Theoretical Computer Science 411 (2010) 1949–1973

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Typed event structures and the linear π-calculus

Daniele Varacca [a,*], Nobuko Yoshida [b]

[a] PPS, Université Paris Diderot & CNRS, France
[b] Imperial College London, UK

ARTICLE INFO

Keywords:
Event structures
Types
Linearity
Confusion freeness
π-calculus

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.

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* [45]. Interleaving models are very successful in defining observational equivalences, by means of bisimulation [28]. In causal models, causality and concurrency are explicitly represented. Instances of such models are *Petri nets* [33], *Mazurkiewicz traces* [26] and *event structures* [31]. True concurrent models can easily represent interesting behavioural properties such as absence of conflict, independence of the choices and sequentiality [33].

In this paper we address a particular true concurrent model: the model of *event structures* [31,42]. Event structures have been used to give semantics to concurrent process languages. The earliest and possibly the most intuitive is Winskel's semantics of Milner's CCS [41].

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.

* Corresponding author.
E-mail address: varacca@pps.jussieu.fr (D. Varacca).

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2010.01.024

Confusion freeness was first identified in the context the theory of Petri nets [33]. It has been studied in that context, in the form of free choice nets [15]. Confusion free event structures are also known as *concrete data structures* [4], and their domain-theoretic counterpart are the *concrete domains* [25]. Finally, confusion freeness has been recognised as an important property in the context of probabilistic models [35,1].

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 [29]. Various causal semantics of the π-calculus existed before [24,9,16,5,14,10], but none was 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 (more precisely, the extension of the calculus in [3] to the nondeterministic one). This fragment is expressive enough to encode the typed λ-calculus (in fact, to encode it *fully abstractly* [3,47]). 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.

To substantiate this intuition, we provide a fully abstract encoding of the linearly typed fragment of the π-calculus into an intermediate process language, which is syntactically similar to the π-calculus except that α-conversion is not allowed. We devise a typing system for this language that makes use of the event structure types. We then provide the language with a semantics in terms of typed event structures. Via this fully abstract intermediate translation, we thus obtain a sound event structure semantics of the π-calculus, which follows the same lines as Winskel's: 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 [28]. 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 This paper is the full version of [37], with complete definitions and detailed proofs. The present paper provides the full definition of the intermediate language which was omitted from [37] and gives more examples and explanations on typing systems and event structures. Comparisons with related work are also updated.

Section 2 presents a linearly typed version of the πI-calculus. This section is inspired from [47], 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. We briefly introduce the category of event structures and we explicitly describe the categorical product. The product of event structures is one of the basic ingredients in the definition of the parallel composition. The explicit definition we present allows us to carry out the proofs in the following sections. 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. In Section 5, we present the intermediate process language which is used to bridge between the typed event structures and the linear π-calculus. We call this calculus *Name Sharing CCS* or NCCS. We define a notion of typing for NCCS processes and its typed operational semantics. In Section 6, we give a semantics of typed NCCS processes in terms of event structures. The main result of this section is that the semantics of a typed process is a typed event structure. We also show that this semantics is sound with respect to bisimulation. In Section 7, we provide a fully abstract translation of the the typed πI-calculus, into NCCS. Through the sound event structure semantics of NCCS, we obtain a sound semantics of the π-calculus in terms of event structures. 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 8 concludes with related and future works. The Appendix contains the proofs of the results presented in the paper.

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 [46]. The reader may refer to [3,46] for a more detailed description and more examples.

2.1. Syntax and reduction

As anticipated, we consider a restricted version of the π-calculus [29], where only bound names are passed in interaction. The resulting calculus is called the πI-calculus in the literature [34]. Syntactically we restrict all outputs to be of 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 nondeterministic.

Nondeterministic input is called *branching*, and it is already present in [3], while nondeterministic 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 [39]. The formal grammar of the calculus is defined below.

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

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. The names in $\tilde{y_i}$ are bound in the continuation P_i. The process $!x(\tilde{y}).P$ is a replicated input, binding $\tilde{y}$. $P \mid Q$ is a parallel composition and $(\boldsymbol{\nu}\, x)P$ is a restriction that binds x. We omit the empty tuple: for example, $\overline{x}$ stands for $\overline{x}()$. When the index in the branching indexing set is a singleton we use the notation $x(\tilde{y}).P$ when it is binary, we write $x((\tilde{y_1}).P_1\&(\tilde{y_2}).P_2)$ (and similarly for selection). Notions of bound/free names, $\alpha-$ and structural equivalences, and of evaluation contexts are defined as usual [29,3,47,23].

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:

$$x\&_{i\in I}\mathtt{in}_i(\tilde{y}_i).P_i \mid \overline{x}\bigoplus_{j\in J}\mathtt{in}_j(\tilde{y}_j).Q_j \;\longrightarrow\; (\boldsymbol{\nu}\,\tilde{y}_h)(P_h \mid Q_h) \quad (h \in I \cap J)$$

$$!x(\tilde{y}).P \mid \overline{x}(\tilde{y}).Q \;\longrightarrow\; !x(\tilde{y}).P \mid (\boldsymbol{\nu}\,\tilde{y})(P \mid Q)$$

closed under evaluation contexts and structural equivalence. A nondeterministic branching synchronises with a selection on one of the common branches, the communicated names are restricted, and the continuations triggered. An output can also synchronise with a replicated server which is still present after the reduction. Note that α-conversion may be necessary for two processes to synchronise. For instance, consider the process $x(y).P \mid \overline{x}(z).Q$. Assuming y is fresh for Q, it can be α-converted to $x(y).P \mid \overline{x}(y).Q[y/z]$, allowing synchronisation.

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 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:

$$Q_1 \stackrel{\text{def}}{=} \overline{x}.y \mid \overline{x}.z \mid x \qquad Q_2 \stackrel{\text{def}}{=} y.\overline{x} \mid z.\overline{y} \mid x.(\overline{z} \mid \overline{w})$$

Then Q_1 is not typable as x 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:

$$Q_3 \stackrel{\text{def}}{=} \overline{x}.(y \oplus z) \mid x.(\overline{w} \,\&\, \overline{v})$$

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

$$Q_4 \stackrel{\text{def}}{=} !\,y.\overline{x} \mid !\,y.\overline{z} \qquad Q_5 \stackrel{\text{def}}{=} !\,y.\overline{x} \mid \overline{y} \mid !\,z.\overline{y}$$

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

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

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

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

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

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

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

Fig. 1. Linear typing rules.

$$\overline{x} \bigoplus_{i\in I}(\tilde{y}_i).P_i \xrightarrow{\overline{x}\mathtt{in}_j(\tilde{y}_j)} P_j \qquad x \,\&_{i\in I}(\tilde{y}_i).P_i \xrightarrow{x\mathtt{in}_j(\tilde{y}_j)} P_j \qquad !x(\tilde{y}).P \xrightarrow{x(\tilde{y})} P \mid !x(\tilde{y}).P \qquad \overline{x}(\tilde{y}).P \xrightarrow{\overline{x}(\tilde{y})} P$$

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

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

Fig. 2. Labelled transition system for the πI-calculus.

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:

(1) $\tau \odot \overline{\tau} = \updownarrow$ with $MD(\tau) = \downarrow$
(2) $\tau \odot \overline{\tau} = \overline{\tau}, \quad \tau \odot \tau = \tau$ with $MD(\tau) = ?$

Then, the 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 defined in Fig. 1. They are identical to the affine π-calculus [3] except a straightforward modification to deal with the non-deterministic output. We refer to [3] for an informal discussion on the meaning of the rules. We just note here that, in the rule (Par) the use of $\Gamma_1 \odot \Gamma_2$ guarantees the consistent channel usage. For instance it guarantees that linear inputs are only composed with linear outputs.

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:

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

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. It is convenient, for the proofs, to use synchronisation labels that keep track of which synchronisation took place. However, as it is customary, we consider synchronisation transitions not to be observable. Thus for the purpose of defining observational equivalences, all $\boldsymbol{\tau}$-labels will be identified.

The standard untyped transition relation is defined in Fig. 2. 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 $x\mathtt{in}_i(\tilde{y})$;
- if $MD(\Gamma(x)) = \uparrow$, then Γ allows $\bar{x}\mathtt{in}_i(\tilde{y})$;
- if $MD(\Gamma(x)) = \mathbf{!}$, then Γ allows $x(\tilde{y})$;
- if $MD(\Gamma(x)) = \mathbf{?}$, then Γ allows $\bar{x}(\tilde{y})$.

Whenever Γ allows β, we define a new environment $\Gamma \setminus \beta$ 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 $\Gamma \setminus \bar{x}\mathtt{in}_i(\tilde{y}) = \Delta, \tilde{y} : \tilde{\tau}$;
- 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 $\Gamma \setminus \bar{x}(\tilde{y}) = \Gamma, \tilde{y} : \tilde{\tau}$.

The environment $\Gamma \setminus \beta$ represents what remains of Γ 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.

The typed transition, written $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma'$ is defined by

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

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 $x : (\tilde{\tau})^{\mathbf{!}}$ 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 $\bar{x}.y \,|\, \bar{y}.x$ which is typed in the environment $x : \updownarrow, y : \updownarrow$. Although the process has some untyped transitions, none of them is allowed by the environment.

By induction on the rules in Fig. 2, we can obtain:

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

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

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

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

As anticipated, in the above definition we allow a $\boldsymbol{\tau}$ label to be matched by a different $\boldsymbol{\tau}$ label. The identities of different $\boldsymbol{\tau}$ labels are considered only in some of the proofs.

If there exists a bisimulation between two judgments, we say that they are bisimilar $(P \triangleright \Gamma) \approx (P' \triangleright \Gamma)$. It can be proved that $\approx$ is a congruent relation. The proof is analogous to the one in Appendix C.3 of [47].

3. Event structures

Event structures were introduced by Nielsen, Plotkin and Winskel [31,40], 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* [42]. For the relations of event structures with other models for concurrency, the standard reference is [45].

3.1. Basic definitions

An *event structure* is a triple $\mathcal{E} = \langle E, \leq, \smile \rangle$ 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*. The set of maximal elements of $[e)$ is denoted by *parents*(e). A *configuration* x of an event structure $\mathcal{E}$ 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. The set of configurations of $\mathcal{E}$, partially ordered by inclusion, is denoted as $\mathcal{L}(\mathcal{E})$. It is a coherent ω-algebraic domain [31], whose compact elements are the finite configurations.

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

Roughly speaking, $\mathcal{E}\lfloor e$ is $\mathcal{E}$ 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

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

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

3.2. Conflict free and confusion free event structures

An interesting subclass of event structures is the following.

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 $\mathcal{E}$ is conflict free, then $TS(\mathcal{E})$ 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 [35].

3.3. A category of event structures

Event structures form the class of objects of a category [45]. The morphisms are defined as follows. Let $\mathcal{E}_1 = \langle E_1, \leq_1, \smile_1 \rangle$, $\mathcal{E}_2 = \langle E_2, \leq_2, \smile_2 \rangle$ be two event structures. A *morphism* $f : \mathcal{E}_1 \to \mathcal{E}_2$ is a partial function $f : E_1 \to E_2$ such that

- f preserves configurations: if x is a configuration of $\mathcal{E}_1$, then $f(x)$ is a configuration of $\mathcal{E}_2$;
- f is locally injective: let x be a configuration of $\mathcal{E}_1$, if $e, e' \in x$ and $f(e), f(e')$ are both defined with $f(e) = f(e')$, then $e = e'$.

It is straightforward to verify that the identity is a morphism and that morphisms compose, so that what we obtain is indeed a category.

Morphisms reflect conflict and causality and preserve concurrency. They can be equivalently characterised as follows.

Proposition 3.5 (*[45]*). *A partial function $f : E_1 \to E_2$ is a morphism of event structures $f : \mathcal{E}_1 \to \mathcal{E}_2$ if and only if the following are satisfied:*

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

There are various ways of dealing with labels. For the general treatment we refer to [45]. Here we present the simplest notion: take two labelled event structures $\mathcal{E}_1 = \langle E_1, \leq_1, \smile_1, \lambda_1 \rangle$, $\mathcal{E}_2 = \langle E_2, \leq_2, \smile_2, \lambda_2 \rangle$ on the same set of labels L. A morphism $f : \mathcal{E}_1 \to \mathcal{E}_2$ 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.

- Prefixing $\alpha.\mathcal{E}$, where $\mathcal{E} = \langle E, \leq, \smile, \lambda\rangle$. It is the event structure $\langle E', \leq', \smile', \lambda'\rangle$, where $E' = E \uplus \{e'\}$ for some new event e', $\leq'$ coincides with $\leq$ on E and moreover, for every $e \in E$ we have $e' \leq e$, the conflict relation $\smile'$ coincides with $\smile$, that is e' is in conflict with no event. Finally λ' coincides with λ on E and $\lambda'(e') = \alpha$. Intuitively, we add a new initial event, labelled by α.
- Prefixed sum $\sum_{i\in I} \alpha_i.\mathcal{E}_i$. This is obtained by disjoint union of copies of the event structures $\alpha_i.\mathcal{E}_i$, 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. This is a generalisation of prefixing, where we add an initial *cell*, instead of an initial event.
- Restriction $\mathcal{E} \setminus X$ where $\mathcal{E} = \langle E, \leq, \smile, \lambda\rangle$ and $X \subseteq L$ 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 $\mathcal{E}[f]$. This is just composing the labelling function λ with a function $f : L \to L$. The new event structure has thus labelling function $f \circ \lambda$.

It is easy to verify that all these constructions preserve the class of confusion free event structures. Also, with the obvious 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 [45] as the categorical product followed by restriction and relabelling. The existence of the product is deduced via general categorical arguments, but not explicitly constructed.

In order to carry out our proofs, we needed a more concrete representation of the product. We have devised such a representation, which is inspired by the one given in [13], but which is more suitable to an inductive reasoning.

