A Petri Net Semantics for the Join-Calculus

(Technical Report 2012-06)

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.

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 $\pi$-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 $\pi$-calculus, a restricted $\pi$-calculus, tries to reduce the gap between system specifications and implementations. By the asynchronous $\pi$-calculus, we are able describe asynchronously communicating systems, but implementations still rely on hard to implement constructs, such as rendezvous or leader election [12].

The join-calculus by Fournet and Gonthier [8] 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, $\pi$-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 [13], describe independence explicitly. Therefore, we present
A Petri Net Semantics for the Join-Calculus

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.

The general idea of our Petri net semantics is inspired by the work of Busi and Gorrieri [5], where they propose a Petri net semantics with inhibitor arcs for the $\pi$-calculus. They decompose a $\pi$ term into places and construct the nets by transition rules working on decompositions. They solve scoping issues in the $\pi$-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 [4]. 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 [3]. 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 $\pi$-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 [14]. 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 [6], it is not necessary to compute the potentially infinite structure of the net, but make use of a finite representation called prefix.

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 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 \( \pi_i \) is the projection function on the \( i \)th component of the product. \( \text{id} \) 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 \( \Sigma \) be an alphabet. A stack \( s \) over \( \Sigma \) is either \( \bot \) or \( s \) contains at least one element \( e \in \Sigma \), i.e., \( s = [e,s'] \), where \( s' \) is a stack over \( \Sigma \). The set of all stacks over \( \Sigma \) is denoted by \( \mathcal{S}_\Sigma \). The following operations are defined on \( \mathcal{S}_\Sigma \).

- \( \top : \mathcal{S}_\Sigma \rightarrow \Sigma \) denotes the top element of a stack \( s \) with
  \[
  s\top := \begin{cases} \epsilon & s = \bot \\ e & s = [e,s'] \end{cases}.
  \]
- \( \downarrow : \mathcal{S}_\Sigma \times \Sigma \rightarrow \mathcal{S}_\Sigma \) denotes the push operation. For a stack \( s \) and a symbol \( e \), \( s \downarrow e := [e,s] \).
- \( \uparrow : \mathcal{S}_\Sigma \rightarrow \mathcal{S}_\Sigma \) 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 \( \rightarrow \) and a start state \( q_0 \). The labels for so-called transitions are obtained from some alphabet \( \Sigma \).

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

In [10], 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 \( A_1 = (Q_1,\rightarrow_1,q_1) \) and \( A_2 = (Q_2,\rightarrow_2,q_2) \) be labeled transition systems over some alphabet \( \Sigma \). A relation \( \mathcal{R} \subseteq Q_1 \times Q_2 \) is called a bisimulation between \( A_1 \) and \( A_2 \) iff

- \( (q_1,q_2) \in \mathcal{R} \).
- if \( (p,q) \in \mathcal{R} \) and \( p \xrightarrow{a_1} p' \), then there exists \( q' \in Q_2 \) such that \( q \xrightarrow{a_2} q' \) and \( (p',q') \in \mathcal{R} \), and
- if \( (p,q) \in \mathcal{R} \) and \( q \xrightarrow{a_2} q' \), then there exists \( p' \in Q_1 \) such that \( p \xrightarrow{a_1} p' \) and \( (p',q') \in \mathcal{R} \).

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 [15]. 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 \( \mathcal{P} \). We later specify \( \mathcal{P} \) to meet the purposes of our semantics. The set of net places is a subset of \( \mathcal{P} \). As in labeled transition systems we have a fixed alphabet \( \Sigma \) 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 \( \Sigma \) iff

- \( P \subseteq \mathcal{P} \) is a set and
- \( T \subseteq 2^P \times \Sigma \times 2^P \).

The label \( \pi_t(t) \) 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 \( ^*t := \pi_1(t) \), the postset of \( t \) is denoted by \( t* := \pi_2(t) \). Pre- and postsets of places are defined by \( ^* p := \{ t \in T \mid p \in t* \} \) and \( p* := \{ t \in T \mid p \in ^*t \} \). The arc relation is obtained by \( F = \{ (p,t) \in P \times T \mid p \in ^*t \} \cup \{ (t,p) \in T \times P \mid p \in t* \} \).

