An Operational Petri Net Semantics for the Join-Calculus

Stephan Mennicke
Institute for Programming and Reactive Systems
TU Braunschweig, Germany
mennicke@ips.cs.tu-bs.de

We present a concurrent operational Petri net semantics for the join-calculus, a process calculus for specifying concurrent and distributed systems. There often is a gap between system specifications and the actual implementations caused by synchrony assumptions on the specification side and asynchronously interacting components in implementations. The join-calculus is promising to reduce this gap by providing an abstract specification language which is asynchronously distributable. Classical process semantics establish an implicit order of actually independent actions, by means of an interleaving. So does the semantics of the join-calculus. To capture such independent actions, step-based semantics, e. g., as defined on Petri nets, are employed. Our Petri net semantics for the join-calculus induces step-behavior in a natural way. We prove our semantics behaviorally equivalent to the original join-calculus semantics by means of a bisimulation. We discuss how join specific assumptions influence an existing notion of distributability based on Petri nets.

1 Introduction

Specifications for distributed systems usually employ synchrony assumptions to keep the modeling as simple as possible. Properties of specifications cannot be reused for real implementations, because components in a distributed system run concurrently and communicate in an asynchronous fashion. This leaves a gap between specifications and implementations.

Process calculi, e. g., the π-calculus, concentrate on the essential parts in system specifications, keeping in mind that they represent actual systems. Therefore, they come with a syntax and a semantics to describe the behavior of a system as precise as possible. The asynchronous π-calculus, a restricted π-calculus, tries to reduce the gap between system specifications and implementations. By the asynchronous π-calculus, we are able describe asynchronously communicating systems, but implementations still rely on hard to implement constructs, such as *rendezvous* or *leader election* [16].

The join-calculus by Fournet and Gonthier [9] is a process calculus equipped with a basic language and an abstract notion of computation, the *reflexive chemical abstract machine*. Fournet and Gonthier extend Berry and Boudol's *chemical abstract machine* [2] by explicit reaction sites – similar to *locations* in distributed systems – and combine the concepts of restriction, reception and recursion in one construct called a *join definition*. By join definitions, they force receptors, i. e., names which are used to receive messages, to reside on one location. In contrast, π-calculus allows the use of sent names as receptors (cf.*scope extrusion*) which enables the calculus to describe the concept of *mobility*, but makes distributed implementations of the calculus difficult.

Still, as many other process calculi, the join-calculus only comes with an interleaving semantics which makes it hard to reason about the distributed behavior of processes. Although the join-calculus is equipped with a parallel composition operator, it is rather difficult to describe independence of actions, whereas other models, such as Petri nets [19], describe independence explicitly. Therefore, we present an operational Petri net semantics for the join-calculus taking advantage of the parallel structure to obtain a large degree of independence, i. e., concurrency.

B. Luttik and M.A. Reniers (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2012)
EPTCS 89, 2012, pp. 131–147, doi:10.4204/EPTCS.89.10

The general idea of our Petri net semantics is inspired by the work of Busi and Gorrieri [6], where they propose a Petri net semantics with inhibitor arcs for the π-calculus. They decompose a π term into places and construct the nets by transition rules working on decompositions. They solve scoping issues in the π-calculus by a global renaming. Our semantics does not rely on such a renaming as we store the message scopes in places. As in Busi and Gorrieri's semantics, all necessary information is encoded in the initial decomposition corresponding to initially marked places. We concentrate on the core join-calculus which is not equipped with an explicit choice. Therefore, we can also abandon inhibitor arcs from our semantics. In general, our semantics yields infinite but 1-safe Petri nets. It also comes with a bisimulation result to the original join-calculus semantics ensuring the correctness of our approach.

Petri nets and Petri net related formalisms have already been used to describe the semantics of the join-calculus. Buscemi and Sassone propose a type-theoretic approach by suggesting a hierarchy on the syntax of the join-calculus [5]. For each level, they prove that, if a join-calculus term is typable, i. e., is satisfying a restriction on the syntax, then the Petri net of the join term they construct is bisimilar to the original join-calculus semantics. They get place/transition nets by restricting processes to top-level join definitions. To handle more expressive join terms, they use colored, reconfigurable and dynamic Petri nets. In our work, we cover full expressiveness of the join-calculus by an infinite construction. Bruni et al. propose an event structure semantics for the join-calculus [4]. Their main goal is to establish so called *persistent graph grammars* as a tool to describe name passing process calculi. They focus on an encoding from the asynchronous π-calculus into persistent graph grammars. The unfolding of the grammars yields event structures. For the join-calculus they yield event structures with empty concurrency relations. The semantics we propose includes concurrency by exploiting the parallel structure of a join term. There are also more general approaches which do not give a semantics for the join-calculus, but use the same ideas to obtain new Petri net classes. Prominent examples are *mobile and dynamic Petri nets* by Asperti and Busi [1] and *functional nets* by Odersky [18]. Our approach does not aim at extending Petri nets or introducing new extensions to Petri net theory.

Unfortunately, our net semantics yields infinite nets which seems to make it impossible to be useful for any real-world applications. Due to nice structural properties of the nets, the semantics could be directly used for any *unfolding based* techniques on Petri nets. One of such applications is model-checking. In Petri net unfoldings [7], it is not necessary to compute the potentially infinite structure of the net, but make use of a finite representation called *prefix*. In this paper, we want to investigate the join-calculus in terms of distributability. Recent research [13, 20] suggest a notion of distributed systems in terms of Petri nets and proved a Petri net structure, which refers to symmetric confusion, to be impossible to distribute. If our proposed semantics is reasonable and correct, we may argue on the distributability of the calculus itself.

The rest of the paper is structured as follows. Sect. 2 introduces the necessary notions for this paper including Petri nets (Sect. 2.1) and an overview of the join-calculus (Sect. 2.2). The following section is concerned with the definition of our Petri net semantics for the join-calculus and its correctness results. In Sect. 4, we discuss a notion of distributability and how the join-calculus influences it. In Sect. 5, we conclude our work and give some further research directions.

2 Preliminaries

In this section, we introduce the basic notions and concepts used in our net semantics. First, we need the notion of multisets.

Definition 1 (Multisets)**.** Let A be a set. A *multiset M over A* is a mapping from A to $\mathbb{N}$. For $a \in A$,

$M(a) = 0$ iff $a \notin M$. Otherwise $a \in M$. Two multisets M_1, M_2 over A can be unified by $\uplus$. $M_1 \uplus M_2$ is a multiset where for each $a \in A$, $(M_1 \uplus M_2)(a) = M_1(a) + M_2(a)$.

Whenever f is a function from a set A to a cartesian product $\prod_{i=0}^{n} A_i$, then we define the projections on the result of f by $f^i := \pi_i \circ f$, where π_i is the projection function on the ith component of the product. id denotes the *identity function* defined on any set.

In our semantics we need to store scopes for objects. These scopes may be nested. To handle this nesting of scope we introduce the notion of *stacks* – a common data structure also used in compilers. A stack may be empty ($\bot$) or filled with elements of an alphabet. It is equipped with three operations. First, the *push* operation adds an element on top of a stack. Second, the *top* operation returns the top element of a stack. Last, the *pop* operation removes the top element of a stack.

Definition 2 (Stack)**.** Let Σ be an alphabet. A *stack s over Σ* is either $\bot$ or s contains at least one element $e \in \Sigma$, i. e.,, $s = [e, s']$, where s' is a stack over Σ. The set of all stacks over Σ is denoted by $\mathscr{S}_\Sigma$. The following operations are defined on $\mathscr{S}_\Sigma$.