Let $\mathcal{E}_1 := \langle E_1, \leq_1, \smile_1\rangle$ and $\mathcal{E}_2 := \langle E_2, \leq_2, \smile_2\rangle$ be two event structures. Let $E_i^* := E_i \uplus \{*\}$. Consider the set $\tilde{E}$ obtained as the initial solution of the equation $X = \mathcal{P}_{fin}(X) \times E_1^* \times E_2^*$. Its elements have the form (x, e_1, e_2) for x finite, $x \subseteq \tilde{E}$. Initiality allows us to define inductively a notion of *height* of an element of $\tilde{E}$ as

$$h(\emptyset, e_1, e_2) = 0 \quad \text{and} \quad h(x, e_1, e_2) = \max\{h(e) \mid e \in x\} + 1$$

Most of our reasoning will be by induction on the height of the elements. We now carve out of $\tilde{E}$ a set E which will be the support of our product event structure $\mathcal{E}$. At the same time we define the order relation and the conflict relation on $\mathcal{E}$.

Base: we have that $(\emptyset, e_1, e_2) \in E$ if

- $e_1 \in E_1$, $e_2 \in E_2$, and e_1 minimal in E_1, e_2 minimal in E_2 or
- $e_1 \in E_1$, $e_2 = *$ and e_1 minimal in E_1 or
- $e_1 = *$, $e_2 \in E_2$ and e_2 minimal in E_2.

The order on the elements of height 0 is trivial.

Finally we have $(\emptyset, e_1, e_2) \asymp (\emptyset, d_1, d_2)$ if $e_1 \asymp d_1$ or $e_2 \asymp d_2$.

Inductive case: Assume that all elements in E of height $\leq n$ have been defined. Assume that an order relation and a conflict relation has been defined on them. Let (x, e_1, e_2) be of height $n + 1$. Let y be the set of maximal elements of x. Let $y_1 = \{d_1 \in E_1 \mid (z, d_1, d_2) \in y\}$ and $y_2 = \{d_2 \in E_2 \mid (z, d_1, d_2) \in y\}$, be the projections of y onto the two components. We have that $(x, e_1, e_2) \in E$ if x is downward closed and conflict free, and furthermore:

- Suppose $e_1 \in E_1$, $e_2 = *$. Then it must be the case that $y_1 = \mathit{parents}(e_1)$.
- Suppose $e_2 \in E_2$, $e_1 = *$. Then it must be the case that $y_2 = \mathit{parents}(e_2)$.
- Suppose $e_1 \in E_1$, $e_2 \in E_2$. Then
  - if $(z, d_1, d_2) \in y$, then either $d_1 \in \mathit{parents}(e_1)$ or $d_2 \in \mathit{parents}(e_2)$;
  - for all $d_1 \in \mathit{parents}(e_1)$, there exists $(z, d_1, d_2) \in x$;
  - for all $d_2 \in \mathit{parents}(e_2)$ there exists $(z, d_1, d_2) \in x$.
- Let $x_1 = \{d_1 \in E_1 \mid (z, d_1, d_2) \in x\}$ and $x_2 = \{d_2 \in E_2 \mid (z, d_1, d_2) \in x\}$. Then for no $d_1 \in x_1$, $d_1 \asymp e_1$ and for no $d_2 \in x_2$, $d_2 \asymp e_2$.

The partial order is extended by $e \leq (x, e_1, e_2)$ if $e \in x$, or $e = (x, e_1, e_2)$. Note that if $e < e'$ then $h(e) < h(e')$.

Finally for the conflict, take $e = (x, e_1, e_2)$ and $d = (z, d_1, d_2)$, where either $h(e) = n + 1$ or $h(d) = n + 1$ or both. Then we define $e \smile d$ if one of the following holds:

- $e_1 \asymp d_1$ or $e_2 \asymp d_2$, and $e \neq d$;
- there exists $e' = (x', e_1', e_2') \in x$ such that $e_1' \asymp d_1$ or $e_2' \asymp d_2$, and $e' \neq d$;

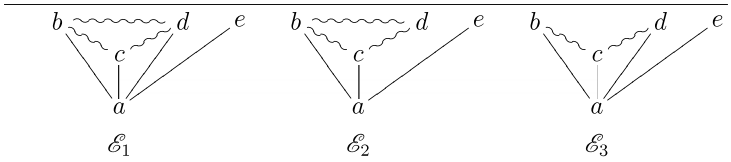

Fig. 3. Event structures.

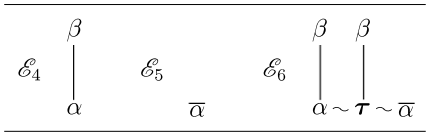

Fig. 4. Parallel composition of event structures.

- there exists $d' = (z', d'_1, d'_2) \in z$ such that $e_1 \asymp d'_1$ or $e_2 \asymp d'_2$, and $e \neq d'$;
- there exists $e \in x$, $d \in z$ such that $e \smile d$.

As the following lemma shows, some of the clauses above are redundant, but are kept for simplicity.

Lemma 3.6 (*Stability*). *If* $(x, e_1, e_2), (x', e_1, e_2) \in E$ *and* $x \neq x'$, *then there exist* $d \in x$, $d' \in x'$ *such that* $d \smile d'$.

Now we are ready to state the main new result of this section: take two event structures $\mathcal{E}_1$, $\mathcal{E}_2$, and let $\mathcal{E} = \langle E, \leq, \smile \rangle$ be defined as above. Then we have:

Theorem 3.7. *The structure* $\mathcal{E}$ *is an event structure and it is the categorical product of* $\mathcal{E}_1$, $\mathcal{E}_2$.

We will not make explicit use of the properties of the categorical product, except that projections preserve configurations. However Theorem 3.7 is necessary to fit in the general framework of models for concurrency, and to avoid building "ad hoc" models.

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_* [45]. Given two labelled event structures $\mathcal{E}_1$, $\mathcal{E}_2$, the parallel composition $\mathcal{E}_1 \|_S \mathcal{E}_2$ is defined as the categorical product followed by restriction and relabelling: $(\mathcal{E}_1 \times \mathcal{E}_2 \setminus X)[f]$ where X is the set of pairs $(\alpha_1, \alpha_2) \in L_* \times L_*$ for which $\alpha_1 \bullet_S \alpha_2$ is undefined, while the function f : is defined as $f(\alpha_1, \alpha_2) = \alpha_1 \bullet_S \alpha_2$. The subscripts S are omitted when the synchronisation algebra is clear from the context.

The simplest possible synchronisation algebra is defined as $\alpha \bullet * = * \bullet \alpha = \alpha$, 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 $(\mathcal{E}_i)_{i \in I}$. In such a case we denote the parallel composition as $\prod_{i \in I} \mathcal{E}_i$.

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 of event structures

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

Example 3.1. Consider the following event structures $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$, defined on the same set of events $E := \{a, b, c, d, e\}$. In $\mathcal{E}_1$, we have $a \leq b, c, d, e$ and $b \smile_\mu c$, $c \smile_\mu d$, $b \smile_\mu d$. In $\mathcal{E}_2$, we do not have $a \leq d$, while in $\mathcal{E}_3$, we do not have $b \smile_\mu d$. The three event structures are represented in Fig. 3, where curly lines represent immediate conflict, while the causal order proceeds upwards along the straight lines.

The event structure $\mathcal{E}_1$ is confusion free, with three cells: $\{a\}$, $\{b, c, d\}$, $\{e\}$. In $\mathcal{E}_2$, there are four cells: $\{a\}$, $\{b, c\}$, $\{d\}$, $\{e\}$. $\mathcal{E}_2$ is not confusion free, because some cells are not closed under immediate conflict. This is an example of *asymmetric* confusion [32]. In $\mathcal{E}_3$ there are four cells: $\{a\}$, $\{b, c\}$, $\{c, d\}$, $\{e\}$. $\mathcal{E}_3$ is not confusion free, because immediate conflict is not transitive. This is an example of *symmetric* confusion.

Example 3.2. Next we show an example of parallel composition, see Fig. 4. Consider the two labelled event structures $\mathcal{E}_4$, $\mathcal{E}_5$, 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 $\alpha \bullet \overline{\alpha} = \tau, \alpha \bullet * = \alpha, \overline{\alpha} \bullet * = \overline{\alpha}, \beta \bullet * = \beta$ and undefined otherwise. Then $\mathcal{E}_6 := \mathcal{E}_4 \| \mathcal{E}_5$ 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 $\lambda(e) = \alpha$, $\lambda(e') = \overline{\alpha}$, $\lambda(e'') = \tau$, $\lambda(d) = \lambda(d'') = \beta$. Note that, while $\mathcal{E}_4$, $\mathcal{E}_5$ are confusion free, $\mathcal{E}_6$ is not, since the reflexive immediate conflict is not transitive (Fig. 5).

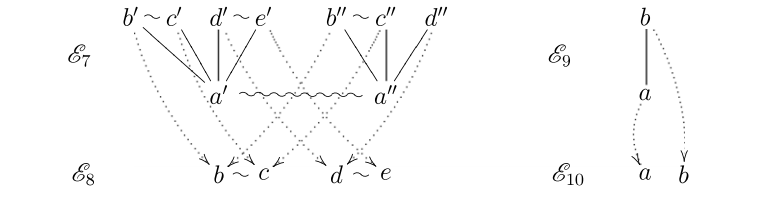

Fig. 5. Morphisms of event structures.

Example 3.3. Finally we show two examples of morphisms. First, consider the two event structures $\mathcal{E}_7$, $\mathcal{E}_8$ 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 $\mathcal{E}_7$ and $\mathcal{E}_8$ are confusion free.

We define a morphism $f : \mathcal{E}_7 \to \mathcal{E}_8$ 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 $\mathcal{E}_9$, $\mathcal{E}_{10}$, where $E_9 = E_{10} = \{a, b\}$, both have empty conflict, and in $\mathcal{E}_9$ we have $a \leq b$. The identity function on $\{a, b\}$ is a morphism $\mathcal{E}_9 \to \mathcal{E}_{10}$ but not vice versa. We can say that the causal order of $\mathcal{E}_9$ refines the causal order of $\mathcal{E}_{10}$.

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 and type environments are generated by the following grammar

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

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.

We say a name is *confidential* for a type environment Γ if it appears inside a type in the range of Γ. A name is *public* if it is in the domain of Γ. Intuitively, confidential names are used to identify different occurrences of events that have the same public label. We will see this explicitly when we introduce the event structure semantics. Technically, in client and server types, we also require the enviroments Γ_i to be nonempty in order to distinguish different components. This is not restrictive, as we can always introduce "dummy" names.

The form of event structures types and environments is similar to those of the π-calculus. In the π-calculus we only keep track of the types of the object names, as their precise identity is irrelevant. In event structure types we recursively keep track not only of the types, but also of the identity of the confidential names. Moreover server and client types explicitly represent each copy of the resource.

Branching types represent the notion of "environmental choice": several choices are available for the environment to choose. Selection types represent the notion of "process choice": some choice is made by the process. In both cases the choice is alternative: one excludes all the others. Server types represent the notion of "available resource": I offer to the environment something that is available regardless of whatever else happens. Client types represent the notion of "concurrent request": I want to reserve a resource that I may use at any time.

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 also produces a set of names that are to be considered as "closed", as they have met their dual. Finally, after two types have matched, they produce a "residual" type.

We define the relations $match[\tau, \sigma] \to S$, $match[\Gamma, \Delta] \to S$ symmetric in the first two arguments, 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 $match[\Gamma, \Delta] \to S$ if $n = m$, for every $i \leq n$ $x_i = y_i$, $match[\sigma_i, \tau_i] \to S_i$ and $S = \bigcup_{i \leq n} S_i \cup \{x_i\}$;
- let $\tau = \&_{i \in I} \Gamma_i$ and $\sigma = \bigoplus_{j \in J} \Delta_j$. Then $match[\tau, \sigma] \to S$ if $I = J$, for all $i \in I$, $match[\Gamma_i, \Delta_i] \to S_i$ and $S = \bigcup_{i \in I} S_i$ 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 $match[\tau, \sigma] \to S$ if $J \subseteq I$, for all $j \in J$, $match[\Gamma_j, \Delta_j] \to S_j$, and $S = \bigcup_{j \in J} S_j$ In such a case $res[\tau, \sigma] = \bigotimes_{i \in I \setminus J} \Gamma_i$.
- $match[\updownarrow, \updownarrow] \to \emptyset$, $res[\updownarrow, \updownarrow] = \updownarrow$.

A branching type matches a corresponding selection types, all their names are closed and the residual type is the special type recording that the matching has taken place. A client type matches a server type if every request corresponds 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. All the closed names are recorded. Client types can be 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 $\Gamma_1 \odot \Gamma_2 \stackrel{\text{def}}{=} \Gamma$ and the set of names $cl(\Gamma_1, \Gamma_2)$ as follows:

- if $x \notin Dom(\Gamma_1)$ and no name in $\Gamma_2(x)$ appears in Γ_1, then $\Gamma(x) = \Gamma_2(x)$, $S_x = \emptyset$ and symmetrically;
- if $\Gamma_1(x) = \tau$, $\Gamma_2(x) = \sigma$ and $match[\tau, \sigma] \to S$, then $\Gamma(x) = res[\tau, \sigma]$ and $S_x = S$;
- 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 $i, j \in I \cup J$ we have then $\Gamma(x) = \biguplus_{i \in I \cup J} \Delta_i$ and $S_x = \emptyset$;
- if any of the other cases arises, then Γ is not defined;
- $cl(\Gamma_1, \Gamma_2) = \bigcup_{x \in Dom(\Gamma_1, \Gamma_2)} S_x$.