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 \( p \in ^*t \). An enabled transition may fire, which means that it consumes one token from each input place and produces one token on any output place \( p \in t* \). This procedure is formally defined by the firing rule.

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

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

*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 \( m_0[t_1] \ldots [t_n]m \). The set of all reachable markings of \( N \) is denoted by \( \text{Reach}(N) \). 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 \( \Sigma \). The LTS of \( N \) is defined by \( \text{LTS}(N) := (\text{Reach}(N), \rightarrow, m_0) \) where

- \( \rightarrow \subseteq \text{Reach}(N) \times \Sigma \times \text{Reach}(N) \) and
- \((m,a,m') \in \rightarrow \) iff there exists \( t \in T \) with \( m[t]m' \) and \( l(t) = a \).

Instead of \((m,a,m') \in \rightarrow \) we often use the abbreviation \( m \xrightarrow{a} m' \).
2.2 Join-Calculus

The join-calculus [8] 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 [12], 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 [8] 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 \( \mathcal{N} \). The syntax of the core join-calculus is defined in Fig. 1.

For further notions, we assume an infinite set of names \( \mathcal{N} \). The syntax of the core join-calculus is defined in Fig. 1.

\[
P := 0 \mid x\langle v \rangle \mid P \parallel P \mid \text{def } x\langle u \rangle \parallel y\langle v \rangle \triangleright P \text{ in } P
\]

Figure 1: Syntax of \( \mathcal{J}_{\text{core}} \)

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. [7]).

We use \( \sigma_{\text{fv}}, \sigma_{\text{dv}}, \sigma_{\text{rv}} \) 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. [7]).

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

\[
\text{def } D_1 \text{ in } P_1 \parallel \ldots \parallel \text{def } D_k \text{ in } P_k.
\]

**Definition 9** (Normality of \( \mathcal{J}_{\text{core}} \)). We call a process \( P \in \mathcal{J}_{\text{core}} \) normal if for all definitions \( D, D' \) occurring in parallel in \( P \), it holds that \( \text{dv}(D) \cap \text{dv}(D') = \emptyset \).
A Petri Net Semantics for the Join-Calculus

\[
\begin{align*}
\text{fv}[x(v_1, \ldots, v_n)] & \overset{\text{Def}}{=} \{x, v_1, \ldots, v_n\} \\
\text{fv}[\text{def } D \text{ in } P] & \overset{\text{Def}}{=} (\text{fv}[P] \cup \text{fv}[D]) \setminus \text{dv}[D] \\
\text{fv}[P | P'] & \overset{\text{Def}}{=} \text{fv}[P] \cup \text{fv}[P'] \\
\text{fv}[0] & \overset{\text{Def}}{=} \emptyset \\
\text{fv}[J \triangleright P] & \overset{\text{Def}}{=} \text{dv}[J] \cup (\text{fv}[P] \setminus \text{rv}[J]) \\
\text{dv}[J \triangleright P] & \overset{\text{Def}}{=} \text{dv}[J] \\
\text{dv}[x(y_1, \ldots, y_n)] & \overset{\text{Def}}{=} \{x\} \\
\text{dv}[J | J'] & \overset{\text{Def}}{=} \text{dv}[J] \cup \text{dv}[J'] \\
\text{rv}[x(y_1, \ldots, y_n)] & \overset{\text{Def}}{=} \{y_1, \ldots, y_n\} \\
\text{rv}[J | J'] & \overset{\text{Def}}{=} \text{rv}[J] \cup \text{rv}[J']
\end{align*}
\]

Figure 2: Free, defined and received variables of \( \mathcal{J}_{\text{core}} \) terms

\[
\begin{align*}
P \mid 0 & \equiv P \\
P \mid Q & \equiv Q \mid P \\
(P \mid Q) \mid R & \equiv P \mid (Q \mid R) \\
P \mid \text{def } D \text{ in } Q & \equiv \text{def } D \text{ in } P \mid Q \quad \text{if } \text{fv}(P) \cap \text{dv}(D) = \emptyset \\
\text{def } D \text{ in } \text{def } D' \text{ in } P & \equiv \text{def } D' \text{ in } \text{def } D \text{ in } P \quad \text{if } \text{fv}(D) \cap \text{fv}(D') = \emptyset \\
\text{def } D \text{ in } P & \equiv \text{def } D \sigma_{dv} \text{ in } P \sigma_{dv} \quad \text{if } \sigma_{dv} \text{ injective} \\
\text{def } D \text{ in } P & \equiv \text{def } D \sigma_{rv} \text{ in } P \quad \text{if } \sigma_{rv} \text{ injective}
\end{align*}
\]

