17] G. Kahn and G. D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1-2):187–277, 1993.

[18] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 279–324. Springer, 1986.

[19] P.-A. Melliès. Asynchronous games 4: A fully complete model of propositional linear logic. In *Proceedings of 20th LICS*, pages 386–395, 2005.

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

[21] R. Milner. *Communicating and Mobile Systems: The Pi Calculus.* Cambridge University Press, 1999.

[22] U. Montanari and M. Pistore. Concurrent semantics for the π-calculus. *Electr. Notes Theor. Comput. Sci.*, 1, 1995.

[23] M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

[24] G. Rozenberg and J. Engelfriet. Elementary net systems. In *Dagstuhl Lecturs on Petri Nets*, volume 1491 of *LNCS*, pages 12–121. Springer, 1996.

[25] G. Rozenberg and P. Thiagarajan. Petri nets: Basic notions, structure, behaviour. In *Current Trends in Concurrency*, volume 224 of *LNCS*, pages 585–668. Springer, 1986.

[26] D. Sangiorgi. Internal mobility and agent passing calculi. In *Proc. ICALP'95*, 1995.

[27] D. Varacca, H. Völzer, and G. Winskel. Probabilistic event structures and domains. In *Proceedings of 15th CONCUR*, volume 3170 of *LNCS*, pages 481–496. Springer, 2004.

[28] D. Varacca and N. Yoshida. Event structures, types and the π-calculus. Technical Report 2005/06, Imperial College London, 2005. Available at www.doc.ic.ac.uk/~varacca.

[29] V. Vasconcelos. Typed concurrent objects. In *Proc. ECOOP'94*, 1994.

[30] G. Winskel. *Events in Computation.* Ph.D. thesis, Dept. of Computer Science, University of Edinburgh, 1980.

[31] G. Winskel. Event structure semantics for CCS and related languages. In *Proceedings of 9th ICALP*, volume 140 of *LNCS*, pages 561–576. Springer, 1982.

[32] G. Winskel. Event structures. In *Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, volume 255 of *LNCS*, pages 325–392. Springer, 1987.

[33] G. Winskel. Name generation and linearity. In *Proceedings of 20th LICS*, pages 301–310. IEEE Computer Society, 2005.

[34] G. Winskel. Relations in concurrency. In *Proceedings of 20th LICS*, pages 2–11. IEEE Computer Society, 2005.

[35] G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of logic in Computer Science*, volume 4. Clarendon Press, 1995.

[36] N. Yoshida. Type-based liveness guarantee in the presence of nontermination and nondeterminism. Technical Report 2002-20, MCS Technical Report, University of Leicester, 2002.

[37] N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the π-Calculus. In *Proceedings of LICS'01*, pages 311–322. IEEE, 2001. The full version in *Journal of Inf. & Comp.*., 191 (2004) 145–202, Elsevier.

A Appendix: π-calculus types

The typing system for the linear nondeterministic π-calculus is defined in Figure A.1. The (Zero) rule types $\mathbf{0}$. As $\mathbf{0}$ has no free names, it is not being given any channel types. In (Par), $\Gamma_1 \odot \Gamma_2$ guarantees the consistent channel usage like linear inputs being only composed with linear outputs, etc. In (Res), we do not allow $\uparrow$, $\mathbf{?}$ or $\downarrow$-channels to be restricted since they carry actions which expect their dual actions to exist in the environment. (WeakOut) and (WeakCl) weaken with $\mathbf{?}$-names or $\updownarrow$-names, respectively, since these modes do not require *further* interaction. (LIn) ensures that x occurs precisely once. (LOut) is dual. (RIn) is the same as (LIn) except that no free linear channels are suppressed. This is because a linear channel under replication could be used more than once. (ROut) is similar with (LOut). Note we need to apply (WeakOut) before the first application of (ROut).