- $\top : \mathscr{S}_\Sigma \to \Sigma$ denotes the top element of a stack s with

$$s\top := \begin{cases} \varepsilon & s = \bot \\ e & s = [e, s']. \end{cases}$$

- $\downarrow : \mathscr{S}_\Sigma \times \Sigma \to \mathscr{S}_\Sigma$ denotes the push operation. For a stack s and a symbol e, $s \downarrow e := [e, s]$.
- $\uparrow : \mathscr{S}_\Sigma \to \mathscr{S}_\Sigma$ denotes the pop operation. For a stack s,

$$s\uparrow = \begin{cases} \bot & s = \bot \\ s' & s = [e, s']. \end{cases}$$

Instead of $[e_1, [e_2, [\ldots, [e_n, \bot]\ldots]]$ we write $[e_1, e_2, \ldots, e_n, \bot]$.

Labeled transition systems serve as the common semantic model of both formalisms, Petri nets and the join-calculus. It consists of three components, a set of states Q, a labeled relation between states $\to$ and a start state q_0. The labels for so called transitions are obtained from some alphabet Σ.

Definition 3 (LTS)**.** A *labeled transition system (over Σ)*, LTS is a triple, $(Q, \to, q_0)$ where Q is a set, $\to \subseteq Q \times \Sigma \times Q$, and $q_0 \in Q$.

In [11], van Glabbeek gives a huge collection of behavioral equivalences for LTSs. *Bisimulation* is a very strong equivalence taking the branching structure, i. e., the structure of decisions, of a system into account. As already mentioned, Petri nets as well as the join-calculus have an LTS semantics. Therefore, we introduce the notion of bisimulation. Later in Sect. 3.3 we will prove our semantics introduced in Sect. 3.1 to be *bisimilar* to the original semantics of the join-calculus.

Definition 4 (Bisimulation)**.** Let $A_1 = (Q_1, \to_1, q_1)$ and $A_2 = (Q_2, \to_2, q_2)$ be labeled transition systems over some alphabet Σ. A relation $\mathscr{R} \subseteq Q_1 \times Q_2$ is called a *bisimulation between A_1 and A_2* iff

- $(q_1, q_2) \in \mathscr{R}$,
- if $(p, q) \in \mathscr{R}$ and $p \xrightarrow{a}_1 p'$, then there exists $q' \in Q_2$ such that $q \xrightarrow{a}_2 q'$ and $(p', q') \in \mathscr{R}$, and
- if $(p, q) \in \mathscr{R}$ and $q \xrightarrow{a}_2 q'$, then there exists $p' \in Q_1$ such that $p \xrightarrow{a}_1 p'$ and $(p', q') \in \mathscr{R}$.

If such a relation exists, then A_1 and A_2 are *bisimilar*.

2.1 Petri Nets

Petri nets were first introduced by Carl Adam Petri [19]. Petri nets are directed bipartite graphs with places drawn as circles and transitions drawn as boxes. Places and transitions are the nodes of a net. Directed edges called arcs, either connect places with transitions or transitions with places. An example is depicted in Fig. 5. We assume a universe of places denoted by $\mathscr{P}$. We later specify $\mathscr{P}$ to meet the purposes of our semantics. The set of net places is a subset of $\mathscr{P}$. As in labeled transition systems we have a fixed alphabet Σ for transition labels representing the actions of a system. In contrast to classical net definitions, we directly encode the set of arcs into transitions.

Definition 5 (Net)**.** The tuple $N = (P,T)$ is called a *labeled net over* Σ iff

- $P \subseteq \mathscr{P}$ is a set and
- $T \subseteq 2^P \times \Sigma \times 2^P$.

The label $\pi_2(t)$ of a transition t is also referred to as $l(t)$. Here, l is implicitly given and not a part of the net definition. The preset of a transition t is denoted by ${}^\bullet t := \pi_1(t)$, the postset of t is denoted by $t^\bullet := \pi_3(t)$. Pre- and postsets of places are defined by ${}^\bullet p := \{t \in T \mid p \in t^\bullet\}$ and $p^\bullet := \{t \in T \mid p \in {}^\bullet t\}$. The arc relation is obtained by $F = \{(p,t) \in P \times T \mid p \in {}^\bullet t\} \cup \{(t,p) \in T \times P \mid p \in t^\bullet\}$.

A net is called *finite* iff $(P \cup T)$ is finite. Otherwise, the net is called *infinite*.

The potential state of nets is described by *markings*, which are multisets over the set of places. Tokens, drawn as black dots (cf. Fig. 5), represent the number of places in a marking. These states may change by *firing* transitions. Transitions are *enabled* iff there is at least one token on any *input place* $p \in {}^\bullet t$. An enabled transition may fire, which means that it *consumes* one token from each input place and *produces* one token on any output place $p \in t^\bullet$. This procedure is formally defined by the *firing rule*.

Definition 6 (Enabledness, Firing rule)**.** Let $N = (P,T)$ be a net and let $m : P \to \mathbb{N}$ be a marking of N. A transition $t \in T$ is *enabled under* m, written $m[t\rangle$, iff $m(p) > 0$ for all $p \in {}^\bullet t$. An enabled transition $t \in T$ may *fire*. The *successor marking of* m by firing t is m', written $m[t\rangle m'$, with

$$m'(p) = \begin{cases} m(p) + 1 & \text{if } p \in (t^\bullet \setminus {}^\bullet t) \\ m(p) - 1 & \text{if } p \in ({}^\bullet t \setminus t^\bullet) \\ m(p) & \text{else.} \end{cases}$$

Petri nets are nets with an initial marking m_0 corresponding to the start state of a net.

Definition 7 (Petri net)**.** The triple $N = (P,T,m_0)$ is called a *Petri net* iff (P,T) is a net and m_0 is a marking of (P,T).

A marking m is *reachable* in a net $N = (P,T,m_0)$ iff there exists a sequence of transitions $t_1,\ldots,t_n$ ($t_i \in T$) such that $m_0[t_1\rangle \ldots [t_n\rangle m$. The set of all reachable markings of N is denoted by $Reach(N)$. By relating reachable markings we derive an LTS from a Petri net.

Definition 8. Let $N = (P,T,m_0)$ be a Petri net labeled over Σ. The LTS of N is defined by $\mathsf{LTS}(N) := (Reach(N), \longrightarrow, m_0)$ where

- $\longrightarrow \subseteq Reach(N) \times \Sigma \times Reach(N)$ and
- $(m,a,m') \in \longrightarrow$ iff there exists $t \in T$ with $m[t\rangle m'$ and $l(t) = a$.

Instead of $(m,a,m') \in \longrightarrow$ we often use the abbreviation $m \xrightarrow{a} m'$.

$$P \quad ::= \quad 0 \;\Big|\; x\langle v\rangle \;\Big|\; P \,|\, P \;\Big|\; \mathsf{def}\, x\langle u\rangle \,|\, y\langle v\rangle \triangleright P \,\mathsf{in}\, P$$

Figure 1: Syntax of $\mathscr{J}_{core}$

2.2 Join-Calculus

The join-calculus [9] is a process algebra describing the model of the *reflexive chemical abstract machine* based on Berry's and Boudol's chemical abstract machine [2]. One of the reasons for the development of the join-calculus was the difficulty to actually implement distributed CCS or distributed π-calculus. In comparison to the π-calculus by Milner [16], the join-calculus combines restriction, recursion and reception in one construct called *join definition*, forcing receptors to reside on one *location*. Hence, it is not possible to *extrude* a name and use the same name for reception. Fournet and Gonthier [9] identified a strict subset of the join-calculus which is proven to be as expressive as the full calculus. This subset is called *core join-calculus*. This section and our Petri net semantics is based on the core calculus.