Figure 3: Structural congruence on \( \mathcal{J}_{\text{core}} \)

\[
\begin{align*}
\text{(JOIN)} & \quad x(s) \mid y(t) \xrightarrow{x(u) \mid y(v) \triangleright R} R[s/u, t/v] \\
\text{(REACT)} & \quad \frac{P \xrightarrow{D} P'}{\text{def } D \text{ in } P \xrightarrow{D} \text{def } D \text{ in } P'} \\
\text{(PAR1)} & \quad \frac{P \xrightarrow{D} P'}{P \mid Q \xrightarrow{D} P' \mid Q} \quad \text{(PAR2)} \quad \frac{P \xrightarrow{D} P'}{P \mid Q \xrightarrow{D} P' \mid Q} \\
\text{(JUMP1)} & \quad \frac{P \xrightarrow{D} P', \text{dv}(D) \cap \text{fv}(D') = \emptyset}{\text{def } D' \text{ in } P \xrightarrow{D} \text{def } D' \text{ in } P'} \\
\text{(JUMP2)} & \quad \frac{P \xrightarrow{D} P'}{\text{def } D' \text{ in } P \xrightarrow{D} \text{def } D' \text{ in } P'} \\
\text{(STRUCT1)} & \quad \frac{P \xrightarrow{D} P', P \equiv Q}{Q \xrightarrow{D} Q'} \quad \text{(STRUCT2)} \quad \frac{P \xrightarrow{D} P', P \equiv Q}{Q \xrightarrow{D} Q'}
\end{align*}
\]

Figure 4: Labeled transition semantics of \( \mathcal{J}_{\text{core}} \)
We define the semantics of core join processes by their labeled transition systems respecting the reduction semantics given by Fournet [7]. In Fig. 4 we extended Fournet’s semantics by an extra type of labeled arrows which represent the $\tau$-labeled steps in Fournet’s semantics. $\xrightarrow{D}$ describes potential steps over $D$, while $\xrightarrow{\cdot}$ 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 $J_{\text{core}}$).** Let $P \in J_{\text{core}}$. The labeled transition system of $P$ is

$$\text{LTS}(P) := (J_{\text{core}}, \xrightarrow{\cdot}, P)$$

where $\xrightarrow{\cdot} \subseteq J_{\text{core}} \times D \times J_{\text{core}}$ 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$_3$. 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 $P = \text{def } x(u) | y(v) \triangleright u(v) \text{ in } x(k) | x(j) | y(2)$. For simplicity, we use the definition variable $D = x(u) | y(v) \triangleright u(v)$. Intuitively, $P$ has two possible executions. First, $x(k)$ and $y(2)$ react with $D$ or second, $x(j)$ and $y(2)$ react under $D$. In both cases, one message $x(\_)$ remains in the process. As $x(j) | y(2)$ potentially react with $D$, the rule JOIN tells us that $x(j) | y(2) \xrightarrow{D} j(2)$. Now, REACT can be directly applied, i.e., $\text{def } D \text{ in } x(k) | x(j) | y(2) \xrightarrow{D} \text{def } D \text{ in } x(k) | j(2)$. From there on, there is no other step possible. The second execution can be obtained by the use of STRUCT1. We needed $x(k)$ and $y(2)$ in parallel. Due to commutativity and associativity of the parallel operator, this is possible. Therefore, by STRUCT1 we obtain $x(k) | x(j) | y(2) \xrightarrow{D} k(2) | x(j)$. Again, we may apply REACT to get the actual reaction, i.e., $\text{def } x(u) | y(v) \triangleright u(v) \text{ in } x(k) | x(j) | y(2) \xrightarrow{D} \text{def } x(u) | y(v) \triangleright u(v) \text{ in } k(2) | x(j)$. These are the only $\xrightarrow{\cdot}$ steps. So, the LTS of $P$ is a choice between the message $j(2)$ and $k(2)$.