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 confidential names. Consider a label α, an environment Γ, and 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 by Γ;
- 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 by Γ;
- 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 by Γ;
- 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 by Γ;
- if $\alpha = \tau$, then α is allowed by Γ;
- if α is allowed by any of the environments appearing in the types in the range of Γ, then α is allowed by 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. Let $Dis(\Gamma)$ be the set of labels that are *not allowed* by the environment Γ.

The semantics of types is presented in Fig. 6, 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 semantics of selection and branching is obtained using the sum of event structures. The semantics of client and server is given using the parallel composition. To define the parallel composition, we use a symmetric synchronisation algebra which extends the one defined in Section 2: $\alpha \bullet * = \alpha$, $x\mathtt{in}_i(\tilde{y}_i) \bullet \overline{x}\mathtt{in}_i(\tilde{y}_i) = (x, \overline{x})\mathtt{in}_i(\tilde{y}_i)$, $x(\tilde{y}) \bullet \overline{x}(\tilde{y}) = (x, \overline{x})(\tilde{y})$, and undefined otherwise. Also the semantics of an environment is obtained as the parallel composition of the semantics of the types, with initial events labelled using the corresponding names. Such parallel compositions do not involve synchronisation due to the condition on uniqueness of names and thus, as we already explained, they can be thought of as disjoint unions.

The following result is a sanity check for our definitions. It shows that matching of types corresponds to parallel composition with synchronisation.

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

4.3. Typing event structures

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

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

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

$$[\![x : \otimes_{i\in I}\, \Gamma_i]\!] = \textstyle\prod_{i\in I} x(\tilde{y_i}).[\![\Gamma_i]\!] \qquad [\![x : \uplus_{i\in I}\, \Gamma_i]\!] = \textstyle\prod_{i\in I} \overline{x}(\tilde{y_i}).[\![\Gamma_i]\!]$$

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

Fig. 6. Denotational semantics of types.

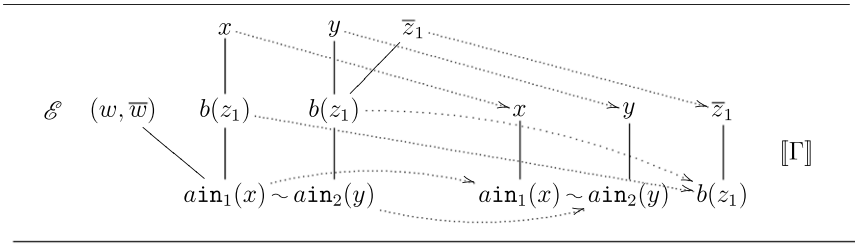

Fig. 7. Typed event structure.

Definition 4.2. We say that $\mathcal{E} \triangleright \Gamma$, if the following conditions are satisfied:

- each cell in $\mathcal{E}$ 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 $f : \mathcal{E} \to [\![\Gamma]\!]$ such that $f(e)$ is undefined if and only if $\lambda(e) \in \boldsymbol{\tau}$.

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

Lemma 4.3. *Suppose $\mathcal{E} \triangleright \Gamma$, and let $e, e' \in E$ be distinct events.*

- *If $\lambda(e) = \lambda(e') \neq \boldsymbol{\tau}$, then $e \smile e'$.*
- *If $\lambda(e), \lambda(e') \neq \boldsymbol{\tau}$ and $\lambda(e)$ and $\lambda(e')$ have the same subject and different branch, then $e \smile e'$.*
- *If $e \smile_\mu e'$, then $\lambda(e)$ and $\lambda(e')$ have the same subject and different branch.*

Theorem 4.4. *Take two labelled confusion free event structures $\mathcal{E}_1, \mathcal{E}_2$. Suppose $\mathcal{E}_1 \triangleright \Gamma_1$ and $\mathcal{E}_2 \triangleright \Gamma_2$. Assume $\Gamma_1 \odot \Gamma_2$ is defined. Then $(\mathcal{E}_1 \| \mathcal{E}_2) \setminus (Dis(\Gamma_1 \odot \Gamma_2))$ is confusion free and $(\mathcal{E}_1 \| \mathcal{E}_2) \setminus (Dis(\Gamma_1 \odot \Gamma_2)) \triangleright \Gamma_1 \odot \Gamma_2$.*

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 4.5. *Take two labelled deterministic (resp. simple) event structures $\mathcal{E}_1 \triangleright \Gamma_1$ and $\mathcal{E}_2 \triangleright \Gamma_2$. Suppose $\Gamma_1 \odot \Gamma_2$ is defined. Then $(\mathcal{E}_1 \| \mathcal{E}_2) \setminus Dis(\Gamma_1 \odot \Gamma_2)$ 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 $(\Gamma)^\uparrow$. When the indexing set of a type is $\{1, 2\}$, we write $(\Gamma_1 \& \Gamma_2)$ or $(\Gamma_1 \otimes \Gamma_2)$.

Example 4.1. Consider the types $\tau_1 = (x : ()^\downarrow \,\&\, y : ()^\downarrow)$, $\sigma_1 = \biguplus_{i\in\{2\}}(z_i :\updownarrow)$ $\tau_2 = (x : ()^\uparrow \oplus y : ()^\uparrow)$, $\sigma_2 = \bigotimes_{i\in\{1,2,3\}}(z_i :\updownarrow)$. We have $match[\tau_1, \tau_2]$, with $res[\tau_1, \tau_2] =\updownarrow$; and $match[\sigma_1, \sigma_2]$, with $res[\sigma_1, \sigma_2] = \bigotimes_{i\in\{2\}}(z_i :\updownarrow)$. 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 4.2. As an example of typed event structures, consider the environment $\Gamma = a : (x : ()^\downarrow \,\&\, y : ()^\downarrow), b : \biguplus_{i\in\{1\}}(z_i : ()^\uparrow)$. Fig. 7 shows an event structure $\mathcal{E}$, such that $\mathcal{E} \triangleright \Gamma$, together with a morphism $\mathcal{E} \to [\![\Gamma]\!]$. Note that the two events in $\mathcal{E}$ labelled with $b(z_1)$ are mapped to the same event and indeed they are in conflict.

5. Name sharing CCS

Our goal is to use typed event structures to interpret the linearly typed π-calculus. We would like this interpretation to be similar, in a sense to extend, Winskel's semantics of CCS [41]. However we face two main difficulties.

The first problem is that Winskel's semantics is strictly related to the labelled transition semantics of CCS. The labelled transition semantics of the π-calculus is more complex, and in particular the communication rule involves α-conversion. This rule seems difficult to represent using the available techniques. The second problem is that we want to use typed event structures, and this implies confusion free event structures. However, even if we applied Winskel's semantics to a fragment of the π-calculus without name passing, we would obtain confused event structures. This is due to the replicated server. If we interpret the replicated sever as an infinite parallel composition of copies of the resource, Winskel's semantics allows each of such components to compete for the same client. This competition creates some spurious conflicts that break confusion freeness. Alternatively, we can model the server as one single resource that, after providing its service, spawns another copy of itself. This would create another spurious conflict between two clients to decide who is going to be served first.

To see this with an example, imagine the server to be a post office. A post office allows a client to post a letter. How do we implement this service? If the post office has only one employee, that accepts one letter at a time, then two clients could end up fighting for the right of going first. If the post office has infinitely many employees, still two clients may fight over the same one (for instance because she is more efficient), or two employees could fight over the same client (because their salary is proportional to their activity).

Our solution to this problem would be to assign *in advance* an employee for each client, so that when the client decides to post his letter, he knows which till to go to. This solution has also the advantage to solve the α-conversion problem. If we know in advance whom we are going to communicate with, we can also decide in advance which "private" channels we are going to share. In a sense we perform α-conversion *before* we start the computation, or, one could say, at *compile time*.

To formalise this intutition we first introduce a variant of CCS that will be interpreted using typed event structures. Our language differs from CCS in many technical details, but the only relevant difference is that synchronisation between actions happens only if the actions share the same confidential names. In a second moment we will see the correspondence between this calculus, and the π-calculus.

5.1. Syntax

Syntactically the calculus we present is very similar to the π-calculus. Communication happens along channels, and information is "passed" along such channels. The difference between the two is in the semantics. In our variant of CCS names are not sent from a process to another: processes decide their confidential names before communicating, and there is not α-conversion. If the chosen names do not coincide, the processes do not synchronise.

Another important technical difference from standard π-calculus and CCS is that we allow infinite parallel composition and infinite restriction. The former is necessary in order to translate replicated processes of the π-calculus. The standard intuition in the π-calculus is that the process $!P$ represents the parallel composition of infinitely many copies of P. We need to represent this explicitly in order to be able to provide each copy with different confidential names. Infinite restriction is also necessary, because we need to restrict all confidential names that are shared between two processes in parallel, and these are in general infinitely many.

We call this language Name Sharing CCS, or NCCS. The syntax is as follows:

P	$::=$	$x\&_{i\in I}\mathtt{in}_i(\tilde{y_i}).P_i$	branching
	$\mid$	$\overline{x}\bigoplus_{i\in I}\mathtt{in}_i(\tilde{y_i}).P_i$	selection
	$\mid$	$x(\tilde{y}).P$	single offer
	$\mid$	$\overline{x}(\tilde{y}).P$	single request
	$\mid$	$\prod_{i\in I} P_i$	parallel composition
	$\mid$	$P \setminus S$	restriction
	$\mid$	$\mathbf{0}$	zero

For the notation, we use conventions analogous to the π-calculus. Processes are identified up to a straightforward structural congruence, which includes the rule $(P\setminus S)\setminus T \equiv P\setminus(S\cup T)$, but no notion of α-equivalence. Names of a process are partitioned into *public* and *confidential*, similarly to the free/bound partition in the π-calculus. The change of name undelines the fact that α-conversion is not allowed.

As for the π-calculus, the fragment of NCCS where the indexing sets of branching and selection are singleton is called *simple*. The fragment where the selection is always a singleton, but the branching is arbitrary is called *deterministic*. The general language is for clarity denoted as the *nondeterministic* fragment.

The operational semantics is completely analogous to the one of CCS, and it is shown in Fig. 8. Labels are the same as for the π-calculus, and synchronisation labels are globally denoted by $\boldsymbol{\tau}$. The main difference with CCS is the presence of the confidential names that are used only for synchronisation. Note also that only the subject of an action is taken into account for restriction.

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

$$\overline{x}(\tilde{y}).P \xrightarrow{\overline{x}(\tilde{y})} P \qquad x(\tilde{y}).P \xrightarrow{x(\tilde{y})} P$$

$$\frac{P \xrightarrow{\beta} P' \quad subj(\beta) \notin S}{P \setminus S \xrightarrow{\beta} P' \setminus S} \qquad \frac{P \xrightarrow{\tau} P'}{P \setminus S \xrightarrow{\tau} P' \setminus S}$$

$$\frac{P_n \xrightarrow{\beta} P'}{\prod_{i\in\mathbb{N}} P_i \xrightarrow{\beta} (\prod_{i\in\mathbb{N}\setminus\{n\}} P_i) \mid P'} \qquad \frac{P_n \xrightarrow{\alpha} P' \quad P_m \xrightarrow{\beta} P''}{\prod_{i\in\mathbb{N}} P_i \xrightarrow{\alpha\bullet\beta} (\prod_{i\in\mathbb{N}\setminus\{n,m\}} P_i) \mid P' \mid P''}$$

Fig. 8. Labelled transition system for Name Sharing CCS.

Example 5.1. For instance the process

$$(x(y).P \mid \overline{x}(z).R) \setminus \{x\}$$

cannot perform any transition, because y and z do not match. The process

$$(x(y).P \mid \overline{x}(y).Q \mid \overline{x}(y).R) \setminus \{x\}$$

can perform two different initial $\boldsymbol{\tau}$ transitions. Since the name x is not bound, it does not become private to the subprocesses involved in the communication. The process

$$\left(x\&_{i\in\{1,2\}}\mathtt{in}_i.P_i \mid \overline{x} \bigoplus_{i\in\{1,2\}} \mathtt{in}_i.R_i\right) \setminus \{x\}$$

can perform, nondeterministically, two $\boldsymbol{\tau}$ transitions to $(P_1 \mid R_1) \setminus \{x\}$ or to $(P_2 \mid R_2) \setminus \{x\}$.

5.2. Typing rules

Using the notions of type and type enviroment presented in Section 4, we are going to present a typing system for NCCS. This typing system is very similar to the one of the π-calculus.

Before introducing the typing rules, we have to define the operation of "parallel composition of environments". This operation intuitevly combine environments for which the only possible shared public names are client requests.

Let Γ_h $h \in H$ be a family of enviroments such that for every name x, either for every h, $\Gamma_h(x) = \biguplus_{k_h\in K_h} \Delta_{k_h}$, or $x \in Dom(\Gamma_h)$ for at most one h. We define $\Gamma = \prod_{h\in H} \Gamma_h$ as follows. If for every h, $\Gamma_h(x) = \biguplus_{k_h\in K_h} \Delta_{k_h}$, then $\Gamma(x) = \biguplus_{k_h\in K_h, h\in H} \Delta_{k_h}$, assuming all the names involved are distinct. If $x \in Dom(\Gamma_h)$ for at most one h, then $\Gamma(x) = \Gamma_h(x)$.

A special case, which will be of particular interest when encoding the π-calculus, is when all the Γ_h are different instances of the same environment, up to renaming of the confidential names. For any set K, let $F_K : Names \to \mathcal{P}(Names)$ be a function such that, for every name x, there is a bijection between K and $F_K(x)$. Concretely we can represent $F_K(x) = \{x^k \mid k \in K\}$. In the following we assume that each set K is associated to a unique F_K, and that for distinct x, y, $F_K(x) \cap F_K(y) = \emptyset$.