For further notions, we assume an infinite set of names $\mathscr{N}$. The syntax of the core join-calculus is defined in Fig. 1. 0 stands for the *null process*, a process with no behavior. $x\langle v\rangle$ represents *output messages*. As in the π-calculus, x stands for the channel name and v is a value passed through x. The *parallel composition* of two processes P and Q is denoted by $P \,|\, Q$, where P and Q work independently. The last syntactic element is the *definition*, $\mathsf{def}\, x\langle u\rangle \,|\, y\langle v\rangle \triangleright Q \,\mathsf{in}\, P$. Definitions combine restriction, reception of names and recursion in one construct. $x\langle u\rangle \,|\, y\langle v\rangle$ is called the *join-pattern*. $x\langle u\rangle \,|\, y\langle v\rangle \triangleright Q$ is called the *join definition* or, together with a process P, the *enclosing definition* of P. We denote the set of all join definitions by $\mathscr{D}$. P is the *enclosed process*. The set of all core join terms is denoted by $\mathscr{J}_{core}$.

Variables in a join-term are partitioned into three categories which are not necessarily disjoint. The *free variables* (fv) are those being visible to the environment. *Defined variables* (dv) are variables bound to a join definition, i. e., those channels that are processed by a definition. *Received variables* (rv) are only locally bound to new processes resulting from the application of join definitions. These three sets are defined in Fig. 2 (cf. [8]).

We use $\sigma_{\mathsf{fv}}, \sigma_{\mathsf{dv}}, \sigma_{\mathsf{rv}}$ to denote a renaming on the set of free, defined and received variables.

The core join-calculus has its roots in an abstract machine called the *reflexive chemical abstract machine*. Instead of specifying a set of reduction rules, the chemical abstract machine first defines a structural congruence and, on top of that, there is only one reduction rule. In process calculi this method is adopted to reduce the number of rules for a structural operational semantics significantly. As we want to use the structural operational semantics to define labeled transition systems of the core join-calculus, we first need the structural congruence of core join terms. The congruence defined in Fig. 3 is reduced to the core join-calculus (cf. [8]).

From the structural congruence we observe that it does not matter what the exact defined variables are. In consequence, we may rename them. We thereby need to make sure that all occurrences of defined variables in the enclosed process are renamed as well. Later, our semantics will keep track of definitions. To make sure that there are no name clashes, we introduce a minimal notion of *normality* on which we rely. Our normality criterion is concerned with join definitions occurring in parallel, i. e., definitions $D_1, \ldots, D_k$ in processes of the form,

$$\mathsf{def}\, D_1 \,\mathsf{in}\, P_1 \,|\, \ldots \,|\, \mathsf{def}\, D_k \,\mathsf{in}\, P_k.$$

Definition 9 (Normality of $\mathscr{J}_{core}$)**.** We call a process $P \in \mathscr{J}_{core}$ *normal* if for all definitions D, D' occurring in parallel in P, it holds that $\mathsf{dv}(D) \cap \mathsf{dv}(D') = \emptyset$.

$$
\begin{array}{rcl}
\mathsf{fv}[x\langle v_1,\ldots,v_n\rangle] & \stackrel{\mathsf{Def}}{=} & \{x,v_1,\ldots,v_n\} \\
\mathsf{fv}[\mathsf{def}\,D\,\mathsf{in}\,P] & \stackrel{\mathsf{Def}}{=} & (\mathsf{fv}[P]\cup\mathsf{fv}[D])\setminus\mathsf{dv}[D] \\
\mathsf{fv}[P\,|\,P'] & \stackrel{\mathsf{Def}}{=} & \mathsf{fv}[P]\cup\mathsf{fv}[P'] \\
\mathsf{fv}[0] & \stackrel{\mathsf{Def}}{=} & \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{fv}[J\triangleright P] & \stackrel{\mathsf{Def}}{=} & \mathsf{dv}[J]\cup(\mathsf{fv}[P]\setminus\mathsf{rv}[J]) \\
\mathsf{dv}[J\triangleright P] & \stackrel{\mathsf{Def}}{=} & \mathsf{dv}[J]
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{dv}[x\langle y_1,\ldots,y_n\rangle] & \stackrel{\mathsf{Def}}{=} & \{x\} & \mathsf{dv}[J\,|\,J'] & \stackrel{\mathsf{Def}}{=} & \mathsf{dv}[J]\uplus\mathsf{dv}[J'] \\
\mathsf{rv}[x\langle y_1,\ldots,y_n\rangle] & \stackrel{\mathsf{Def}}{=} & \{y_1,\ldots,y_n\} & \mathsf{rv}[J\,|\,J'] & \stackrel{\mathsf{Def}}{=} & \mathsf{rv}[J]\uplus\mathsf{rv}[J']
\end{array}
$$

Figure 2: Free, defined and received variables of $\mathscr{J}_{core}$ terms

$$
\begin{array}{rcll}
P\,|\,0 & \equiv & P & \\
P\,|\,Q & \equiv & Q\,|\,P & \\
(P\,|\,Q)\,|\,R & \equiv & P\,|\,(Q\,|\,R) & \\
P\,|\,\mathsf{def}\,D\,\mathsf{in}\,Q & \equiv & \mathsf{def}\,D\,\mathsf{in}\,P\,|\,Q & \text{if } \mathsf{fv}(P)\cap\mathsf{dv}(D)=\emptyset \\
\mathsf{def}\,D\,\mathsf{in}\,\mathsf{def}\,D'\,\mathsf{in}\,P & \equiv & \mathsf{def}\,D'\,\mathsf{in}\,\mathsf{def}\,D\,\mathsf{in}\,P & \text{if } \mathsf{fv}(D)\cap\mathsf{fv}(D')=\emptyset \\
\\
\mathsf{def}\,D\,\mathsf{in}\,P & \equiv & \mathsf{def}\,D\sigma_{\mathsf{dv}}\,\mathsf{in}\,P\sigma_{\mathsf{dv}} & \text{if } \sigma_{\mathsf{dv}} \text{ injective} \\
\mathsf{def}\,D\,\mathsf{in}\,P & \equiv & \mathsf{def}\,D\sigma_{\mathsf{rv}}\,\mathsf{in}\,P & \text{if } \sigma_{\mathsf{rv}} \text{ injective}
\end{array}
$$

Figure 3: Structural congruence on $\mathscr{J}_{core}$

$$
\text{(JOIN)} \qquad x\langle s\rangle\,|\,y\langle t\rangle \xrightarrow{x\langle u\rangle\,|\,y\langle v\rangle\triangleright R} R[s/u,t/v]
$$

$$
\text{(REACT)} \qquad \frac{P \xrightarrow{D} P'}{\mathsf{def}\,D\,\mathsf{in}\,P \overset{D}{\longmapsto} \mathsf{def}\,D\,\mathsf{in}\,P'}
$$

$$
\text{(PAR1)}\ \frac{P \xrightarrow{D} P'}{P\,|\,Q \xrightarrow{D} P'\,|\,Q} \qquad \text{(PAR2)}\ \frac{P \overset{D}{\longmapsto} P'}{P\,|\,Q \overset{D}{\longmapsto} P'\,|\,Q}
$$

$$
\text{(JUMP1)}\ \frac{P \xrightarrow{D} P', \mathsf{dv}(D)\cap\mathsf{fv}(D')=\emptyset}{\mathsf{def}\,D'\,\mathsf{in}\,P \xrightarrow{D} \mathsf{def}\,D'\,\mathsf{in}\,P'}
$$

$$
\text{(JUMP2)}\ \frac{P \overset{D}{\longmapsto} P'}{\mathsf{def}\,D'\,\mathsf{in}\,P \overset{D}{\longmapsto} \mathsf{def}\,D'\,\mathsf{in}\,P'}
$$

$$
\text{(STRUCT1)}\ \frac{P \xrightarrow{D} P', P\equiv Q}{Q \xrightarrow{D} Q'} \qquad \text{(STRUCT2)}\ \frac{P \overset{D}{\longmapsto} P', P\equiv Q}{Q \overset{D}{\longmapsto} Q'}
$$

Figure 4: Labeled transition semantics of $\mathscr{J}_{core}$

We define the semantics of core join processes by their labeled transition systems respecting the reduction semantics given by Fournet [8]. In Fig. 4, we extended Fournet's semantics by an extra type of labeled arrows which represent the τ-labeled steps in Fournet's semantics. $\xrightarrow{D}$ describes potential steps over D, while $\overset{D}{\longmapsto}$ describes actual reaction steps. We extended the original semantics to make the LTS of join comparable to the labeled net semantics we propose in Sect. 3.