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

$$P = \text{def } x(u) | y(v) \triangleright u(v) \text{ in } \text{def } a(v) \triangleright v(\_) \text{ in } x(a) | y(2).$$

Again, we abbreviate the definitions occurring in $P$, i.e., $D_1 = x(u) | y(v) \triangleright u(v)$ and $D_2 = a(v) \triangleright v(\_)$. In a first step, we need to identify the potential steps of $P$. Considering $P$, there is only one potential step that matters, namely $x(a) | y(2) \xrightarrow{D_1} a(2)$. With that knowledge we can apply JUMP 1, because $\text{fv}(D_1) \cap \text{fv}(D_2) = \emptyset$. This yields the following arrow, $\text{def } D_2 \text{ in } x(a) | y(2) \xrightarrow{D_1} \text{def } D_2 \text{ in } a(2)$. The REACT rule does the rest, i.e., $\text{def } D_1 \text{ in } \text{def } D_2 \in x(a) | y(2) \xrightarrow{D_1} \text{def } D_1 \text{ in } \text{def } D_2 \in a(2)$. We are almost done. The REACT rule exhibits the next arrow, $\text{def } D_2 \text{ in } a(2) \xrightarrow{D_2} 2(\_)$. To transfer this result to the whole process, we apply JUMP 2, i.e., $\text{def } D_1 \text{ in } \text{def } D_2 \in a(2) \xrightarrow{D_2} \text{def } D_1 \text{ in } \text{def } D_2 \in 2(\_)$.

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. ??, 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 \([9]\) 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., \( x(v) \), 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.**

\[
P = \text{def } x(v) | y(w) \triangleright v(w) \text{ in def } a(v) \triangleright 0 \text{ in } x(a) | y(2).
\]

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 \( x(a) \) has the same scope as \( x \), but \( a \) is scoped by \( D_2 \). So after a \( D_1 \) step, there is a message \( a(2) \), which may react in \( D_2 \).

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 \( x(a), y(2) \) 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 \( v(w) \), where \( v \) is mapped to \( a \) and \( w \) to 2. The just described decomposition yields a place \( a(2) \) 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](image)

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 $\mathcal{P}$ and second, the set of transition labels $\Sigma$. As already mentioned in the last section, places are triples. The first component is a join message, e.g., $x(v)$. The second and third components are stacks over the set of join definitions $\mathcal{D}$. The first stack represents the scope of the sender name, the second stack that of the sent name.

$$\mathcal{P} := \{x(v) \mid x, v \in \mathcal{N}\} \times \mathcal{I}_\mathcal{D} \times \mathcal{I}_\mathcal{D}$$

denotes the universe of places. Labels for transitions are join definitions, i.e., $\Sigma := \mathcal{D}$. 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 $f$. For an auxiliary function to manage the name scoping. In the following, such functions are referred to as $f$. In the following, such functions are referred to as $f$. Initially, we use $f_\perp$ with $f_\perp(n) := (n, \perp) (n \in \mathcal{N})$.

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 $f : \mathcal{N} \rightarrow (\mathcal{N} \times \mathcal{I}_\mathcal{D})$. This function shall reduce the stack of $n$ by one element. $g_n(f) : \mathcal{N} \rightarrow (\mathcal{N} \times \mathcal{I}_\mathcal{D})$ 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.,

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

The decomposition function $\text{dec}$ is defined inductively over the structure of core join processes.

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

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

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
A Petri Net Semantics for the Join-Calculus

must be able to send messages over. By several applications of \(g_x\) and \(g_y\), it assigns the correct scopes to the resulting place. Note that we assume \(n \in \text{dv}((\bot))\) for all \(n \in \mathcal{N}\).