We recall that for a label β, the predicate Γ allows β is defined as follows:

- for all Γ, Γ allows $\boldsymbol{\tau}$;
- if $MD(\Gamma(x)) = \downarrow$, then Γ allows $x\mathtt{in}_i(\tilde{y})$;
- if $MD(\Gamma(x)) = \uparrow$, then Γ allows $\overline{x}\mathtt{in}_i(\tilde{y})$;
- if $MD(\Gamma(x)) = \mathbf{!}$, then Γ allows $x(\tilde{y})$;
- if $MD(\Gamma(x)) = \mathbf{?}$, then Γ allows $\overline{x}(\tilde{y})$.

Whenever Γ allows β, we define $\Gamma \setminus \beta$ as follows:

- for all Γ, $\Gamma \setminus \boldsymbol{\tau} = \Gamma$;
- if $\Gamma = \Delta, x : \&_{i\in I} (\tilde{\tau}_i)^{\downarrow}$, then $\Gamma \setminus x\mathtt{in}_i(\tilde{y}) = \Delta, \tilde{y} : \tilde{\tau}$;
- if $\Gamma = \Delta, x : \bigoplus_{i\in I} (\tilde{\tau}_i)^{\uparrow}$, then $\Gamma \setminus \overline{x}\mathtt{in}_i(\tilde{y}) = \Delta, \tilde{y} : \tilde{\tau}$;
- if $\Gamma = \Delta, x : (\tilde{\tau})^{\mathbf{!}}$, then $\Gamma \setminus x(\tilde{y}) = \Gamma, \tilde{y} : \tilde{\tau}$;
- if $\Gamma = \Delta, x : (\tilde{\tau})^{\mathbf{?}}$, then $\Gamma \setminus \overline{x}(\tilde{y}) = \Gamma, \tilde{y} : \tilde{\tau}$.

B Appendix: Event structure types

In this section we refer to the types and the type environments defined in Section 4.

It is straightforward to define duality between types by exchanging branching and offer, with selection and request, respectively. Therefore, for every type τ and environment Γ, we can define their dual $\overline{\tau}$, $\overline{\Gamma}$. However types and environments enjoy a more general notion of duality that is expressed by the following definition. We define a notion of matching for types. The matching of two types produces a "residual" type.

We define the symmetric relations $match[\tau, \sigma]$, $match[\Gamma, \Delta]$ and the partial function $res[\tau, \sigma]$ as follows:

- let $\Gamma = x_1 : \sigma_1 \dots x_n : \sigma_n$ and $\Delta = y_1 : \tau_1 \dots y_m : \tau_m$. Then $match[\Gamma, \Delta]$ if $n = m$ and for every $i \le n$ we have that $x_i = y_i$ and $match[\sigma_i, \tau_i]$;

$$\frac{P \triangleright \Gamma, a : \tau \quad a \notin \Gamma \quad MD(\tau) = \mathbf{!}, \updownarrow}{(\boldsymbol{\nu}\, a)P \triangleright \Gamma}\ \text{Res} \qquad \frac{}{\mathbf{0} \triangleright \emptyset}\ \text{Zero} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : \updownarrow}\ \text{WeakCl}$$

$$\frac{P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i \quad a \notin \Gamma \quad I \subseteq J}{\overline{a} \bigoplus_{i \in I} \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, a : \bigoplus_{i \in J} (\tilde{\tau}_i)^{\uparrow}}\ \text{LOut} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : (\tilde{\tau})^{\mathbf{?}}}\ \text{WeakOut}$$

$$\frac{P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i \quad a \notin \Gamma}{a \,\&_{i \in I} \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, a : \&_{i \in I} (\tilde{\tau}_i)^{\downarrow}}\ \text{LIn} \qquad \frac{P_i \triangleright \Gamma_i \quad (i = 1, 2)}{P_1 \mid P_2 \triangleright \Gamma_1 \odot \Gamma_2}\ \text{Par}$$

$$\frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau} \quad a \notin \Gamma \quad \forall (x{:}\tau) \in \Gamma.\ MD(\tau) = ?}{!a(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{\mathbf{!}}}\ \text{RIn} \qquad \frac{P \triangleright \Gamma, a : (\tilde{\tau})^{\mathbf{?}}, \tilde{y} : \tilde{\tau}}{\overline{a}(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{\mathbf{?}}}\ \text{ROut}$$