Definition 10 (LTS of $\mathscr{J}_{core}$)**.** Let $P \in \mathscr{J}_{core}$. The *labeled transition system of P* is

$$\mathsf{LTS}(P) := (\mathscr{J}_{core}, \longmapsto, P)$$

where $\longmapsto \subseteq \mathscr{J}_{core} \times \mathscr{D} \times \mathscr{J}_{core}$ is the smallest relation respecting the structural operational semantics in Fig. 4.

In general, this labeled transition system is infinite and has unreachable parts. The JOIN rule reveals potential reactions. The actual reaction rule, i. e., REACT, introduces the new arrow type. Only if P has a potential D step to P', then the reaction actually takes place. For the remaining rules we have one for the potential arrows and one for the reaction arrow. The PAR rules work as expected. A join definition can be skipped if a reaction has already taken place, i. e., JUMP$_2$, or the potential step D does not interfere with other free variables, i. e., as in JUMP$_2$. The STRUCT rules refer to the structural congruences as defined in Fig. 3. For a better understanding of the labeled transition semantics we give two examples.

Example 1. Consider the process $P = \mathsf{def}\, x\langle u\rangle \,|\, y\langle v\rangle \triangleright u\langle v\rangle \,\mathsf{in}\, x\langle k\rangle \,|\, x\langle j\rangle \,|\, y\langle 2\rangle$. For simplicity, we use the definition variable $D = x\langle u\rangle \,|\, y\langle v\rangle \triangleright u\langle v\rangle$. Intuitively, P has two possible executions. First, $x\langle k\rangle$ and $y\langle 2\rangle$ react with D or second, $x\langle j\rangle$ and $y\langle 2\rangle$ react under D. In both cases, one message $x\langle \_\rangle$ remains in the process. As $x\langle j\rangle \,|\, y\langle 2\rangle$ potentially react with D, the rule JOIN tells that $x\langle j\rangle \,|\, y\langle 2\rangle \xrightarrow{D} j\langle 2\rangle$. Now, REACT can be directly applied, i. e., $\mathsf{def}\, D \,\mathsf{in}\, x\langle k\rangle \,|\, x\langle j\rangle \,|\, y\langle 2\rangle \overset{D}{\longmapsto} \mathsf{def}\, D \,\mathsf{in}\, x\langle k\rangle \,|\, j\langle 2\rangle$. From there on, there is no other step possible. The second execution can be obtained by the use of STRUCT1. We needed $x\langle k\rangle$ and $y\langle 2\rangle$ in parallel. Due to commutativity and associativity of the parallel operator, this is possible. Therefore, by STRUCT1 we obtain $x\langle k\rangle \,|\, x\langle j\rangle \,|\, y\langle 2\rangle \xrightarrow{D} k\langle 2\rangle \,|\, x\langle j\rangle$. Again, we may apply REACT to get the actual reaction, i. e., $\mathsf{def}\, x\langle u\rangle \,|\, y\langle v\rangle \triangleright u\langle v\rangle \,\mathsf{in}\, x\langle k\rangle \,|\, x\langle j\rangle \,|\, y\langle 2\rangle \overset{D}{\longmapsto} \mathsf{def}\, x\langle u\rangle \,|\, y\langle v\rangle \triangleright u\langle v\rangle \,\mathsf{in}\, k\langle 2\rangle \,|\, x\langle j\rangle$. These are the only $\longmapsto$-steps. So, the LTS of P is a choice between the message $j\langle 2\rangle$ and $k\langle 2\rangle$.

In the last example we already saw how JOIN, REACT and STRUCT are applied. The application of the PAR rules is as expected. The next example considers a process where both JUMP rules are applied.

Example 2. Consider

$$P = \mathsf{def}\, x\langle u\rangle \,|\, y\langle v\rangle \triangleright u\langle v\rangle \,\mathsf{in}\, \mathsf{def}\, a\langle v\rangle \triangleright v\langle\rangle \,\mathsf{in}\, x\langle a\rangle \,|\, y\langle 2\rangle.$$

Again, we abbreviate the definitions occurring in P, i. e., $D_1 = x\langle u\rangle \,|\, y\langle v\rangle \triangleright u\langle v\rangle$ and $D_2 = a\langle v\rangle \triangleright v\langle\rangle$. In a first step, we need to identify the potential steps of P. Considering P, there is only one potential step that matters, namely $x\langle a\rangle \,|\, y\langle 2\rangle \xrightarrow{D_1} a\langle 2\rangle$. With that knowledge we can apply JUMP1, because $\mathsf{dv}(D_1) \cap \mathsf{fv}(D_2) = \emptyset$. This yields the following arrow, $\mathsf{def}\, D_2 \,\mathsf{in}\, x\langle a\rangle \,|\, y\langle 2\rangle \xrightarrow{D_1} \mathsf{def}\, D_2 \,\mathsf{in}\, a\langle 2\rangle$. The REACT rule does the rest, i. e., $\mathsf{def}\, D_1 \,\mathsf{in}\, \mathsf{def}\, D_2 \,\mathsf{in}\, x\langle a\rangle \,|\, y\langle 2\rangle \overset{D_1}{\longmapsto} \mathsf{def}\, D_1 \,\mathsf{in}\, \mathsf{def}\, D_2 \,\mathsf{in}\, a\langle 2\rangle$. We are almost done. The REACT rule exhibits the next arrow, $\mathsf{def}\, D_2 \,\mathsf{in}\, a\langle 2\rangle \overset{D_2}{\longmapsto} 2\langle\rangle$. To transfer this result to the whole process, we apply JUMP2, i. e., $\mathsf{def}\, D_1 \,\mathsf{in}\, \mathsf{def}\, D_2 \,\mathsf{in}\, a\langle 2\rangle \overset{D_2}{\longmapsto} \mathsf{def}\, D_1 \,\mathsf{in}\, \mathsf{def}\, D_2 \,\mathsf{in}\, 2\langle\rangle$.

Note that this example is similar to the one at the beginning of Sect. 3. For discussions on the distributability of the join-calculus in Sect. 4, we need to mention the notion of *locality*. In the join-calculus, receptors must reside on one location, i. e., they cannot be extruded to more than one location.

Therefore, a join definition $J \triangleright P$ can be seen as such a location and hence, a location function is implicitly given in core join. We assume each join definition appearing in a join process, either directly or by reduction, to constitute a location. This is an approximation, because system modelers might summarize several join definitions to one location. To express this freedom, a distributed version of the join-calculus has been developed. The *distributed join-calculus* [10] employs explicit location functions and comes with a fully abstract encoding into the join-calculus. However, we concentrate on the core join-calculus. For later discussions, we rely on the above mentioned assumptions on locality.

3 Petri Net Semantics for Join

The semantics operates in two steps. First, the join term is decomposed into an initial set of places. Each place is equipped with a message term of core join, e. g., $x\langle v\rangle$, and the scopes of x and v, because both names may have their individual scopes. Example 3 shows the need for both scopes.