Given a type τ, and an index k, define τ^k as follows:

- $\bigotimes_{h\in H}(\tilde{y}_h : \tilde{\tau}_h)^k = \bigotimes_{h\in H}(\tilde{y}_h^k : \tilde{\tau}_h^k)$, where $\tilde{y}_h = (y_{i,h})_{i\in I}$ and $\tilde{y}_h^k = (y_{i,h}^k)_{i\in I}$;
- and similarly for all other types.

Given an environment Γ, we define Γ^k where for every name $x \in Dom(\Gamma)$, $\Gamma^k(x) = \Gamma(x)^k$. The environment $\Gamma[K]$ is defined as $\prod_{k\in K} \Gamma^k$, and is thus defined only when for every $x \in Dom(\Gamma)$, $MD(\Gamma(x)) = \mathbf{?}$. We will also assume that all names in the range of the substitution are fresh, in the sense that no name in the range of F_K appears in the domain of Γ. Under this assumption we easily have that if Γ is well formed and if $\Gamma[K]$ is defined, then $\Gamma[K]$ is also well formed.

We are now ready to write the rules: see Fig. 9. The rule for weakening of the client type tells us that we can request a resource even if we are not actually using it. The rule for the selection tells us that we can choose less than what the types offers. The parallel composition is well typed only if the names used for communication have matching types, and if the matched names are restricted. This makes sure that communication can happen, and that the shared names are indeed private to the processes involved.

5.3. Typed semantics

The relation Γ allows β was defined in Section 4. We also need a definition of the environment $\Gamma \setminus \beta$, similar to the one defined in Section 2.

- $\Gamma \setminus \boldsymbol{\tau} = \Gamma$;
- if $\Gamma = \Delta, x : \&_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)$, then $\Gamma \setminus x\mathtt{in}_i(\tilde{y}_i) = \Delta, \tilde{y}_i : \tilde{\tau}_i$;

$$\frac{}{0 \triangleright \emptyset}\ \text{Zero} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : \uplus_{h\in H} \Gamma_h}\ \text{WeakReq} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : \updownarrow}\ \text{WeakCl}$$

$$\frac{P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i \quad x \notin \Gamma}{x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \&_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)}\ \text{Branch}$$

$$\frac{P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i \quad x \notin \Gamma \quad I \subseteq J}{\overline{x} \oplus_{i\in I} \mathtt{in}_i p_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \oplus_{i\in J}(\tilde{y}_i : \tilde{\tau}_i)}\ \text{Sel}$$

$$\frac{P \triangleright \Gamma, \tilde{w}_j : \tilde{\tau}_j, x : \uplus_{h\in H}(\tilde{w}_h : \tilde{\tau}_h) \quad \tilde{w}_j \text{ fresh}}{\overline{x}(\tilde{w}_j).P \triangleright \Gamma, x : \uplus_{h\in H\uplus\{j\}}(\tilde{w}_h : \tilde{\tau}_h)}\ \text{Req}$$

$$\frac{P_h \triangleright \Gamma_h, \tilde{y}_h : \tilde{\tau}_h \quad a \notin \Gamma}{\prod_{h\in H} x(\tilde{y}_h).P_h \triangleright \prod_{h\in H} \Gamma_h, a : \otimes_{h\in H}(\tilde{y}_h : \tilde{\tau}_h)}\ \text{Offer}$$

$$\frac{P \triangleright \Gamma, x : \tau \quad MD(\tau) = !, \updownarrow}{P \setminus x \triangleright \Gamma}\ \text{Res} \qquad \frac{P_i \triangleright \Gamma_i \quad (i = 1, 2) \quad S = cl(\Gamma_1, \Gamma_2)}{(P_1 \| P_2) \setminus S \triangleright \Gamma_1 \odot \Gamma_2}\ \text{Par}$$

Fig. 9. Typing rules for NCCS.

- if $\Gamma = \Delta, x : \bigoplus_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)$, then $\Gamma \setminus \overline{x}\mathtt{in}_i(\tilde{y}_i) = \Delta, \tilde{y}_i : \tilde{\tau}_i$;
- if $\Gamma = \Delta, x : \bigotimes_{h\in H\uplus\{j\}}(\tilde{y}_h : \tilde{\tau}_h)$, then $\Gamma \setminus x(\tilde{y}_j) = \Delta, \tilde{y}_j : \tilde{\tau}_j, x : \bigotimes_{h\in H}(\tilde{y}_h : \tilde{\tau}_h)$;
- if $\Gamma = \Delta, x : \biguplus_{h\in H\uplus\{j\}}(\tilde{y}_h : \tilde{\tau}_h)$, then $\Gamma \setminus \overline{x}(\tilde{y}_j) = \Delta, \tilde{y}_j : \tilde{\tau}_j, x : \biguplus_{h\in H}(\tilde{y}_h : \tilde{\tau}_h)$.

Note that $\Gamma \setminus \beta$ is defined precisely when Γ allows β. We have the following

Proposition 5.1. *If $P \triangleright \Gamma$, $P \xrightarrow{\beta} Q$ and Γ allows β, then $Q \triangleright \Gamma \setminus \beta$.*

Corollary 5.2 (*Subject Reduction*). *If $P \triangleright \Gamma$, $P \xrightarrow{\tau} Q$ then $Q \triangleright \Gamma$.*

Proposition 5.1 allows us to define the notion of *typed transition*, written $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma'$ by adding the constraint:

$$\frac{P \xrightarrow{\beta} Q \; \Gamma \text{ allows } \beta}{P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma \setminus \beta}$$

We are going to define a notion of bisimulation which is slightly different from one might expect. The reason is that labels, as we have presented them, contain somehow too much information, more than a typed context should recognise. Normal CCS bisimulation would be too fine and our full abstraction result would fail. In principle a label should represent what a context can observe. But a typed context cannot really take apart two processes with different confidential names. Either the context does not synchronise on the subject of the label, and then the confidential names do not matter. Or, if it does synchronise, the typing rules ensure it must do it with the proper confidential names, whatever they are. We want thus to allow processes that use different confidential names to be identified.

In the following ρ will be a fresh injective renaming of the confidential names of an environment Δ. In such a case then $\Delta[\rho]$ is also a well formed environment.

Definition 5.3. Let $\mathcal{R}$ be a symmetric relation between judgments such that if $(P \triangleright \Gamma)\ \mathcal{R}\ (P' \triangleright \Gamma')$, then $\Gamma' = \Gamma[\rho]$, for some injective renaming ρ. We say that $\mathcal{R}$ is a *bisimulation up to renaming* if the following is satisfied:

- whenever $(P \triangleright \Gamma)\ \mathcal{R}\ (P' \triangleright \Gamma')$, $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma \setminus \beta$, then there exists a renaming ρ and a process Q' such that $P'[\rho] \triangleright \Gamma'[\rho] \xrightarrow{\beta} Q' \triangleright \Gamma[\rho' \circ \rho] \setminus \beta$, and $(Q \triangleright \Gamma \setminus \beta)\ \mathcal{R}\ (Q' \triangleright \Gamma'[\rho] \setminus \beta)$.

If there exists a bisimulation up to renaming between two judgments, we say that they are bisimilar $(P \triangleright \Gamma) \approx (P' \triangleright \Gamma')$.

6. Event structure semantics of Name Sharing CCS

6.1. Semantics of nondeterministic NCCS

The event structure semantics of typed NCCS is presented in Fig. 10. It is given in terms of labelled event structures, using the operations, in particular the parallel composition, as defined in Section 3.5. This construction is perfectly analogous to

$$\begin{aligned}
[\![0 \triangleright \emptyset]\!] &= \emptyset \\
[\![P \triangleright \Gamma, x : \uplus_{h\in H} \Gamma_h]\!] &= [\![P \triangleright \Gamma]\!] \\
[\![P \triangleright \Gamma, x : \updownarrow]\!] &= [\![P \triangleright \Gamma]\!] \\
[\![P \setminus x \triangleright \Gamma]\!] &= [\![P \triangleright \Gamma, x : \tau]\!] \setminus \{x\} \\
[\![\overline{x} \oplus_{i\in I} \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \oplus_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)]\!] &= \textstyle\sum_{i\in I} \overline{x}\mathtt{in}_i(\tilde{y}_i).[\![P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i]\!] \\
[\![x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \&_{i\in I}(\tilde{y}_i : \tilde{\tau}_i)]\!] &= \textstyle\sum_{i\in I} x\mathtt{in}_i(\tilde{y}_i).[\![P_i \triangleright \Gamma, \tilde{y}_i : \tilde{\tau}_i]\!] \\
[\![\overline{x}(\tilde{y}).P \triangleright \Gamma, x : \uplus_{k\in K\uplus\{j\}}(\tilde{y}_k : \tilde{\tau}_k)]\!] &= \overline{x}(\tilde{y}).[\![P \triangleright \Gamma, x : \uplus_{k\in K}(\tilde{y}_k : \tilde{\tau}_k), \tilde{y}_j : \tilde{\tau}_j]\!] \\
[\![\textstyle\prod_{k\in K} x(\tilde{y}_k).P_k \triangleright \prod_{k\in K}\Gamma_k, x : \bigotimes_{k\in K}(\tilde{y}_k : \tilde{\tau}_k)]\!] &= \textstyle\prod_{k\in K} x(\tilde{y}_k).[\![P_k \triangleright \Gamma_k, \tilde{y}_k : \tilde{\tau}_k]\!] \\
[\![(P_1 \| P_2) \setminus S \triangleright \Gamma_1 \odot \Gamma_2]\!] &= [\![P_1 \triangleright \Gamma_1]\!] \mid [\![P_2 \triangleright \Gamma_2]\!] \setminus (Dis(\Gamma_1 \odot \Gamma_2))
\end{aligned}$$

Fig. 10. Denotational semantics of simple Name Sharing CCS.

the one in [45], the only difference being the synchronisation algebra. However, since the synchronisation algebra is the same for both the operational and the denotational semantics, we obtain automatically the correspondence between the two, as in [45].

In the parallel composition, we have to restrict all the channels that are subject of communication. More generally, we need to restrict all the actions that are not allowed by the new type environment.

The main property of the semantics is that the denotation of a typed process is a typed event structure. In particular all denoted event structures are confusion free.

Theorem 6.1. *Let P be a process and Γ an environment such that $P \triangleright \Gamma$. Then $[\![P \triangleright \Gamma]\!]$ is confusion free, and $[\![P \triangleright \Gamma]\!] \triangleright [\![\Gamma]\!]$.*

6.2. Semantics of deterministic NCCS

The syntax of NCCS introduces the conflict explicitly, therefore we cannot obtain conflict free event structures. The result above shows that no new conflict is introduced through synchronisation. Moreover, in the deterministic fragment, synchronisation does indeed resolve the conflicts.

First it is easy to show that the semantics of deterministic NCCS is in term of deterministic event structures:

Proposition 6.2. *Suppose P is a deterministic process, and that $P \triangleright \Gamma$. Then $[\![P \triangleright \Gamma]\!]$ is deterministic.*

The main theorem is the following, which justifies the term "deterministic". It states that once all choices have been matched with selections, or cancelled out, what remains is a conflict free event structure.

Theorem 6.3. *If let X be the set of names in P, then $[\![P \triangleright \Gamma]\!] \setminus X$ is a conflict free event structure.*

Corollary 6.4. *If $[\![\Gamma]\!] = \emptyset$, then $[\![P \triangleright \Gamma]\!]$ is conflict free.*

6.3. Semantics of simple NCCS

Although the syntax of NCCS does not introduce directly any conflict, there is in principle the possibility that conflict is introduced by the parallel composition. The typing system is designed in such a way that this is not the case.

Theorem 6.5. *Suppose P is a simple process such that $P \triangleright \Gamma$. Then $[\![P \triangleright \Gamma]\!]$ is conflict free.*

6.4. Correspondence between the semantics

In order to show the correspondence between the operational and the denotational semantics, we invoke Winskel and Nielsen's handbook chapter [45]. Note that our semantics are a straightforward modification of the standard CCS semantics. This is the main reason why we chose the formalism presented in this paper: we wanted to depart as little as possible from the treatment of [45].

The main difference is that typed semantics modifies the behaviour, by forbidding some of the actions. However this modification acts precisely as a special form of name restriction: in the labelled transition system it blocks some action, while

in event structures it cancel them out (together with all events enabled by them). With a straightforward generalisation of the notion of restriction, we then preserve the correspondence between the two semantics and the proof technique of [45] carries over. In particular we have

Theorem 6.6. *Take two typed NCCS processes $P \rhd \Gamma$, $Q \rhd \Gamma$. Suppose that $[\![P \rhd \Gamma]\!] = [\![Q \rhd \Gamma]\!]$, then $P \rhd \Gamma \approx Q \rhd \Gamma$.*

This theorem is the best result we can get: indeed, as for standard CCS, we cannot expect the event structure semantics to be fully abstract. Bisimilarity is a "interleaving" semantics, equating the two processes $\boldsymbol{\tau} \| \boldsymbol{\tau}$ and $\boldsymbol{\tau}.\boldsymbol{\tau}$, which have different event structure semantics.

A more direct correspondence is described next. Recall the way we derive a transition system from an event structure, as presented in Section 3: if $\lambda(e) = \beta$, then $\mathcal{E} \xrightarrow{\beta} \mathcal{E} \lfloor e$. We can therefore state the following correspondence:

Theorem 6.7. *Let $\cong$ denote isomorphism of labelled event structures;*

- *if $P \rhd \Gamma \xrightarrow{\beta} P' \rhd \Gamma \setminus \beta$, then $[\![P \rhd \Gamma]\!] \xrightarrow{\beta} \cong [\![P' \rhd \Gamma \setminus \beta]\!]$.*
- *if $[\![P \rhd \Gamma]\!] \xrightarrow{\beta} \mathcal{E}'$ then $P \rhd \Gamma \xrightarrow{\beta} P' \rhd \Gamma \setminus \beta$ and $\mathcal{E}' \cong [\![P' \rhd \Gamma \setminus \beta]\!]$.*

The proof is by induction on the operational rules. The only difficult case is the parallel composition.