The recursive application of \(\text{dec}\) 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 \(x(u) \mid y(v) \triangleright Q\), 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 \((\mathcal{P}, \mathcal{D})\). \(N\) satisfies the transition rule iff for every two places \(p, q \in P\) with

- \(p = (x(a), s, s_a), q = (y(b), s, s_b)\) and
- \(s \triangleright x(a) \mid y(v) \triangleright R,\)

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

- \(t = \{p, q\}, x(a) \mid y(v) \triangleright R, P',\)
- \(P' \cap m_0 = \emptyset\) and \(\bullet P' = \{t\}\)

where \(P' = \text{dec}(R, f_t)\) and \(f_t : \mathcal{N} \to (\mathcal{N} \times \mathcal{D})(\bot)\) with for \(n \in \mathcal{N}\)

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

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 3 as an illustration.

**Example 4.**

\[Q = \text{def } a(k) \mid b(k') \triangleright k\langle \rangle \mid k\langle k' \rangle \mid b(c) \triangleright c\langle \rangle \mid 0 \text{ in } a(c).\]

\(Q\) 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 \(k\langle \rangle \mid k\langle k' \rangle\), to \(c\langle \rangle \mid c\langle \rangle\) 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., $\text{dec}(J, f_{\perp})$. 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 $J \in \mathcal{J}_{\text{core}}$ 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. $m_0 = \text{dec}(J, f_{\perp}) \subseteq P$ 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 [13].

**Proposition 1.** Let $J \in \mathcal{J}_{\text{core}}$ 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 $J \in \mathcal{J}_{\text{core}}$ 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 $\bullet t \cap m_0 \neq \emptyset$. 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 [4] 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 $p \in t' \cap t'\bullet$. 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, \(*q = \{t_q, t\} \) which contradicts \[2\] unless \( t \neq t_q \). Therefore, \( F^+ \) is irreflexive.

\[\square\]

Proposition \[1\] enables us to show that our net semantics produces 1-safe Petri nets. We use the fact that \( \max\{m_0(p) | p \in P\} \leq 1 \), for all \( J \in \mathcal{J}_{core} \) 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 \( J \in \mathcal{J}_{core} \) 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 \( m_1[t_1]m_2 \) and a reachable marking \( m_3 \) from \( m_2 \) with \( m_3[t_2] \) but no such markings which enable \( t_2 \) first. \( t_1 \) and \( t_2 \) are in direct conflict iff \( *t_1 \cap *t_2 \neq \emptyset \). 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.** For \( P, Q, R \in \mathcal{J}_{core} \) and \( f : \mathcal{N} \rightarrow (\mathcal{N} \times \mathcal{J}_g) \) it holds that

1. \( \text{dec}(P | 0, f) = \text{dec}(P, f) \),
2. \( \text{dec}(P | Q, f) = \text{dec}(Q | P, f) \), and
3. \( \text{dec}((P | Q) | R, f) = \text{dec}(P | (Q | R), f) \).

**Proof.** We need to show that the resulting sets of places are equal. We use the mathematical properties of the disjoint union operator \( \uplus \).

1. \( \text{dec}(P | 0, f) = \text{dec}(P, f) \uplus \text{dec}(0, f) = \text{dec}(P, f) \uplus \emptyset = \text{dec}(P, f) \)
2. \( \text{dec}(P | Q, f) = \text{dec}(P, f) \uplus \text{dec}(Q, f) = \text{dec}(Q, f) \uplus \text{dec}(P, f) = \text{dec}(Q | P, f) \)
3. \[ \text{dec}(P|Q)|R,f) = \text{dec}(P|Q,f) \cup \text{dec}(R,f) \]
\[ = (\text{dec}(P,f) \cup \text{dec}(Q,f)) \cup \text{dec}(R,f) \]
\[ = \text{dec}(P,f) \cup (\text{dec}(Q,f) \cup \text{dec}(R,f)) \]
\[ = \text{dec}(P,f) \cup \text{dec}(Q|R,f) \]
\[ = \text{dec}(P|(Q|R),f) \]

Lemma 2. Let \( P \in \mathcal{J}_{\text{core}} \) and \( D \in \mathcal{D} \) where \( \text{fv}(P) \cap \text{dv}(D) = \emptyset \). Then for all stacks \( s \) it holds that
\[ \text{dec}(P,f) = \text{dec}(P,(\text{id} \times D) \circ f). \]