Example 3.

$$P = \mathsf{def}\ \underbrace{x\langle v\rangle \,|\, y\langle w\rangle \triangleright v\langle w\rangle}_{D_1}\ \mathsf{in}\ \mathsf{def}\ \underbrace{a\langle v\rangle \triangleright 0}_{D_2}\ \mathsf{in}\, x\langle a\rangle \,|\, y\langle 2\rangle.$$

In P, we have six names: $a, 2, x, y, v, w$. While x and y are defined by D_1, a is defined by D_2 and 2 is free. The names v and w are received variables and do not occur in a message. Here $x\langle a\rangle$ has the same scope as x, but a is scoped by D_2. So after a D_1 step, there is a message $a\langle 2\rangle$, which may react in D_2. Therefore, each place is equipped with both, the scope of the sender and the scope of the sent name.

The decomposition yields only places for message terms. Parallel compositions and join definitions are represented in the net structure.

The second step of our semantics consists of applications of a transition rule which makes use of the information stored in places. Given two places representing $x\langle a\rangle, y\langle 2\rangle$ in the example above, our transition rule ensures that there exists a transition, labeled by D_1, consuming from both places and producing to places that correspond to the right side of the reaction rule, i. e., the decomposition of $v\langle w\rangle$, where v is mapped to a and w to 2. The just described decomposition yields a place $a\langle 2\rangle$ which can react in D_2 producing no new messages. The Petri net representation of Example 3 is depicted in Fig. 5.

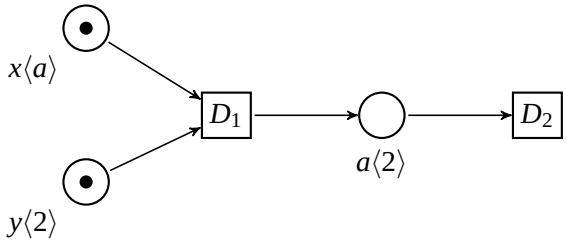

Figure 5: Petri net semantics of P in Example 3

Note that, although we exploit the parallel structure of a process, join definition applications are only unfolded. Therefore, our semantics yields in general infinite net representations.

3.1 Operational Semantics

Our Petri net definitions in Sect. 2.1 left two main points open, which need to be defined in advance. First, the universe of places $\mathscr{P}$ and second, the set of transition labels Σ. As already mentioned in the last section, places are triples. The first component is a join message, e. g., $x\langle v\rangle$. The second and third components are stacks over the set of join definitions $\mathscr{D}$. The first stack represents the scope of the sender name, the second stack that of the sent name.

$$\mathscr{P} := \{x\langle v\rangle \mid x, v \in \mathscr{N}\} \times \mathscr{S}_{\mathscr{D}} \times \mathscr{S}_{\mathscr{D}}$$

denotes the universe of places. Labels for transitions are join definitions, i. e., $\Sigma := \mathscr{D}$. In Fig. 5 we have labeled each place with the message it represents.

The decomposition function returning sets of places for core join terms needs to be equipped with an auxiliary function to manage the name scoping. In the following, such functions are referred to as f or $f_\perp$. f maps names in $\mathscr{N}$ to names in $\mathscr{N}$ and stacks over $\mathscr{D}$, i. e., $f : \mathscr{N} \to (\mathscr{N} \times \mathscr{S}_{\mathscr{D}})$. For $n \in \mathscr{N}$, $f^1(n)$ represents a certain renaming of n (cf. Definition 12). $f^2(n)$ stores the scope of $f^1(n)$. Initially, we use $f_\perp$ with $f_\perp(n) := (n, \perp)$ $(n \in \mathscr{N})$.

During the application of the decomposition, it is necessary to alter the scopes for names. For this purpose, we use a special function g_n operating on any $f : \mathscr{N} \to (\mathscr{N} \times \mathscr{S}_{\mathscr{D}})$. This function shall reduce the stack of n by one element. $g_n(f) : \mathscr{N} \to (\mathscr{N} \times \mathscr{S}_{\mathscr{D}})$ works like f if the parameter is not n. Otherwise, it returns what f returns, but the stack component is reduced by one element, i. e.,

$$(g_n(f))(x) := \begin{cases} (\mathsf{id} \times \uparrow) \circ f(x) & x = n, \\ f(x) & \text{otherwise.} \end{cases}$$

The decomposition function *dec* is defined inductively over the structure of core join processes.

Definition 11. The function $dec : (\mathscr{J}_{core} \times (\mathscr{N} \to (\mathscr{N} \times \mathscr{S}_{\mathscr{D}}))) \to 2^{\mathscr{P}}$ is called *decomposition function*. For all $x, v \in \mathscr{N}$, $P, Q \in \mathscr{J}_{core}$, $D \in \mathscr{D}$, and $f : (\mathscr{N} \to (\mathscr{N} \times \mathscr{S}_{\mathscr{D}}))$ the decomposition is defined by

$$\begin{aligned}
(0, f) &\mapsto \emptyset, \\
(x\langle v\rangle, f) &\mapsto \begin{cases} dec(x\langle v\rangle, g_x(f)) & f^1(x) \notin \mathsf{dv}(f^2(x)\top) \\ dec(x\langle v\rangle, g_v(f)) & f^1(v) \notin \mathsf{dv}(f^2(v)\top) \\ \{(f^1[x\langle v\rangle], f^2(x), f^2(v))\} & \text{otherwise,} \end{cases} \\
(P \mid Q, f) &\mapsto dec(P, f) \uplus dec(Q, f), \\
(\mathsf{def}\, D \,\mathsf{in}\, P, f) &\mapsto dec(P, (\mathsf{id} \times \downarrow D) \circ f).
\end{aligned}$$

Note that the decomposition always yields finite sets of places. The 0 process yields the empty set of places. The result of the decomposition also corresponds to markings. Here, the empty marking represents exactly what we expect from the behavior of 0, i. e., no behavior. The decomposition of the parallel operator is represented by the disjoint union of both components. So, even two equal messages running in parallel are decomposed into two places. Therefore, we use the equality symbol $=$ as *equality up to isomorphism*, when we refer to decompositions or markings of the resulting nets, respectively. In the decomposition of join definitions, we need to adjust the renaming function f, which also handles the scoping of names. A join definition is decomposed as P, but the renaming function is extended by

$(\mathsf{id} \times \downarrow D)$, meaning, that each name now has a new scope, in particular D and all other definitions which were already stored in f.

The decomposition of messages $x\langle v\rangle$ does the main work, because it handles the scopes of x and v. By several applications of g_x and g_v, it assigns the correct scopes to the resulting place. Note that we assume $n \in \mathsf{dv}(\bot)$ for all $n \in \mathscr{N}$.

The recursive application of *dec* eventually terminates, because in each step, the terms in the decomposition get smaller. Either a parallel operator or a join definition is removed. Decompositions of messages also terminate, as the stacks for sender and sent name are reduced by one element as long as they are not empty or the queried name occurs in the set of defined variables. One of the two possibilities holds eventually.

Given a core join process P. The decomposition of P yields the set of initially marked places. The behavior of P is not mapped to the semantics yet. Instead of giving an algorithm to construct a net, we give a rule that must be satisfied by a Petri net to be the semantics of P. To reflect the labeled transition semantics of the core join-calculus, we need to ensure that definitions can be applied, i. e., transitions may fire, if their preconditions are satisfied. Definitions have the form $x\langle u\rangle \,|\, y\langle v\rangle \triangleright Q$, where a process must be able to send messages over x and y to perform the definition, i. e., create a new process Q instantiated with the received variables. As our places carry the necessary scoping, we use that information in Definition 12. A transition consuming from the preconditions of a join definition it represents is forced to produce to places to which another transition does not produce. By this, we reach that places never branch backwards, an important condition discussed later in Sect. 3.2. Furthermore, a transition must not produce to the initially marked places. By this, we obtain an acyclic structure, i. e., bounded places. Indeed, the transition rule and the nature of our decomposition function ensure our Petri net semantics to yield *1-safe Petri nets*.