7. Correspondence between the calculi

7.1. Translation

We are now going to present a fully abstract translation of the π-calculus into Name Sharing CCS. The translation is parametrised over a fixed choice for the confidential names. This parametrisation is necessary because π-calculus terms are identified up to α-conversion, and so the identity of bound names is irrelevant, while in Name Sharing CCS, the identity of confidential names is important. Also, since servers are interpreted as infinite parallel compositions, every bound name of a server must correspond to infinitely many names in the interpretation.

The translation is a family of functions $\{\!\{-\}\!\}^{\rho}$, that take a judgment of the π-calculus and return a process of NCCS. The semantic functions are indexed by a "choice" function ρ that for every bound name assigns a set (possibly a singleton) of fresh distinct names. In order to make this work, we have to use the convention that all bound names in the π-calculus are distinct, and different from the free names. In this way ρ cannot identify two different bindings. Although the translation is defined for all ρ, the target process will not always be typable. In particular, for some choice of renaming, the parallel composition in NCCS will not be typed.

We define the translation by induction on the derivation of the typing judgment. Without loss of generality, we will assume that all the weakenings are applied to the empty process. The translation is defined in Fig. 11.

The notation has to be explained. The notation $\rho, y \to S$ denotes the function ρ extended on a name y not already in the domain of ρ, and such that all names in S are fresh and distinct from any other name in the range of ρ. In the translation of the server, we use Y to denote the set of confidential names of the translation of P. We also use the choice function $\rho[K]$ defined as follows: assume the range of ρ are only singletons, say for every name x in the domain, $\rho(x) = \{y\}$. Then $\rho[K](x) = \{y_k \mid k \in K\}$, where y_k are obtained by a function $F_K : \mathit{Names} \to \mathcal{P}(\mathit{Names})$ as in Section 5. In the translation of the parallel composition, S denotes the set of names that are in the range of both ρ_1 and ρ_2.

Once past the rather heavy notation, the translation is rather simple. Note the way bound variables become confidential names. Observe also that the server is translated into an infinite parallel composition.

We said that the translation is not always typable. In particular, for the wrong choice of ρ_1, ρ_2, the parallel composition may not be typed because the chosen confidential names may not match. However it is always possible to find suitable ρ_1, ρ_2. Intuitively we can say that in translating typed π into typed NCCS, we perform α-conversion "at compile time".

Lemma 7.1. *For every judgment $P \rhd \Gamma$ in the π-calculus, there exists a choice function ρ and a type environment Δ such that $\{\!\{P \rhd \Gamma\}\!\}^{\rho} \rhd \Delta$. Moreover, for every injective fresh renaming ρ', if $\{\!\{P \rhd \Gamma\}\!\}^{\rho} \rhd \Delta$ then $\{\!\{P \rhd \Gamma\}\!\}^{\rho' \circ \rho} \rhd \Delta[\rho']$.*

Example 7.1. We demonstrate how the process which generates an infinite behaviour with infinite new name creation is interpreted into NCCS. Consider the process $\mathtt{Fw}(ab) = !a(x).\overline{b}(y).y.\overline{x}$. This agent links two locations 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 $P_{\omega} \rhd (a : \tau, b : \overline{\tau}) \odot (b : \tau, a : \overline{\tau})$, that is $P_{\omega} \rhd a, b : \tau$. One possible translation for $\mathtt{Fw}(ab) \rhd a : (()^{\uparrow})^{!}, b : (()^{\downarrow})^{?}$ is

$$Q_1 = \prod_{k \in K} a(x^k).\overline{b}(y^k).y^k.\overline{x}^k \rhd a : \bigotimes_{k \in K} (x^k : ()^{\uparrow}), b : \biguplus_{k \in K} (y^k : ()^{\downarrow})$$

while for $\mathtt{Fw}(ba) \rhd b : (()^{\uparrow})^{!}, a : (()^{\downarrow})^{?}$ is

$$Q_2 = \prod_{h \in H} b(z^h).\overline{a}(w^h).w^h.\overline{z}^h \rhd b : \bigotimes_{h \in H} (z^h : ()^{\uparrow}), a : \biguplus_{h \in H} (()^{?} w^h : ()^{\downarrow})$$

$$\{\!\{0 \triangleright x_i : (\tau_i)^?, y_j :\updownarrow\}\!\}^\rho = 0$$

$$\{\!\{(\boldsymbol{\nu}\, x)P \triangleright \Gamma\}\!\}^\rho = \{\!\{P \triangleright \Gamma, x : \tau\}\!\}^\rho \setminus x$$

$$\{\!\{P_1 \| P_2 \triangleright \Gamma_1 \odot \Gamma_2\}\!\}^{\rho_1 \cup \rho_2} = (\{\!\{P_1 \triangleright \Gamma_1\}\!\}^{\rho_1} \| \{\!\{P_2 \triangleright \Gamma_2\}\!\}^{\rho_2}) \setminus S$$

$$\{\!\{\overline{x} \oplus_{i\in I} \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \oplus_{i\in I}(\tilde{\tau}_i)^\uparrow\}\!\}^{\rho,(\tilde{y}_i \to \tilde{z}_i)_{i\in I}} = \overline{x} \oplus_{i\in I} \mathtt{in}_i(\tilde{z}_i).\{\!\{P_i[\tilde{z}_i/\tilde{y}_i] \triangleright \Gamma, \tilde{z}_i : \tau_i\}\!\}^\rho$$

$$\{\!\{x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{y}_i).P_i \triangleright \Gamma, x : \&_{i\in I}(\tilde{\tau}_i)^\downarrow\}\!\}^{\rho,(\tilde{y}_i \to \tilde{z}_i)_{i\in I}} = x \,\&_{i\in I}\, \mathtt{in}_i(\tilde{z}_i).\{\!\{P_i[\tilde{z}_i/\tilde{y}_i] \triangleright \Gamma, \tilde{z}_i : \tau_i\}\!\}^\rho$$

$$\{\!\{!x(\tilde{y}).P \triangleright \Gamma, x : (\tilde{\tau})^!\}\!\}^{\rho[K],\tilde{y}\to\{\tilde{y}_k\}_{k\in K}} = \textstyle\prod_{k\in K} x(\tilde{y}^k).\{\!\{P \triangleright \Gamma\}\!\}^\rho[\tilde{y}^k/\tilde{y}][Y^k/Y]$$

$$\{\!\{\overline{x}(\tilde{y}).P \triangleright \Gamma, x : (\tilde{\tau})^?\}\!\}^{\rho,\tilde{y}\to\tilde{w}} = \overline{x}(\tilde{w}).\{\!\{P \triangleright \Gamma, x : (\tilde{\tau})^?[\tilde{w}/\tilde{y}]\}\!\}^\rho$$

Fig. 11. Translation from π to NCCS.

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 we can compose the two environments. Therefore the translation of $P_\omega \triangleright a, b : \tau$ is $(Q_1 \mid Q_2) \setminus S \triangleright \Delta$ for

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

The reader can check that any transition of P_ω is matched by a corresponding transition of its translation. This is what we formally show next.

7.2. Full abstraction

To show the correctness of the translation, we first prove the correspondence between the labelled transition semantics. If ρ is a choice function and S is a set of names, by $\rho \setminus S$ we denote the function ρ restricted to the names not in S.

Theorem 7.2. *Suppose $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ in the π-calculus, then there exists ρ and Δ such that $\{\!\{P \triangleright \Gamma\}\!\}^\rho \triangleright \Delta$ and $\{\!\{P \triangleright \Gamma\}\!\}^\rho \triangleright \Delta \xrightarrow{\beta} \{\!\{P' \triangleright \Gamma \setminus \beta\}\!\}^{\rho\setminus obj(\beta)} \triangleright \Delta \setminus \beta$.*

Conversely, suppose $\{\!\{P \triangleright \Gamma\}\!\}^\rho \triangleright \Delta \xrightarrow{\beta} Q \triangleright \Delta \setminus \beta$. Then there exists P' such that $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ and $\{\!\{P' \triangleright \Gamma \setminus \beta\}\!\}^{\rho\setminus obj(\beta)} = Q$.

The full abstraction is then a corollary.

Theorem 7.3 (*Full Abstraction*). *We have $P \triangleright \Gamma \approx P' \triangleright \Gamma$ if and only if for some ρ, ρ', Δ, Δ' we have $\{\!\{P \triangleright \Gamma\}\!\}^\rho \triangleright \Delta \approx \{\!\{P' \triangleright \Gamma\}\!\}^{\rho'} \triangleright \Delta'$.*

Recall that in NCCS we use bisimilarity up to renaming.

7.3. Event structure semantics of the π-calculus

By composing the translation obtained in this section with the event structure semantics of Section 6, we obtain an event structure semantics of the π-calculus.

Given a π-calculus judgment $P \triangleright \Gamma$, we define

$$[\![P \triangleright \Gamma]\!]^\rho_\Delta = [\![\{\!\{P \triangleright \Gamma\}\!\}^\rho \triangleright \Delta]\!]$$

We thus have

Lemma 7.4. *For every judgment $P \triangleright \Gamma$ in the π-calculus, there exist ρ and Δ such that $[\![P \triangleright \Gamma]\!]^\rho_\Delta$ is defined. When this is the case $[\![P \triangleright \Gamma]\!]^\rho_\Delta$ is a confusion free event structure, and $[\![P \triangleright \Gamma]\!]^\rho_\Delta \triangleright \Delta$.*

Proposition 7.5 (*Soundness*). *Suppose that for some ρ, ρ', Δ, Δ', $[\![P \triangleright \Gamma]\!]^\rho_\Delta = [\![P' \triangleright \Gamma]\!]^{\rho'}_{\Delta'}$. 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 [41], hence the other direction does not hold in our case either.

However, there is another kind of correspondence between the labelled transition systems and the event structures, analogous to the one discussed in Section 6.4. Combining Theorem 6.7 with Theorem 7.2, we obtain:

Theorem 7.6. *Suppose* $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ *in the* π*-calculus, and that* $[\![P \triangleright \Gamma]\!]^{\rho}_{\Delta}$ *is defined. Then* $[\![P \triangleright \Gamma]\!]^{\rho}_{\Delta} \xrightarrow{\beta} \cong [\![P' \triangleright \Gamma \setminus \beta]\!]^{\rho\setminus obj(\beta)}_{\Delta\setminus\beta}$.

Conversely, suppose $[\![P \triangleright \Gamma]\!]^{\rho}_{\Delta} \xrightarrow{\beta} \mathcal{E}'$. *Then there exists* P' *such that* $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ *and* $[\![P' \triangleright \Gamma \setminus \beta]\!]^{\rho\setminus obj(\beta)}_{\Delta\setminus\beta} \cong \mathcal{E}'$.

8. 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 [28] 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 an 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 below.

8.1. Subsequent work

Since this work was first presented, some followup have appeared. In [38], we extended this semantics the probabilistic π-calculus [21,11], by using a typed version of probabilistic event structures [36]. In [12], together with Silvia Crafa, we also extended the semantics to an untyped version of the internal π-calculus. We are currently working to a semantics for the full π-calculus, where free outputs are allowed. The main difficulty seems to be the handling of scope extrusion.

8.2. Future work

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 [27], and with ludics nets [17].

An event structure *terminates* if all its maximal configurations are finite. It would be interesting to study a typing system of event structures that guarantees termination applying the idea of the strongly normalising typing system of the π-calculus [47].

8.3. Related work

There are several causal models for the π-calculus, that use different techniques. In [5,14], 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 [10], a more abstract approach is followed, which involves indexed transition systems. In [24], a semantics of the π-calculus in terms of pomsets is given, following ideas from dataflow theory. The two papers [9,16] present Petri nets semantics of the π-calculus. Since we can unfold Petri nets into event structures, these could indirectly provide an 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 [30]. In [7], 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.

A recent work [6] claims to provide an event structure semantics of the full π-calculus. However they cater only for the *reduction* semantics. Consequently their semantics is not compositional, nor it is an extension of Winskel's semantics of CCS.

In [44], 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 [43] is moving in that direction.

Some interesting results connecting game semantics and event structures can be found in [18]. A fundamental work on the connections between the linear π-calculus and polarised linear logic is [22]. See also [19].

Infinite behaviour is introduced in our version of CCS by means of the infinite parallel composition. Infinite parallel composition is similar to replication in that it provides infinite behaviour "in width" rather that "in depth". Recent studies on recursion versus replication are [8,20].

Acknowledgements

We want to thank the anonymous referees for their comments. We acknowledge the support of the EPSRC grant GR/T04724/01 "Program Analysis and the Typed Pi-Calculus". The second author is also partially supported by EPSRC Advanced Fellowship GR/T03208/01 and EPSRC grant EP/F003757/1.

Appendix. Proofs

A.1. Proof of Lemma 3.6

We prove it by induction on the joint size of x, x'. The base case is vacuously true. Now take $(x, e_1, e_2), (x', e_1, e_2) \in E$ with $x \neq x'$. Since x, x' are downward closed sets, if their maximal elements coincide, they coincide. Therefore, w.l.o.g. there must be a maximal element $(y, d_1, d_2) \in x$ such that $(y, d_1, d_2) \notin x'$. By definition of E, and without loss of generality, we can assume that $d_1 \in parents(e_1)$. Therefore, by definition of E, there must be a $(y', d_1, d_2') \in x'$. Suppose $d_2 \neq d_2'$. Then by definition of conflict $(y, d_1, d_2) \smile (y', d_1, d_2')$. If $d_2 = d_2'$ then it must be $y \neq y'$. Then by induction hypothesis there exist $f \in y, f' \in y'$ such that $f \smile f'$. And since x, x' are downward closed, we have $f \in x, f' \in x'$.