Fig. A.1. Linear Typing Rules

- let $\tau = \&_{i \in I} \Gamma_i$ and $\sigma = \bigoplus_{j \in J} \Delta_j$. Then $match[\tau, \sigma]$ if $I = J$ and for all $i \in I$, $match[\Gamma_i, \Delta_i]$; in such a case $res[\tau, \sigma] = \updownarrow$;
- let $\tau = \bigotimes_{i \in I} \Gamma_i$ and $\sigma = \biguplus_{j \in J} \Gamma_j$. Then $match[\tau, \sigma]$ if $J \subseteq I$ and for all $j \in J$, $match[\Gamma_j, \Delta_j]$; in such a case $res[\tau, \sigma] = \bigotimes_{i \in I \setminus J} \Gamma_i$;
- $match[\updownarrow, \updownarrow]$, $res[\updownarrow, \updownarrow] = \updownarrow$.

A branching type matches a corresponding selection types and the residual type is the special type recording that the matching has taken place. A client type matches a server type if all requests correspond to an available resource. The residual type records which resources are still available.

We now define the composition of two environments. Two environments can be composed if the types of the common names match. Such names are given the residual type by the resulting environment. Client types can by joined, so that the two environments are allowed to independently reserve some resources. Given two type environments Γ_1, Γ_2 we define the environment $\Gamma_1 \odot \Gamma_2 \stackrel{\text{def}}{=} \Gamma$ as follows:

- if $x \notin Dom(\Gamma_1)$ and no name in $\Gamma_2(x)$ appears in Γ_1, then $\Gamma(x) = \Gamma_2(x)$, and symmetrically;
- if $\Gamma_1(x) = \tau, \Gamma_2(x) = \sigma$ and $match[\tau, \sigma]$, then $\Gamma(x) = res[\tau, \sigma]$;
- if $\Gamma_1(x) = \biguplus_{i \in I} \Delta_i$ and $\Gamma_2(x) = \biguplus_{j \in J} \Delta_j$ and no name appears in both Δ_i and Δ_j for every $i, j \in I \cup J$ then $\Gamma(x) = \biguplus_{i \in I \cup J} \Delta_i$;
- if any of the other cases arises, then Γ is not defined.

We now define what it means for a label α to be *allowed* by a type environment Γ. Suppose $\Gamma(x) = \sigma$, then:

- if $\alpha = x\mathtt{in}_j(\tilde{y})$, and if $\sigma = \&_{i \in I} \Gamma_i$ where $\tilde{y}$ is the domain of Γ_j, then α is allowed;

- if $\alpha = \overline{x}\mathtt{in}_j(\tilde{y})$, and if $\sigma = \bigoplus_{i \in I} \Gamma_i$ where $\tilde{y}$ is the domain of Γ_j then α is allowed;
- if $\alpha = x(\tilde{y})$, and if $\sigma = \bigotimes_{i \in I} \Gamma_i$ where $\tilde{y}$ is the domain of Γ_j then α is allowed;
- if $\alpha = \overline{x}(\tilde{y})$, and if $\sigma = \biguplus_{i \in I} \Gamma_i$ where $\tilde{y}$ is the domain of Γ_j then α is allowed;
- if $\alpha = \tau$, then α is allowed.

And finally, α is also allowed by Γ if α is allowed by any of the environments appearing in the types in the range of Γ. Note that if a label is allowed, the definition of well-formedness guarantees that it is allowed in a unique way. Note also that if a label α has subject x and x does not appear in Γ, then α is not allowed by Γ.