Definition 12. Let $N = (P, T, m_0)$ be a labeled Petri net over $(\mathscr{P}, \mathscr{D})$. N satisfies the *transition rule* iff for every two places $p, q \in P$ with

- $p = (x\langle a\rangle, s, s_a)$, $q = (y\langle b\rangle, s, s_b)$ and
- $s\top = x\langle u\rangle \,|\, y\langle v\rangle \triangleright R$,

it holds that there exists a transition $t \in T$ with

- $t = (\{p, q\}, x\langle u\rangle \,|\, y\langle v\rangle \triangleright R, P')$,
- $P' \cap m_0 = \emptyset$ and ${}^\bullet P' = \{t\}$

where $P' = dec(R, f_t)$ and $f_t : \mathscr{N} \to (\mathscr{N} \times \mathscr{S}_{\mathscr{D}})$ with for $n \in \mathscr{N}$

$$f_t(n) = \begin{cases} (a, s_a) & n = u, \\ (b, s_b) & n = v, \\ (n, s) & \text{otherwise.} \end{cases}$$

In the transition rule, renamings encoded in f_t become important. As it is possible to have equal names with different scopes, a reaction, i. e., a transition in our nets, needs to respect the scopes although the names are equal. Therefore, we postponed the renaming in the decomposition function to the end of the procedure. Consider Example 4 as an illustration.

Example 4.

$$Q = \mathsf{def}\, a\langle k\rangle \,|\, b\langle k'\rangle \triangleright k\langle\rangle \,|\, k'\langle\rangle \,\mathsf{in}\, b\langle c\rangle \,|\, \mathsf{def}\, c\langle\rangle \triangleright 0 \,\mathsf{in}\, a\langle c\rangle.$$

Q contains two names c with different scopes. The c sent over b is free in Q. The c sent over a is defined. Our construction respects both cs via f_t. Instead of renaming the resulting process, here $k\langle\rangle \,|\, k'\langle\rangle$, to $c\langle\rangle \,|\, c\langle\rangle$ first, we decompose the right side of a join definition and apply the necessary renaming afterward. Therefore, our semantics is able to distinguish both variables c.

Given a core join process J. To construct the Petri net semantics for J, we begin with the set of initially marked places. This set corresponds with the initial decomposition, i. e., $dec(J, f_\perp)$. If there are no applicable definitions in J, the net construction is finished. Otherwise, there must be at least two places violating the just defined transition rule. In order to satisfy the transition rule, we add a transition and a set of places as described in Definition 12. We repeat this procedure until the net satisfies the transition rule. The resulting Petri net represents the semantics of J.

Definition 13. Let $J \in \mathscr{J}_{core}$ be some core-join process. The Petri net $N(J) = (P, T, m_0)$ represents the semantics of J if it is the smallest Petri net satisfying

1. $m_0 = dec(J, f_\perp) \subseteq P$ and
2. the transition rule.

In this section, we have already seen an example (Example 3) and its Petri net semantics in Fig. 5. Note that the procedure described above yields exactly those nets satisfying Definition 13. The criterion asking for the *smallest* net ensures that dead transitions and isolated places are left out.

3.2 Structural Properties

In this section, we investigate the net class of our Petri net semantics, i. e.,*1-safe* Petri nets. This net class restricts all places to contain at most one token for any reachable marking, especially the initial marking. As our decomposition function relies on disjoint unions, initial markings in our nets *are 1-safe*.

In order to show the net class, we prove the following properties, also valid for *occurrence nets* [17].

Proposition 1. *Let $J \in \mathscr{J}_{core}$ be a process. $N(J) = (P, T, m_0)$ satisfies the three criteria below.*

1. *For all $p \in m_0$ it holds that ${}^\bullet p = \emptyset$.*
2. *For all $p \in P$ it holds that $|{}^\bullet p| \leq 1$.*
3. *F^+ (transitive closure of F) is irreflexive.*

The first property states that there are no transitions in the net producing tokens to initially marked places in m_0. The second states that there is always one and only one reason, i. e., a transition, that produces a token to a place. The last one is concerned with cycles in the net structure.

Proof. Let $J \in \mathscr{J}_{core}$ be a process and $N(J) = (P, T, m_0)$ its Petri net semantics.

1. We need to show that for all initially marked places, i. e., $p \in m_0$, it holds that their presets are empty. As $N(J)$ needs to fulfill the transition rule (Definition 12), there is no transition $t \in T$ with ${}^\bullet t \neq \emptyset$ and $t^\bullet \cap m_0 \neq \emptyset$. If there are transitions t with ${}^\bullet t = \emptyset$ producing to m_0, then $N(J)$ is not the smallest net after Definition 13. Therefore, there is no transition producing the m_0 and in consequence, the claim holds.
2. We need to show that for all places $p \in P$, there is at most one transition $t \in {}^\bullet p$. By Definition 12, $N(J)$ needs to satisfy the transition rule. From 1 we know that the claim holds for initially marked places. For any other place p, we need to show that there are no two transition $t, t' \in T$ with $p \in t^\bullet \cap t'^\bullet$. From the transition rule we follow that $P_1 = t^\bullet$ and $P_2 = t'^\bullet$. The transition rule also ensures that ${}^\bullet P_1 = \{t\}$ and ${}^\bullet P_2 = \{t'\}$. If p was in P_1 and in P_2, then $P_1 = P_2$ and in consequence $t = t'$. Therefore, $|{}^\bullet p| \leq 1$.

3. We need to show that there are no cycles in our net representations. By the net construction, we prove that our nets do not introduce cycles. Starting with the set of initial places, the transition rule can only introduce transitions producing to places which are not initially marked. Otherwise, this would contradict 1. Let p be an arbitrary place in the net. From some place in m_0 to p are no cycles in the net. Let Q be the set of all places between m_0 and p. A transition t consuming from p produces to a set of places P'. We need to show that P' is disjoint from Q. Assuming, $P' \cap Q \neq \emptyset$. So, there is a place $q \in Q$ which is also in P'. q cannot be in the set of initially marked places. Therefore, there exists a transition t_q producing to q. Now, ${}^\bullet q = \{t_q, t\}$ which contradicts 2, unless $t \neq t_q$. Therefore, F^+ is irreflexive.

□

Proposition 1 enables us to show that our net semantics produces 1-safe Petri nets. We use the fact that $\max\{m_0(p) \mid p \in P\} \leq 1$, for all $J \in \mathscr{J}_{core}$ with $N(J) = (P, T, m_0)$. Furthermore, we have already proven that there are no cycles in our net semantics and for each place, there is at most one transition producing to it. Therefore, we can formulate the following corollary.

Corollary 1. *Let $J \in \mathscr{J}_{core}$ be a process. Then $N(J)$ is 1-safe.*

The proof follows directly from Proposition 1. For further discussions we introduce the notions of causality, conflict and independence on the basis of Petri nets.

Definition 14. Let $N = (P, T, m_0)$ be a Petri net and $t_1, t_2 \in T$. t_1 and t_2 are said to be in *causal order*, t_1 before t_2, iff there is a reachable marking m_1 with $m_1[t_1\rangle m_2$ and a reachable marking m_3 from m_2 with $m_3[t_2\rangle$ but no such markings which enable t_2 first. t_1 and t_2 are *in direct conflict* iff ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$. Two nodes $n_1, n_2 \in P \cup T$ are *in conflict* iff there exist two transitions $t, t' \in T$ which are in conflict and there exist paths from t to n_1 and from t' to n_2. If $n_1 = n_2$, then n_1 is in *self-conflict*. t_1 and t_2 are *independent* (or *concurrent*) iff they are neither in a causal order nor in conflict.

Intuitively, the notion of independence describes actions, i. e., transitions, which can always occur in parallel. There is a remaining property of occurrence nets which is not satisfied by our nets, namely irreflexivity of the conflict relation. This property states that there are no self-conflicting nodes in the net.