A.2. Proof of Theorem 3.7

Recall the the definition of $(E, \leq, \smile)$. In order to show that it is an event structure, we first o have to show that the relation $\leq$ is a partial order. We have that

- it is reflexive by construction;
- it is antisymmetric: suppose $e' \leq e = (x, e_1, e_2)$. If $e' \neq e$, then, by construction $h(e') < h(e)$, so that it cannot be $e \leq e'$.
- it is transitive: suppose $e' \leq e \leq d = (y, d_1, d_2)$. This means that $e \in y$. Since, by construction, y is downward closed, this means that $e' \in y$, so that $e' \leq d$.

Next, for every event $e = (x, e_1, e_2)$, we have that $[e)$ is finite, as it coincides with x.

Then we need to show that the conflict is irreflexive and hereditary. It is hereditary essentially by definition: suppose $e := (x, e_1, e_2) \smile d := (y, d_1, d_2)$, and let $d \leq d' := (y', d_1', d_2')$. By considering all the cases of the definition of $e \smile d$, we derive $e \smile d'$. For instance, suppose there exists $e' := (x', e_1', e_2') \leq e$ such that $e_1' \asymp d_1$, and $e' \neq d$. This means that $e' \smile d$. Notice that $e' \leq e$, and $d \leq d'$. By the fourth clause of the definition, $e \smile d'$. The other cases are analogous.

To prove that the conflict relation is irreflexive, suppose $(x, e_1, e_2) \smile (x, e_1, e_2)$. There are two possible ways of deriving this. First, if there are $e, d \in x$ such that $e \smile d$, but this contradicts the fact that x is a configuration. The other possibility is that, there exist $(x', e_1', e_2') \in x$ such that $(x', e_1', e_2') \smile (x, e_1, e_2)$. Take a minimal such. Then it must be $e_1' \asymp e_1$ or $e_2' \asymp e_2$. But this contradicts the definition of E.

Now we have to show that such event structure is the categorical product of $\mathcal{E}_1, \mathcal{E}_2$. First thing to show is that projections are morphisms. Using Proposition 3.5, it is enough to show that they reflect reflexive conflict and preserves downward closure.

- Take $e, e' \in E$ and suppose by that $\pi_1(e) \asymp \pi_1(e')$. Then, by definition we have $e \asymp e'$.
- To show that π_1 preserves downward closure let $e = (x, e_1, e_2)$ suppose $e_1' \leq e_1 = \pi_1(e)$. Then we show that there is a $e' \leq e$ such that $\pi_1(e') = e_1'$. By induction on the height of e: the basis is vacuously true, since e_1 is minimal. For the step, consider first the case where $e_1' \in parents(e_1)$. Then, by definition of E, we have that there exists $e' = (x', e_1', e_2') \in x$. Therefore $e' \leq e$ and $\pi_1(e') = e_1'$. If $e_1' \notin parents(e_1)$, then there is a $e_1'' \in parents(e_1)$ such that $e_1' \leq e_1'' \leq e_1$ so that there is $e'' = (x'', e_1'', e_2'') \in x$. By induction hypothesis there is $e' \in x''$ such that $\pi_1(e') = e_1'$. And by transitivity, $e' \leq e$.

Now we want to show that $\mathcal{E}$ enjoys the universal property that makes it a categorical product. That is for every event structure $\mathcal{D}$, such that there are morphisms $f_1 : \mathcal{D} \to \mathcal{E}_1, f_2 : \mathcal{D} \to \mathcal{E}_2$, there exists a unique $f : \mathcal{D} \to \mathcal{E}$ such that $\pi_1 \circ f = f_1$ and $\pi_2 \circ f = f_2$.

Clearly, if such f exists, it must be defined as $f(d) = (x, f_1(d), f_2(d))$, for some x. By this we mean $f(d) = (x, f_1(d), *)$, if $f_2(d)$ is undefined, $f(d) = (x, *, f_2(d))$, if $f_1(d)$ is undefined, and undefined if both are undefined. We now define x, by induction on the size of $[d)$. Suppose d is minimal. Then, since f_1, f_2 are morphisms and thus preserve downward closure, we have that $f_1(d), f_2(d)$ are both minimal. Since every maximal element of x must contain the parent of at least one of them, the only possibility is that x be empty.

Putting $f(d) = (\emptyset, f_1(d), f_2(d))$, we obtain, that, on the elements of height 0,

- $f(d)$ is uniquely defined: we have seen that all choices are forced.
- f reflects reflexive conflict: suppose $(\emptyset, f_1(d), f_2(d)) \asymp (\emptyset, f_1(d'), f_2(d'))$, then either $f_1(d) \asymp f_1(d')$ or $f_2(d) \asymp f_2(d')$. In the first case, since f_1 is a morphism, and thus reflects reflexive conflict, we have $d \asymp d'$. Symmetrically for the other case.
- f preserves downward closure vacuously.

Now suppose f is uniquely defined for all elements of height less or equal than n, it reflects reflexive conflict and preserves downward closure. Consider d of height $n + 1$. We want to define $f(d) = (x, f_1(d), f_2(d))$. Define x as follows. For a set A, let $\downarrow A$ be the downward closure of A. Let $X = \{f(d') \mid d' < d \,\&\, [\ f_1(d') \in \mathit{parents}(f_1(d)) \text{ or } f_2(d') \in \mathit{parents}(f_2(d))\ ]\}$ and define x as $\downarrow X$. First of all we should check that this is indeed an element of E. x is downward closed by definition. It is finite because X is and each element of X has finitely many predecessors. Suppose there are $d', d'' < d$ such that $f(d') \smile f(d'')$. We know by induction that f reflects reflexive conflict on elements of height smaller than d, which means that $d' \smile d''$, contradiction.

Now the maximal elements of x contain either a parent of $f_1(d)$ or a parent of $f_2(d)$ by construction. Take a parent e_1 of $f_1(d)$. I claim that e_1 is of the form $f_1(d')$ for some $d' < d$. Since $e_1 \in \mathit{parents}(f_1(d))$, in particular $e_1 \leq f_1(d)$. since f_1 preserves downward closure, there must exists d' as above. Thus all parents are represented in X. Finally, suppose there is $(z, e_1, e_2) \in x$ such that $e_1 \asymp f_1(d)$ or $e_2 \asymp f_2(d)$. If $(z, e_1, e_2) \in X$, then $(z, e_1, e_2) = f(d')$ for some $d' < d$. So that $e_1 = f_1(d')$, and $e_2 = f_2(d')$. Since f_1, f_2 reflect reflexive conflict, we would have $d' \smile d$, contradiction. Otherwise there must be $f(d') \in X$ such that $(z, e_1, e_2) < f(d')$. Since f preserves downward closure on elements of height less or equal than n, there must be $d'' < d'$ such that $f(d'') = (z, e_1, e_2)$. As above we conclude $d'' \smile d$, contradiction.

Thus putting $f(d) = (x, f_1(d), f_2(d))$, we have that f is well defined on d. Moreover

- $f(d)$ is uniquely defined: suppose we have another possible x. Since f must preserve downward closure, for all $e \in x$, we have that $e = f(d')$ for some $d' < d$. Now, suppose there is an element $f(d') \in X$ which is not in x. W.l.o.g assume that $f_1(d') \in \mathit{parents}(f_1(d))$. Then, there must be an element $e' = (y, f_1(d'), d'_2)$ maximal in x. By the observation above it must be $e' = f(d')$, contradiction.
- f preserves downward closure: take d, and consider $e \leq f(d)$. By construction, either $e \in X$, in which case we have $e = f(d')$ for some $d' < d$, of $e \leq e' \in X$, in which case we have $e' = f(d')$ for $d' < d$. Since, by induction f preserves downward closure, we have $e = f(d'')$ for $d'' < d' < d$.
- f reflects reflexive conflict: suppose $(x, f_1(d), f_2(d)) \asymp (x', f_1(d'), f_2(d'))$, then
  - either $f_1(d) \asymp f_1(d')$ or $f_2(d) \asymp f_2(d')$. In either case, since f_1, f_2 reflects reflexive conflict, we have $d \asymp d'$.
  - there exists $(x'', e_1, e_2) \leq (x', f_1(d'), f_2(d'))$, such that $f_1(d) \asymp e_1$ or $f_2(d) \asymp e_2$. Since f preserves downward closure, we have $(x'', e_1, e_2) = f(d'')$ for some $d'' < d'$ and we reason as above.
  - the symmetric case is similar
  - there exists $(y, e_1, e_2) \leq (x, f_1(d), f_2(d))$ and there exists $(y', e'_1, e'_2) \leq (x', f_1(d'), f_2(d'))$, and the reasoning is as above, using that f preserves downward closure.

Thus f is a morphism, is uniquely defined for every $d \in D$, and commutes with the projections. This concludes the proof.

A.3. Proof of Proposition 4.1

Consider a minimal element of $[\![\Gamma_1]\!]$.

- If it synchronises, by the condition on the definition of $\Gamma_1 \odot \Gamma_2$, it must synchronise with a dual minimal element in $[\![\Gamma_2]\!]$. Every event above these two events is either a $\boldsymbol{\tau}$, or it is not allowed, therefore it is deleted by the restriction.
- If it does not synchronise it is left alone, with all above it not synchronising either, and not being restricted.

We can think of $[\![\Gamma_1 \odot \Gamma_2]\!]$, as a disjoint union of $[\![\Gamma_1]\!]$, $[\![\Gamma_2]\!]$, plus some hiding.

A.4. Proof of Lemma 4.3

Suppose $\mathcal{E} \triangleright \Gamma$, witnessed by a morphism $f : \mathcal{E} \to [\![\Gamma]\!]$.

- Let $e, e' \in E$ be such that $\lambda(e) = \lambda(e') \neq \boldsymbol{\tau}$. Therefore, by uniqueness of the labels in $[\![\Gamma]\!]$, $f(e) = f(e')$, and since f reflects reflexive conflict, we have $e \smile e'$.

- A similar reasoning applies for the case when $\lambda(e) = a\mathtt{in}_i(\tilde{x})$ and $\lambda(e') = a\mathtt{in}_j(\tilde{y})$. Then $f(e), f(e')$ belong to the same cell, and thus they are in conflict. Since f reflects conflict, we have $e \smile e'$.
- Suppose $E \triangleright \Gamma$, and let $e, e' \in E$ be such that $e \smile_\mu e'$. Then they belong to the same cell, and by definition they must have same subject but different branches.

A.5. Proof of Theorem 4.4

Define $\Gamma = \Gamma_1 \odot \Gamma_2$ Suppose $\mathcal{E}_1 \triangleright \Gamma_1$, and $\mathcal{E}_2 \triangleright \Gamma_2$. Let $\mathcal{E} = (\mathcal{E}_1 \| \mathcal{E}_2) \setminus Dis(\Gamma)$. We invite the reader to review the definition of the product of event structures, and the consequent definition of parallel composition.

Lemma A.1. *Let (x, e_1, e_2), (y, d_1, d_2) be two events in $\mathcal{E}$. Suppose $(x, e_1, e_2) \smile (y, d_1, d_2)$. Then there exists $(x', e_1', e_2') \leq x$, $(y', d_1', d_2') \leq y$ such that either $e_1' \smile_\mu d_1'$ or $d_1' \smile_\mu d_2'$.*

We check this by cases, on the definition of conflict.

- $e_1 \smile d_1$. In this case there must exists $e_1' \leq e_1$ and $e_2' \leq e_2$ such that $e_1' \smile_\mu e_2'$. Since projection are morphisms of event structures, and since in particular preserve configurations, for every event f below e_1 there must be an event in E below (x, e_1, e_2) that is projected onto f. And similar for d_1. Therefore there are $(x', e_1', e_2') \in x$, $(y', d_1', d_2') \in y$ for some x', y', e_2', d_2'. Note also that $(x', e_1', e_2') \smile (y', d_1', d_2')$.
- $e_2 \smile d_2$ is symmetric.
- $e_1 = d_1$ and $e_2 \neq d_2$. This is the crucial case, where we use the typing. In this case it is not possible that $e_2 = *$ and $d_2 \neq *$ (nor symmetrically). This is because of the typing. If the label dual of e_1 is not in Γ_2 then both $e_2, d_2 = *$. If the label dual of e_1 is in Γ_2, then the label of e_1 is matched and thus it becomes disallowed, so that the event $(x, e_1, *)$ is removed. So both e_2 and d_2 have the same label (the dual of the label of e_1). Thus they are mapped on the same event in $[\![\Gamma_2]\!]$, and thus they must be in conflict. Then we reason as above.
- $e_2 = d_2$ and $e_1 \neq d_1$ is symmetric.
- $e_1 = d_1$ and $e_2 = d_2$. Then the conclusion follow from stability (Lemma 3.6).
- suppose there exists $(\bar{x}, \bar{e_1}, \bar{e_2} \in x$ such that $\bar{e_1} \asymp d_1$ or $\bar{e_2} \asymp d_2$. Then we reason as above to find $(x', e_1', e_2') \in \bar{x}$, $(y', d_1', d_2') \in y$ such that either $e_1' \smile_\mu d_1'$ or $d_1' \smile_\mu d_2'$. Note that, by transitivity, $(x', e_1', e_2') \in x$.
- the symmetric case is analogous.
- Suppose there is $e \in x$, and $d \in y$ such that $e \smile d$. By wellfoundedness this case can be reduce to one of the previous ones.

Lemma A.2. *If $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$, then their labels have the same subject, but different branches and different confidential names.*

By Lemma A.1, either $e_1 \asymp_\mu d_1$ or $e_2 \asymp_\mu d_2$ (or both). In the first case, the labels of e_1, d_1 have the same subject. Thus the labels of (x, e_1, e_2), (y, d_1, d_2) also have the same subject (whether they are synchronisation labels or not). The second case is symmetric.