Proof. \( D \) would only influence the overall structure of the process if for some free variable \( n \) in \( P \) it holds that \( n \in \text{dv}(D) \). As \( \text{fv}(P) \cap \text{dv}(D) \), it is impossible to have \( n \in \text{dv}(D) \). Therefore, if \( n \) participates in a join message \( m \) – either \( m = n(\cdot) \) or \( m = -n(\cdot) \) – then our decomposition function would decide to remove \( D \) from the stack of \( n \) and call \( \text{dec} \) recursively. Therefore, it is not possible for \( D \) to occur on any name stack in the net.

Lemma 3. Let \( P,Q \in \mathcal{J}_{\text{core}} \) be processes with \( P \equiv Q \). Then \( N(P) \) and \( N(Q) \) are isomorphic.

Proof. We need to show that if \( P \equiv Q \), then \( N(P) \) and \( N(Q) \) are isomorphic, denoted by \( N(P) \cong N(Q) \). We need to distinguish between the cases given by Fig. [3]

1. The cases \( P|0 \equiv P, P|Q \equiv Q|P, \) and \( (P|Q)|R \equiv P|(Q|R) \) we have already proven in Lemma [1] using that whenever \( \text{dec}(P,f_{\bot}) = \text{dec}(Q,f_{\bot}) \), there is a canonical isomorphism between the net components of \( N(P) \) and \( N(Q) \).

2. We have already prepared the case of \( P|\text{def \, D in } Q \equiv \text{def \, D in } P|Q \) in Lemma [2]. The assumption of the structural congruence is that \( \text{fv}(P) \cap \text{dv}(D) = \emptyset \), just as in the lemma. So,
\[ \text{dec}(P|\text{def \, D in } Q,f) = \text{dec}(P,f) \cup \text{dec}(\text{def \, D in } Q,f) \]
\[ = \text{dec}(P,f) \cup \text{dec}(Q,(\text{id} \times D) \circ f) \]
\[ \text{Lemma [2]} \]
\[ = \text{dec}(P,(\text{id} \times D) \circ f) \cup \text{dec}(Q,(\text{id} \times D) \circ f) \]
\[ = \text{dec}(P|Q,(\text{id} \times D) \circ f) \]
\[ = \text{dec}(\text{def \, D in } P|Q,f). \]

In consequence, as the sets of initial places are equal, the same holds for the resulting Petri nets by the canonical isomorphism.

3. In the case of \( \text{def } D_1 \text{ in } \text{def } D_2 \text{ in } P \equiv \text{def } D_2 \text{ in } \text{def } D_1 \text{ in } P \) we use the assumption that \( \text{fv}(D_1) \cap \text{fv}(D_2) = \emptyset \). Any free name in \( P \) remains free, defined by \( D_1 \) or defined by \( D_2 \). The order of \( D_1 \) and \( D_2 \) does not matter as the defined variables of \( D_1 \) and \( D_2 \) are disjoint. Furthermore, as also all free variables of \( D_1 \) and \( D_2 \) are disjoint, neither \( D_1 \) produces messages which can be consumed by \( D_2 \) nor \( D_2 \) produces messages which can be consumed by \( D_1 \). Therefore, although the decompositions yield different set of places, the bindings, i.e., the scope of names, remains the same. An isomorphism can therefore be constructed easily. Equal places are mapped to each other. Places with equal message components are mapped to each other if the scope stacks only differ in the top most components, i.e., either \( D_1 \) or \( D_2 \).
4. In the cases where renaming functions are applied to a term, i.e., \( \text{def } D \in P \equiv \text{def } D \sigma_{dv} \in P \sigma_{dv} \), we construct an isomorphism between the respective net representations. We first note that the structural congruence above can only be applied if \( \sigma_{dv} \) is injective. We directly get a bijective renaming function \( \sigma \), which works like \( \sigma_{dv} \) and for all names \( n \) for which \( \sigma_{dv} \) is not defined, it returns \( n \). \( \sigma \) is an over-approximation. We construct the isomorphism \( \varphi \) between \( N(\text{def } D \in P) \) and \( N(\text{def } D \sigma_{dv} \in P \sigma_{dv}) \) inductively over the depth of the nets.