The join-calculus semantics relies on the structural congruences of Fig. 3. Therefore, our net semantics needs to reflect them in a proper way. Indeed, there is a provable correspondence between the structural congruences of the core join-calculus and the Petri net representations. We prove that if two join terms are structurally congruent, then their net representations are isomorphic.

Lemma 1. *Let $P, Q \in \mathscr{J}_{core}$ be processes with $P \equiv Q$. Then $N(P)$ and $N(Q)$ are isomorphic.*

The proof can be found in the technical report to the paper [15]. Lemma 1 also has a side effect to the following behavioral correspondence. We will show a bisimulation between core join terms and their net representations. One of the proof steps is concerned with structurally congruent join terms. As isomorphisms imply bisimulation [11], we can assume it as already proven by Lemma 1.

3.3 Behavioral Properties

In this section, we will prove that the semantics we presented is correct with respect to bisimulation. We already saw LTS interleaving semantics for both, Petri nets and the join-calculus. The states of an LTS for a Petri net is described by markings. States of core join LTS are core join terms. We need to find a bisimulation $\mathscr{R} \subseteq \mathscr{J}_{core} \times 2^{\mathscr{P}}$. Note that any subset of $\mathscr{P}$ describes a valid marking of a Petri net of a core join term.

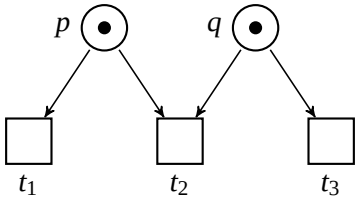

Figure 6: The confusion pattern M

Our bisimulation result relies on the observation, that our decompositions yield valid markings of a net describing the semantics of a core join term. Each state of a process P is represented by its initial decomposition $dec(P, f_\perp)$. When P evolves to P', then our Petri net semantics reflects this behavior by a step from $dec(P, f_\perp)$ to $dec(P', f_\perp)$, because all join definitions of P are preserved by P' and so, they remain on some stack in the decomposition of P'. Conversely, if our net evolves from $dec(P, f_\perp)$ to m, then this m must be equivalent to some $dec(P', f_\perp)$, i. e., there is a step from P to P'. We need to prove that this is actually true for all $P \in \mathscr{J}_{core}$.

Using the just described observation, we formulate a base bisimulation as follows,

$$\mathscr{R} := \{(P, dec(P, f_\perp)) \mid P \in \mathscr{J}_{core}\}.$$

When considering a process P, then we restrict $\mathscr{R}$ to the reachable parts of P, denoted by $R_P := \mathscr{R} \restriction_{P \longmapsto^*}$.

Theorem 1. *Let $P \in \mathscr{J}_{core}$. Then $\mathsf{LTS}(P)$ and $\mathsf{LTS}(N(P))$ are bisimilar.*

The proof can be found in the technical report to this paper [15].

4 Distributability Issues in the Join Calculus

One of the advantages of Petri net semantics for process calculi is the inherent notion of independence. A set of independent actions, i. e., the labels of independent transitions, is called a *step*. A step is enabled if all its transitions are enabled. An enabled step may fire. The resulting marking is the same marking as if all transitions in a step fired in a sequence. Therefore, if we consider an LTS construction in terms of Petri net steps, we do not get more states, but more transitions, because independent actions are summarized in multisets.

The induced steps on the semantics of the core join-calculus correspond to independent join definition applications. Chains of join definitions are are translated into sequences of transitions. Our net semantics also recognizes definition chains which are actually independent, due to the fact that our Petri net semantics respects the structural congruence (cf. Lemma 1).

Steps enable our semantics to argue about the distributability of the join-calculus, or more precisely, about the distributability of our net representations of the join-calculus. We are interested in a particular *confusion* pattern which is depicted in Fig. 6. This structure is called M. The M was introduced by van Glabbeek et al. as a structure which has a major influence to the *distributability* of a system [13, 12]. A net is distributable if there exists a behaviorally equivalent net which is *distributed*. Van Glabbeek et al. call a system distributed if

- it consists of components on different locations,
- the components work concurrently,

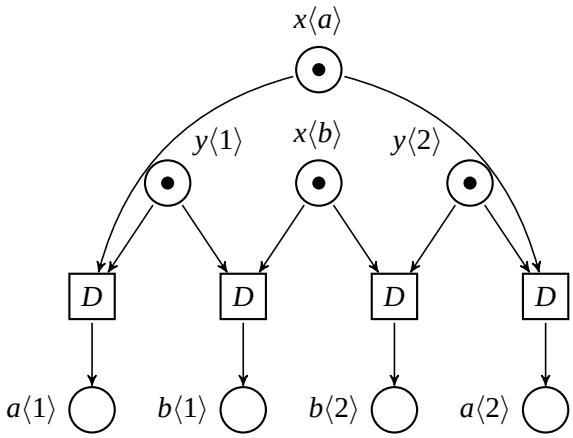

Figure 7: Petri net semantics of P in Example 5.

- the components interact explicitly, and
- communication between components is asynchronous.

They formalized those criteria in a Petri net class called *LSGA nets* – locally sequential, globally asynchronous nets. The crucial point of LSGA nets is that parallel transitions are not allowed to be on one location while transitions sharing input places must share one. The M is not distributed as all transitions need to reside on one location, but t_1 and t_3 may fire in a step, i. e., in parallel. Van Glabbeek et al. proved that if a net contains a fully reachable M, i. e., there is a reachable marking containing at least the places in Fig. 6, then the Petri net is not distributable up to branching-time equivalences [13]. Schicke-Uffmann et al. prove that the M is not distributable in terms of causality respecting equivalences [20]. Their arguments depend on the chosen notion of distributed systems and distributability. However, we consider these notions as reasonable, because the described points above are important phenomena occurring in distributed system design and implementation.

Therefore, if we identify such a structure in our net semantics, there is a potential restriction on the distributability of the join-calculus, i. e., join-calculus processes.

Example 5. Consider the following process,

$$P = \mathsf{def}\ \underbrace{x\langle u\rangle \,|\, y\langle v\rangle \triangleright u\langle v\rangle}_{D}\ \mathsf{in}\, x\langle a\rangle \,|\, y\langle 1\rangle \,|\, x\langle b\rangle \,|\, y\langle 2\rangle .$$

The Petri net semantics $N(P)$ of process P is depicted in Fig. 7. $N(P)$ contains four Ms as depicted in Fig. 6. Initially, the process makes a choice between four different join definition applications. After one application, there is only one possibility for the resulting process to apply the join definition again. Our Petri net semantics reflects this behavior.

The net semantics of the process in Example 5 yields an M just like the one in Fig. 6. It is fully reachable, as the initial marking enables all four Ms. We observe that all transitions are labeled by the same definition. Considering the notion of locality for the join-calculus (cf. Sect. 2.2), this structure remains on one location, although it contains independent transitions. This fact makes a distributability result of the join-calculus incomparable to the results in [13], because van Glabbeek et al. forbid such structures on one location. On the other hand, the implicit location function given by join definitions gives reason to extend the notion of distributability.

In the following, we refer to an M where all transitions are labeled by the same definition as *local*. If all Ms in the join-calculus were local, then the join-calculus would be a distributable process calculus, because our net semantics respects the behavior of the join-calculus and van Glabbeek et al. prove that a Petri net with no fully reachable M is distributable [14]. The following proposition gives proof for this hypothesis.