Lemma A.3. *If $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$, then $x = y$.*

First suppose $e_2 = d_2 = *$. Then $e_1 \asymp_\mu d_1$. Dually when $e_1 = d_1 = *$. Finally, suppose $e_1, d_1, e_2, d_2 \neq *$. Without loss of generality we have $e_1 \asymp_\mu d_1$. But then $e_2 \asymp d_2$, because they have dual labels. Then it must be $e_2 \asymp_\mu d_2$ because otherwise we would not have $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$.

In all cases we have that $(x, d_1, d_2) \in E$. Indeed it satisfies the condition for being in the product (because $parents(e_1) = parents(d_1)$ and $parents(e_2) = parents(d_2)$), and it is allowed if and only if (x, e_1, e_2) is allowed. Suppose $x \neq y$. By stability we have that there are $e' \in x$, $d' \in y$ such that $e' \smile d'$. Which contradicts $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$.

Lemma A.4. *The relation $\asymp_\mu$ is transitive in $\mathcal{E}$.*

Suppose $(x, e_1, e_2) \asymp_\mu (y, d_1, d_2)$, and $(y, d_1, d_2) \asymp_\mu (z, g_1, g_2)$, Then reasoning as above we have that $e_1 \asymp_\mu d_1 \asymp_\mu g_1$ and $e_2 \asymp_\mu d_2 \asymp_\mu g_2$. Which implies $e_1 \asymp_\mu g_1$ and $e_2 \asymp_\mu g_2$, from which we derive $(x, e_1, e_2) \asymp_\mu (z, g_1, g_2)$.

Lemmas A.3 and A.4 together prove that $\mathcal{E}$ is confusion free.

To prove that $\mathcal{E} \triangleright \Gamma$, suppose $f_1 : E_1 \to [\![\Gamma_1]\!]$ and $f_2 : E_2 \to [\![\Gamma_2]\!]$. Recall that $[\![\Gamma]\!] = ([\![\Gamma_1]\!] \| [\![\Gamma_2]\!]) \setminus (Dis(\Gamma) \cup \boldsymbol{\tau})$. As we observed we can think of $[\![\Gamma]\!]$ as the disjoint union of $[\![\Gamma_1]\!]$ and $[\![\Gamma_2]\!]$, plus some hiding.

We define the following partial function $f : \mathcal{E} \to [\![\Gamma]\!]$. $f(x, e_1, *) = f_1(e_1), f(x, *, e_2) = f_2(e_2)$ (where by equality we mean *weak equality*), and undefined otherwise. We have to check that f satisfies the conditions required. The first two conditions are a consequence of (the proof) of the first part of the theorem. It remains to show that f is a morphism of event structures. This follows from general principles, but we repeat the proof here.

We have to check that if $d \leq f(x, e_1, e_2)$ in $[\![\Gamma]\!]$, then there exists (y, d_1, d_2) in $\mathcal{E}$ such that $f(y, d_1, d_2) = d$. Without loss of generality, we assume $e_2 = *$, so that $f(x, e_1, e_2) = f_1(e_1)$. Let $d \leq f_1(e_1)$. Since f_1 is a morphism, then there is $d_1 \leq e_1$ such that $f_1(d_1) = d$. Since projections are morphisms, there must be a $(y, d_1, d_2) \leq (x, e_1, e_2)$. I claim that d_2 must be equal to $*$, so that $f(y, d_1, d_2) = f_1(d_1) = d$. If d_2 were not $*$, then its label would be dual to label of d_1. This means that both labels are in $Dis(\Gamma)$, and that no event in $[\![\Gamma]\!]$, and in particular the d, can be labelled by either of them. This contradicts $f_1(d_1) = d$.

Then we have to check that f reflects $\asymp$. So, suppose $f(x, e_1, e_2) \asymp f(x', e'_1, e'_2)$. By the structure of $[\![\Gamma]\!]$ it cannot be that $f(x, e_1, *) \asymp f(x', *, e'_2)$, because they are mapped to disjoint concurrent components. Therefore, w.l.o.g, the only case to consider is $f(x, e_1, *) \asymp f(x', e'_1, *)$. This means $f_1(e_1) \asymp f_1(e'_1)$. Since f_1 is a morphism, then $e_1 \asymp e'_1$, which implies $(x, e_1, *) \asymp (x', e'_1, *)$.

A.6. Proof of Proposition 5.1

By a straightforward case analysis.

A.7. Proof of Theorem 6.1

The proof is by induction on the semantics. All the cases are easily done directly, with the exception of the parallel composition. The case of the parallel composition is a direct consequence of Theorem 4.4.

A.8. Proof of Theorem 6.7

The proof is by induction on the rules of the operational semantics. All cases are rather straightforward, except the parallel composition. For this we need the following lemma. To avoid distinguishing different cases, lets say that, for every event structure $\mathcal{E}$, we have $\mathcal{E} \xrightarrow{*} \mathcal{E}\lfloor * = \mathcal{E}$.

Lemma A.5. *Let $\cong$ denote isomorphism of event structures. We have that $\mathcal{E}_1 \xrightarrow{\alpha} \mathcal{E}_1\lfloor e_1$, and $\mathcal{E}_2 \xrightarrow{\beta} \mathcal{E}_2\lfloor e_2$ if and only if $\mathcal{E}_1 \| \mathcal{E}_2 \xrightarrow{\alpha \bullet \beta} \mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2)$. Moreover, in such a case, we have $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2) \cong (\mathcal{E}_1\lfloor e_1) \| (\mathcal{E}_2\lfloor e_2)$.*

The first part of theorem is straightforward: if e_1, e_2 are minimal in $\mathcal{E}_1, \mathcal{E}_2$, then $(\emptyset, e_1, e_2)$ is a minimal event in $\mathcal{E}_1 \| \mathcal{E}_2$, and vice versa. Assuming this is the case, we are now going to prove that $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2) \cong (\mathcal{E}_1\lfloor e_1) \| (\mathcal{E}_2\lfloor e_2)$. We will define a bijective function $f : \mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2) \to (\mathcal{E}_1\lfloor e_1) \| (\mathcal{E}_2\lfloor e_2)$, such that both f and f^{-1} are morphism of event structure. We define f by induction on the height of the events. Also by induction we show the properties required. That is we prove that

- for every n, f is bijective on elements of height n;
- f preserves and reflects the conflict relation;
- f preserves and reflects the order relation;
- $\Pi_1 \circ f = \Pi_1$ and $\Pi_2 \circ f = \Pi_2$, where Π_1, Π_2 denote the projections in the parallel composition.

In particular. the above properties imply that both f, and f^{-1} are morphisms of event structure. The preservation of the labels follows from the last point, noting that the labels of an event in the product depend only on the labels of the projected events.

Base: height $= 0$
Events of height 0 in $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2)$ are of two forms:

- the form $(\emptyset, d_1, d_2)$, with d_1 minimal in $\mathcal{E}_1$ and d_2 minimal in $\mathcal{E}_2$ (when different from $*$).[1] In such a case we define $f(\emptyset, d_1, d_2) = (\emptyset, d_1, d_2)$.
- the form $((\emptyset, e_1, e_2), d_1, d_2)$, with $e_1 \leq d_1$ and d_2 minimal in $\mathcal{E}_2$, or $e_2 \leq d_2$ and d_1 minimal in $\mathcal{E}_1$, or both $e_1 \leq d_1$, $e_2 \leq d_2$. In such a case we define $f(\emptyset, d_1, d_2) = (\emptyset, d_1, d_2)$.

Note that from the discussion above, it follows that the events $(\emptyset, d_1, d_2)$ and $((\emptyset, e_1, e_2), d_1, d_2)$ cannot be both in $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2)$. We prove that f is well defined on events of height 0. Consider $d = (\emptyset, d_1, d_2)$. Then both d_1, d_2 are minimal in $\mathcal{E}_1, \mathcal{E}_2$ respectively. Also it is not the case that $d_1 \asymp e_1$, nor $d_2 \asymp e_2$, as otherwise we would have $(\emptyset, d_1, d_2) \asymp (\emptyset, e_1, e_2)$. This means that d_1, d_2 belong to $\mathcal{E}_1\lfloor e_1, \mathcal{E}_2\lfloor e_2$ and are minimal there. So that $f(d) = (\emptyset, d_1, d_2) \in (\mathcal{E}_1\lfloor e_1) \| (\mathcal{E}_2\lfloor e_2)$. A similar reasoning applies when $d = ((\emptyset, e_1, e_2), d_1, d_2)$. Now we prove

- f is bijective on events of height 0; it is surjective: take an event $(\emptyset, d_1, d_2)$ in $(\mathcal{E}_1\lfloor e_1) \| (\mathcal{E}_2\lfloor e_2)$. There are several cases. If both d_1 is minimal in $\mathcal{E}_1$ and d_2 is minimal in $\mathcal{E}_2$, and it is not the case that $e_1 \asymp d_1$ nor $e_2 \asymp d_2$, then $(\emptyset, d_1, d_2) \in \mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2)$. Similarly, in the other cases, it is easy to see that $((\emptyset, e_1, e_2), d_1, d_2) \in \mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2)$. Also f is injective. The only thing to check is that $(\emptyset, d_1, d_2)$ and $((\emptyset, e_1, e_2), d_1, d_2)$ cannot be both events in $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2)$, which, as we have observed, is the case.
- f preserves and reflects conflict on events of height 0. This is easily verified by checking all the cases of definition of conflict. Note that it cannot be the case that $(\emptyset, d_1, d_2) \asymp (\emptyset, e_1, e_2)$, as such events do not belong to $\mathcal{E}_1 \| \mathcal{E}_2\lfloor(\emptyset, e_1, e_2)$.
- f preserves and reflects order on events of height 0, trivially.
- $\Pi_1 \circ f = \Pi_1$ and $\Pi_2 \circ f = \Pi_2$, by definition.

[1] We omit this remark in the following: it will be considered implicit throughout.

Step: height $= n + 1$
We assume that f is defined for all events of height $\leq n$, and that it satisfies the required properties there. On events of height $n + 1$, we define f as follows. $f(x, d_1, d_2) = (f(x), d_1, d_2)$. We prove that f is well defined. Note that in order to show that $(f(x), d_1, d_2)$ is an event, we only use properties of $\Pi_1(f(x))$ and $\Pi_2(f(x))$, by induction hypothesis they coincide with $\Pi_1(x)$, $\Pi_2(x)$ respectively. We consider one case, the others being similar. Suppose $d_1 \in E_1$, $d_2 \in E_2$. Then let y be the set of maximal elements of x. Since f preserves and reflects order, we have that $f(y)$ is the set of maximal elements of $f(x)$. Let $y_1 = \Pi_1(y)$, $y_2 = \Pi_2(y)$. Note that we also have $y_1 = \Pi_1(f(y))$, $y_2 = \Pi_2(f(y))$. Since (x, d_1, d_2) is an event, we have

- if $(z, d_1, d_2) \in y$, then either $d_1 \in \mathit{parents}(e_1)$ or$d_2 \in \mathit{parents}(e_2)$;
- for all $d_1 \in \mathit{parents}(e_1)$, there exists $(z, d_1, d_2) \in x$;
- for all $d_2 \in \mathit{parents}(e_2)$ there exists $(z, d_1, d_2) \in x$.
- for no $d_1 \in \Pi_1(x)$, $d_1 \asymp e_1$ and for no $d_2 \in \Pi_2(x)$, $d_2 \asymp e_2$.

These conditions, show that $(f(x), d_1, d_2)$ is also an event.
We now prove that

- f is bijective on the events of height $n + 1$. First, if (x, d_1, d_2) is of height $n + 1$, so is $(f(x), d_1, d_2)$, because by induction hypothesis, f is bijective on events of height n, so that x contains one such event if and only if $f(x)$ does. To prove that f is surjective, consider now an event $(y, d_1, d_2) \in (\mathcal{E}_1 \lfloor e_1) \| (\mathcal{E}_2 \lfloor e_2)$. Since f is biejctive on events of height $\leq n$, we have that there exists x such that $y = f(x)$, and moreover since f preserves and reflects order and conflict, x is a configuration if and only if $f(x)$ is. We have to argue that if $(f(x), d_1, d_2)$ is an event of $(\mathcal{E}_1 \lfloor e_1 \| \mathcal{E}_2 \lfloor e_2)$ then (x, d_1, d_2) is an event of $\mathcal{E}_1 \| \mathcal{E}_2 \lfloor (\emptyset, e_1, e_2)$. This is done in a similar way than the base case. To prove that f is injective, consider (x, d_1, d_2), (x', d_1, d_2), such that $f(x) = f(x')$. By induction hypothesis f is injective, so that $x = x'$ and we are done.
- f preserves and reflects conflict. This is done as in the base case.
- f preserves and reflects order. In fact by definition $d \in x$ if and only if $f(d) \in f(x)$, which is precisely what we need.
- $\Pi_1 \circ f = \Pi_1$ and $\Pi_2 \circ f = \Pi_2$, by definition.

This concludes the proof.