Proposition 2. *Let $J \in \mathscr{J}_{core}$. If $N(J)$ contains a fully reachable* M*, then it is local.*

Proof. We prove the claim by contradiction. Let $J \in \mathscr{J}_{core}$ be a process and $N(J) = (P,T,m_0)$ be the Petri net semantics of J. Assuming Fig. 6 is a part of $N(J)$ and each transition has a different label, i. e., $l(t_i) \neq l(t_j)$ for $i \neq j$ and $i, j = 1,2,3$. From the transition rule, it follows that all preplaces of a transition have the same stack in their second component. Especially, the top element of these stacks is equal to the label of the transition. Reconsider Fig. 6. As $p \in {}^\bullet t_1$, we know that $p = (\_, s, \_)$ with $s\top = l(t_1)$. $p \in {}^\bullet t_2$, so $p = (\_, s', \_)$ with $s'\top = l(t_2)$. But, by construction, this is not possible if $l(t_1) \neq l(t_2)$. Therefore, either t_1, t_2 do not exist or $l(t_1) = l(t_2)$. The case of t_2, t_3 is analogous, i. e., $l(t_2) = l(t_3)$. By transitivity, we have $l(t_1) = l(t_3)$. □

It is not possible to have an M with different transition labels, i. e., on different locations, in the join-calculus. The proof steps make use of a property of the join-calculus which is reflected by our Petri net semantics. This property is concerned with the assignment of messages to join definitions, i. e., the number of transitions with different labels in the postset of a place. For each join message, there is at most one applicable join definition.

Van Glabbeek et al. [13, 12, 14] and Schicke-Uffmann et al. [20] consider unlabeled nets with no explicit location function to derive their distributability results. If we consider the join-calculus as a distributable process calculus, then it is a natural step to evaluate their results given the assumptions of the join-calculus. Best and Darondeau [3] already consider a given allocation function in their survey paper to argue on the distributability of Petri nets.

5 Conclusion

In this paper we presented an operational Petri net semantics for the join-calculus. We proved that our semantics corresponds to structural congruences and the labeled reduction semantics of the calculus. Furthermore, we investigated issues of distributability in the join-calculus.

In future work, we want to understand how an explicit location function, as implied by the join-calculus, influences the results of [13, 20]. Moreover, we would like to investigate optimizations of the semantics to possibly reach finite net representations of join terms. The mentioned applications in unfolding based techniques is not discussed in this paper. As our suggested semantics has an unfolding nature, it is worthwhile to apply such techniques to the join-calculus by first using our semantics to compute the necessary prefixes of a join term.

Acknowledgments. The author gratefully thanks the DFG (German Research Foundation) for financial support. Moreover, he wishes to thank Malte Lochau and the anonymous reviewers for their useful comments on the paper. Further acknowledgments go to Ursula Goltz, Uwe Nestmann, Kirstin Peters, and Jens-Wolfhard Schicke-Uffmann for valuable discussions.

References

[1] Andrea Asperti & Nadia Busi (2009): *Mobile Petri nets*. *Mathematical Structures in Computer Science* 19(6), pp. 1265–1278, doi:10.1017/S0960129509990193.

[2] Gerard Berry & Gerard Boudol (1990): *The chemical abstract machine*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, ACM, New York, NY, USA, pp. 81–94, doi:10.1145/96709.96717.

[3] Eike Best & Philippe Darondeau (2011): *Petri Net Distributability*. In: *Ershov Memorial Conference*, pp. 1–18, doi:10.1007/978-3-642-29709-0_1.

[4] Roberto Bruni, Hernán Melgratti & Ugo Montanari (2006): *Event Structure Semantics for Nominal Calculi*. In C. Baier & H. Hermanns, editors: *Proceedings of CONCUR 2006, 17th International Conference on Concurrency Theory*, *Lecture Notes in Computer Science* 4137, Springer, pp. 295–309, doi:10.1007/11817949_20.

[5] Maria Grazia Buscemi & Vladimiro Sassone (2001): *High-Level Petri Nets as Type Theories in the Join Calculus*. In: *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '01, Springer-Verlag, London, UK, UK, pp. 104–120, doi:10.1007/3-540-45315-6.

[6] Nadia Busi & Roberto Gorrieri (2009): *Distributed semantics for the pi-calculus based on Petri nets with inhibitor arcs*. *J. Log. Algebr. Program.* 78(3), pp. 138–162, doi:10.1016/j.jlap.2008.08.002.

[7] Javier Esparza & Claus Schröter (2001): *Unfolding Based Algorithms for the Reachability Problem*. *Fundamenta Informaticae* 47(3-4), pp. 231–245.

[8] Cédric Fournet (1998): *The Join-Calculus: a Calculus for Distributed Mobile Programming*. Ph.D. thesis, L'École Polytechnique.

[9] Cédric Fournet & Georges Gonthier (1996): *The reflexive CHAM and the join-calculus*. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 372–385, doi:10.1145/237721.237805.

[10] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget & Didier Rémy (1996): *A calculus of mobile agents*. In Ugo Montanari & Vladimiro Sassone, editors: *CONCUR '96: Concurrency Theory*, *Lecture Notes in Computer Science* 1119, Springer Berlin / Heidelberg, pp. 406–421, doi:10.1007/3-540-61604-7_67.

[11] Rob J. van Glabbeek (2001): *The Linear Time – Branching Time Spectrum I; The Semantics of Concrete, Sequential Processes*. In J.A. Bergstra, A. Ponse & S.A. Smolka, editors: *Handbook of Process Algebra*, chapter 1, Elsevier, pp. 3–99, doi:10.1016/B978-044482830-9/50019-9.

[12] Rob J. van Glabbeek, U. Goltz & J.-W. Schicke (2009): *Symmetric and Asymmetric Asynchronous Interaction*. In: *First Interaction and Concurrency Experiences Workshop (ICE 2008), Satellite Workshop ICALP 2008*, *entcs* Vol. 229, elsevier, pp. 77–95, doi:10.1016/j.entcs.2009.06.040. To appear.

[13] Rob J. van Glabbeek, Ursula Goltz & Jens-Wolfhard Schicke (2008): *On Synchronous and Asynchronous Interaction in Distributed Systems*. In E. Ochmanski & J. Tyszkiewicz, editors: *33rd Intern. Symp. on Mathematical Foundations of Computer Science (MFCS08)*, *Lecture Notes in Computer Science* LNCS 5162, Springer Berlin Heidelberg, pp. 16–35, doi:10.1007/978-3-540-85238-4.

[14] Rob J. van Glabbeek, Ursula Goltz & Jens-Wolfhard Schicke-Uffmann (2012): *On Distributability of Petri Nets - (Extended Abstract)*. In Lars Birkedal, editor: *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, *Lecture Notes in Computer Science* 7213, Springer, pp. 331–345, doi:10.1007/978-3-642-28729-9_22.

[15] Stephan Mennicke (2012): *A Petri Net Semantics for the Join-Calculus*. Technical Report, TU Braunschweig. Available at `https://www.tu-braunschweig.de/Medien-DB/ips/join2petri_techreport.pdf`.

[16] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, I*. *Information and Computation* 100(1), pp. 1 – 40, doi:10.1016/0890-5401(92)90008-4.

[17] Mogens Nielsen, Gordon Plotkin & Glynn Winskel (1979): *Petri nets, event structures and domains*. In Gilles Kahn, editor: *Semantics of Concurrent Computation*, *Lecture Notes in Computer Science* 70, Springer Berlin / Heidelberg, pp. 266–284, doi:10.1007/BFb0022474.

[18] Martin Odersky (2000): *An Introduction to Functional Nets*. In: *Applied Semantics, International Summer School (APPSEM 2000)*, pp. 333–377, doi:10.1007/3-540-45699-6_7.

[19] Carl Adam Petri (1962): *Kommunikation mit Automaten*. Ph.D. thesis, University of Bonn.

[20] Jens-Wolfhard Schicke-Uffmann, Kirstin Peters & Ursula Goltz (2011): *Synchrony vs. Causality in Asynchronous Petri Nets*. In Bas Luttik & Frank Valencia, editors: *Proceedings 18th International Workshop on Expressiveness in Concurrency*, pp. 119–131, doi:10.4204/EPTCS.64.9.