A.9. Proof of Lemma 7.1

Given a NCCS type σ, we define its *erasure* $er(\sigma)$ to be the π type obtained from σ by removing all confidential names. It is a partial function defined as follows

- $er(y_1 : \sigma_1, \ldots, y_n : \sigma_n) = er(\sigma_1), \ldots, er(\sigma_n)$
- $er(\&_{i\in I} \Gamma_i) = (\&_{i\in I} er(\Gamma_i))^{\downarrow}$
- $er(\bigoplus_{i\in I} \Gamma_i) = (\bigoplus_{i\in I} er(\Gamma_i))^{\uparrow}$
- $er(\bigotimes_{i\in I} \Gamma_i) = (er(\Gamma))^{!}$ if for all $i \in I$, $er(\Gamma_i) = er(\Gamma)$.
- $er(\biguplus_{i\in I} \Gamma_i) = (er(\Gamma))^{?}$ if for all $i \in I$, $er(\Gamma_i) = er(\Gamma)$.
- $er(\updownarrow) = \updownarrow$

Lemma A.6. *Suppose $er(\sigma) = \overline{er(\tau)}$, and suppose σ, τ have disjoint sets of names. Suppose for every type of the form $\bigotimes_{k\in K} \Gamma_k$, the set K is* infinite. *Then there is a renaming ρ, such that match$[\tau, \sigma[\rho]] \to S$ and if res$[\tau, \sigma[\rho]] = \bigotimes_{k\in K} \Gamma_k$, then K is infinite.*

By induction on the structure of the types.
We want to prove that for every judgement $P \triangleright \Gamma$, there exists a choice function ρ and an environment Δ, such that $\{\{P \triangleright \Gamma\}\}^{\rho} \triangleright \Delta$. We will prove it by induction on the typing rules. However we need a stronger statement for the induction to go through. We prove that a Δ exists such that it has the following properties

- if $\Delta(x) = \tau$, then $\Gamma(x) = er(\tau)$.
- if $\Gamma(x) = \tau$, then there exists τ' such that $\Delta(x) = \tau'$ and $er(\tau') = \tau$.
- for every type of the form $\bigotimes_{k\in K} \Gamma_k$, the set K is *infinite*.

Finally we prove that if $\{\{P \triangleright \Gamma\}\}^{\rho} \triangleright \Delta$, then for every fresh renaming ρ', $\{\{P \triangleright \Gamma\}\}^{\rho' \circ \rho} \triangleright \Delta[\rho']$.
The proof is trivial for Zero, WeakCl, WeakOut, Res, LIn, LOut, Rout. For Rin, one has just to take care to choose K to be infinite. For the parallel composition, assume $\{\{P_1 \triangleright \Gamma_1\}\}^{\rho_1} \triangleright \Delta_1$ and $\{\{P_2 \triangleright \Gamma_2\}\}^{\rho_2} \triangleright \Delta_2$. First rename all the variables in Δ_1, Δ_2, so that they are disjoint. In this way we can substitute a name of Δ_1 for a name in Δ_2, and Δ_2 would still be well formed.

Then consider a judgement $a : \tau$ in Γ_1 such that there is a matching judgement $a : \sigma$ in Γ_2. Consider the type τ' such that $a : \tau'$ is in Δ_1. Since $er(\tau) = \overline{er(\tau')}$, by Lemma A.6 we find a ρ_a such that $match[\tau, \sigma[\rho_a]] \to S$. For every matching name, we obtain such a renaming. All renamings can be joined to obtain a fresh injective renaming ρ, because no name is involved in two different renamings. Therefore $\Delta_1 \odot \Delta_2[\rho]$ is defined.

A.10. Proof of Theorem 7.2

The proof is by structural induction on $P \triangleright \Gamma$. All the cases are rather easy, taking into account that π-calculus terms can perform any fresh α-variant of an action. For the parallel composition, one has to notice that names that are closed *after* the transition in the π-calculus are closed *before* the transition in NCCS.

A.11. Proof of Theorem 7.3

One direction of the proof (soundness) is easy and it is left to the reader.

To prove full abstraction we define a relation as follows (we omit the environments for simplicity): $(\{\!\{P\}\!\}^{\rho}, \{\!\{Q\}\!\}^{\rho'}) \in \mathcal{R}$ if and only if $P \approx Q$. We want to prove it is a bisimulation. Suppose $\{\!\{P\}\!\}^{\rho} \xrightarrow{\beta} R$. Then $P \xrightarrow{\beta} P'$ and $R = \{\!\{P'\}\!\}^{\rho \setminus obj(\beta)}$. Since $P \approx Q$, then $Q \xrightarrow{\beta} Q'$ with $P' \approx Q'$. Then there exists ρ'' such that $\{\!\{Q\}\!\}^{\rho''} \xrightarrow{\beta} \{\!\{Q'\}\!\}^{\rho'' \setminus obj(\beta)}$. The choice function ρ'' can be obtained from ρ' via a bijection of names ρ''' (note that the cardinality of the Ks is always the same). Then we can write $\{\!\{Q\}\!\}^{\rho'}[\rho'''] \xrightarrow{\beta} \{\!\{Q'\}\!\}^{\rho'' \setminus obj(\beta)}$. We conlcude by noting that $(\{\!\{P'\}\!\}^{\rho \setminus obj(\beta)}, \{\!\{Q'\}\!\}^{\rho'' \setminus obj(\beta)}) \in \mathcal{R}$.

References

[1] Samy Abbes, Albert Benveniste, Branching cells as local states for event structures and nets: Probabilistic applications, in: Proceedings of 8th FoSSaCS, in: LNCS, vol. 3441, Springer, 2005, pp. 95–109.
[2] Paolo Baldan, Andrea Corradini, Ugo Montanari, Unfolding and event structure semantics for graph grammars, in: Proceedings of 2nd FoSSaCS, in: LNCS, vol. 1578, Springer, 1999, pp. 73–89.
[3] Martin Berger, Kohei Honda, Nobuko Yoshida, Sequentiality and the π-calculus, in: Procceedingss of TLCA'01, in: LNCS, vol. 2044, 2001, pp. 29–45.
[4] Gérard Berry, Pierre-Louis Curien, Sequential algorithms on concrete data structures, Theoretical Computer Science 20 (265–321) (1982).
[5] Michele Boreale, Davide Sangiorgi, A fully abstract semantics for causality in the π-calculus, Acta Informatica 35 (5) (1998) 353–400.
[6] Roberto Bruni, Hernán Melgratti, Ugo Montaari, Event structure semantics for nominal calculi, in: Proceedings of 17th CONCUR, in: LNCS, vol. 4137, Springer, 2006, pp. 295–309.
[7] Maria Grazia Buscemi, Vladimiro Sassone, High-level petri nets as type theories in the join calculus, in: Proceedings of 4th FOSSACS, in: LNCS, vol. 2030, Springer, 2001, pp. 104–120.
[8] Nadia Busi, Maurizio Gabbrielli, Gianluigi Zavattaro, Replication vs. recursive definitions in channel based calculi, in: Proceedings of 30th ICALP, in: LNCS, vol. 2719, Springer, 2003, pp. 133–144.
[9] Nadia Busi, Roberto Gorrieri, A petri net semantics for pi-calculus, in: Proceedings of 6th CONCUR, in: Lecture Notes in Computer Science, vol. 962, Springer, 1995, pp. 145–159.
[10] Gian Luca Cattani, Peter Sewell, Models for name-passing processes: Interleaving and causal, in: Proceedings of 15th LICS, 2000, pp. 322–332.
[11] Kostas Chatzikokolakis, Catuscia Palamidessi, A framework to analyze probabilistic protocols and its application to the partial secrets exchange, in: Proceedings of Symposium on Trustworthy Global Computing, in: LNCS, vol. 3705, Springer, 2005.
[12] Silvia Crafa, Daniele Varacca, Nobuko Yoshida, Compositional event structure semantics for the internal pi-calculus, in: Proceedings of 18th CONCUR, in: LNCS, vol. 4703, Springer, 2007, pp. 317–332.
[13] Pierpaolo Degano, Rocco De Nicola, Ugo Montanari, On the consistency of truly concurrent operational and denotational semantics (extended abstract), in: Proceedings of 3rd LICS, 1988, pp. 133–141.
[14] Pierpaolo Degano, Corrado Priami, Non-interleaving semantics for mobile processes, Theoretical Computer Science 216 (1–2) (1999) 237–270.
[15] Jörg Desel, Javier Esparza, Free Choice Petri Nets, Cambridge University Press, 1995.
[16] Joost Engelfriet, A multiset semantics for the pi-calculus with replication, Theoretical Computer Science 153 (1&2) (1996) 65–94.
[17] Claudia Faggian, François Maurel, Ludics nets, a game model of concurrent interaction, in: Proceedings of 20th LICS, 2005, pp. 376–385.
[18] Claudia Faggian, Mauro Piccolo, A graph abstract machine describing event structure composition, in: Proceedings of the workshop GT-VC 06, Electronic Notes on Theoretical Computer Science vol. 175:4 (2007) 21–36.
[19] Claudia Faggian, Mauro Piccolo, Ludics is a model for the finitary linear pi-calculus, in: Proceedings of 8th TLCA 2007, in: LNCS, vol. 4583, Springer, 2007, pp. 148–162.
[20] Pablo Giambiagi, Gerardo Schneider, Frank D. Valencia, On the expressiveness of infinite behavior and name scoping in process calculi, in: Proceedings of 7th FoSSaCS, in: LNCS, vol. 2987, Springer, 2004, pp. 226–240.
[21] Mihaela Herescu, Catuscia Palamidessi, Probabilistic asynchronous π-calculus, in: Proceedings of 3rd FoSSaCS, in: LNCS, vol. 1784, Springer, 2000, pp. 146–160.
[22] Kohei Honda, Olivier Laurent, An exact correspondence between a typed pi-calculus and polarised proof-nets, Theoretical Computer Science (2010), in press (doi:10.1016/j.tcs.2010.01.028).
[23] Kohei Honda, Nobuko Yoshida, On reduction-based process semantics, Theoretical Computer Science 151 (2) (1995) 385–435.
[24] Lalita Jategaonkar Jagadeesan, Radha Jagadeesan, Causality and true concurrency: A data-flow analysis of the pi-calculus (extended abstract), in: Proceedings of 4th AMAST, in: LNCS, vol. 936, Springer, 1995, pp. 277–291.
[25] Gilles Kahn, Gordon D. Plotkin, Concrete domains, Theoretical Computer Science 121 (1–2) (1993) 187–277.
[26] Antoni Mazurkiewicz, Trace theory, in: Petri Nets: Applications and Relationships to Other Models of Concurrency, in: LNCS, vol. 255, Springer, 1986, pp. 279–324.
[27] Paul-André Melliès, Asynchronous games 4: A fully complete model of propositional linear logic, in: Proceedings of 20th LICS, 2005, pp. 386–395.
[28] Robin Milner, Communication and Concurrency, Prentice Hall, 1989.
[29] Robin Milner, Communicating and Mobile Systems: The Pi Calculus, Cambridge University Press, 1999.
[30] Ugo Montanari, Marco Pistore, Concurrent semantics for the π-calculus, Electronic Notes on Theoretical Computer Science 1 (1995).
[31] Mogens Nielsen, Gordon D. Plotkin, Glynn Winskel, Petri nets, event structures and domains, part I, Theoretical Computer Science 13 (1) (1981) 85–108.
[32] Grzegorz Rozenberg, Joost Engelfriet, Elementary net systems, in: Dagstuhl Lecturs on Petri Nets, in: LNCS, vol. 1491, Springer, 1996, pp. 12–121.

[33] Grzegorz Rozenberg, P.S. Thiagarajan, Petri nets: Basic notions, structure, behaviour, in: Current Trends in Concurrency, in: LNCS, vol. 224, Springer, 1986, pp. 585–668.
[34] Davide Sangiorgi, π-calculus, internal mobility and agent passing calculi, Theoretical Computer Science 167 (2) (1996) 235–271.
[35] Daniele Varacca, Hagen Völzer, Glynn Winskel, Probabilistic event structures and domains, in: Proceedings of 15th CONCUR, in: LNCS, vol. 3170, Springer, 2004, pp. 481–496. (A full version appeared in Theoretical Computer Science, 358(2–3):173–199, 2006).
[36] Daniele Varacca, Hagen Völzer, Glynn Winskel, Probabilistic event structures and domains, Theoretical Computer Science 358 (2–3) (2006) 173–199. (Full version of a paper published in CONCUR 2004).
[37] Daniele Varacca, Nobuko Yoshida, Typed event structures and the π-calculus, in: Proceedings of XXII MFPS, ENTCS, 2006.
[38] Daniele Varacca, Nobuko Yoshida, Probabilistic pi-calculus and event structures, in: Proceedings of the Fifth Workshop on Quantitative Aspects of Programming Languages, QAPL 2007, Electronic Notes on Theoretical Computer Science vol. 190:3 (2007) 147–166.
[39] Vasco Vasconcelos, Typed concurrent objects, in: Proc. ECOOP'94, in: Lecture Notes in Computer Science, vol. 821, Springer, 1994, pp. 100–117.
[40] Glynn Winskel, Events in Computation. Ph.D. Thesis, Dept. of Computer Science, University of Edinburgh, 1980.
[41] Glynn Winskel, Event structure semantics for CCS and related languages, in: Proceedings of 9th ICALP, in: LNCS, vol. 140, Springer, 1982, pp. 561–576.
[42] Glynn Winskel, Event structures, in: Advances in Petri Nets 1986 Part II; Proceedings of an Advanced Course, in: LNCS, vol. 255, Springer, 1987, pp. 325–392.
[43] Glynn Winskel, Name generation and linearity, in: Proceedings of 20th LICS, IEEE Computer Society, 2005, pp. 301–310.
[44] Glynn Winskel, Relations in concurrency, in: Proceedings of 20th LICS, IEEE Computer Society, 2005, pp. 2–11.
[45] Glynn Winskel, Mogens Nielsen, Models for concurrency, in: Handbook of Logic in Computer Science, vol. 4, Clarendon Press, 1995.
[46] Nobuko Yoshida, Type-based liveness guarantee in the presence of nontermination and nondeterminism. Technical Report 2002-20, MCS Technical Report, University of Leicester, 2002.
[47] Nobuko Yoshida, Martin Berger, Kohei Honda, Strong Normalisation in the π-Calculus, in: Proceedings of LICS'01, IEEE, 2001, pp. 311–322. (The full version in Journal of Inf. & Comp., 191 (2004) 145–202, Elsevier).