**Base:** We start with nets of depth 0. These consist of places only, i.e., the initial places (marking respectively) \( P_0 = \text{dec}(P, \bot \downarrow D) \) and \( P'_0 = \text{dec}(P \sigma_{dv}, \bot \downarrow D \sigma_{dv}) \). As we neither add nor remove any places by renaming names of a process, there is an isomorphism between \( P_0, P'_0 \). The sets \( P_0, P'_0 \) are unstructured. The only structure we can rely on in this step, is that of the places. We require our isomorphism to respect the labels of the places, i.e., if \( \sigma(\pi_1(p)) = \pi_1(p') \), then \( \varphi(p) = p' \). Such an isomorphism exists, as \( \sigma \) is bijective and therefore, \( P_0, P'_0 \) are isomorphic.

**Step:** Assuming, for nets of depth \( n \), there exists some isomorphism \( \psi \). We need to show that there is an isomorphism \( \varphi \) for nets of depth \( n + 2 \). Now we take two nets of depth \( n + 2 \) and cut the places of depth \( n + 2 \) and the transitions of depth \( n + 1 \) out. By the induction assumption we have an isomorphism \( \psi \) between these nets. Now some places of depth \( n \) contradict the transition rule. So, we re-apply the transition rule and construct the isomorphism \( \varphi \) as follows. For places and transitions with depth \( \leq n \) we get the same pairs as those in \( \psi \). For transitions \( t \) of depth \( n + 1 \) we pick a place \( p \in \cdot t \) and set \( \varphi(t) = t' \in \psi(p) \cdot t \) with \( \sigma(l(t)) = l(t') \) and there is no other \( t'' \) with \( \sigma(l(t'')) = t' \). Now \( \varphi \) contains all nodes up to depth \( n + 1 \). For places \( p \) of depth \( n + 2 \) we do the same as for transitions. Each place has exactly one transition in its preset. Let this transition be \( t \). We set \( \varphi(p) = p' \in \varphi(t) \cdot t \) if \( \pi_1(\sigma(p)) = \pi_1(p') \) and there is no other \( p'' \) with \( \varphi(p'') = p' \). \( \varphi \) exists and is an isomorphism.

5. The case of \( \text{def } D \in P \equiv \text{def } D \sigma_{dv} \in Q \) is similar to the last case. As received variables do not occur as a name in a message, those names do not influence the overall process.

So, if \( P \equiv Q \), then \( N(P) \) is isomorphic to \( N(Q) \).

Lemma 3 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 [10], we can assume it as already proven by Lemma 3.

### 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 \( R \subseteq J_{\text{core}} \times 2^P \). Note that any subset of \( P \) describes a valid marking of a Petri net of a core join term.

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 \( \text{dec}(P, f_\bot) \). When \( P \) evolves to \( P' \), then our Petri net semantics reflects this behavior by a step from \( \text{dec}(P, f_\bot) \) to \( \text{dec}(P', f_\bot) \), 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 \( \text{dec}(P, f_\bot) \) to \( m \),
then this \( m \) must be equivalent to some \( \text{dec}(P', f_\perp) \), i.e., there is a step from \( P \) to \( P' \). We need to prove that this is actually true for all \( P \in \mathcal{J}_{\text{core}} \).

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

\[
\mathcal{R} := \{(P, \text{dec}(P, f_\perp)) \mid P \in \mathcal{J}_{\text{core}}\}.
\]

When considering a process \( P \), then we restrict \( \mathcal{R} \) to the reachable parts of \( P \), denoted by \( R_P := \mathcal{R} \upharpoonright_{P \rightarrow \ast} \).

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

**Proof.** We use the given relation \( \mathcal{R} \), and prove that \( R_P \) is a bisimulation between \( \text{LTS}(P) \) and \( \text{LTS}(N(P)) \) by induction over the structure of \( P \).

**Base:** We need to distinguish between two base cases here which are equivalent as they do not introduce any behavior.

**Case of** \( P = 0 \): \( (0, \emptyset) \in R_P \) and whenever the process made a step, then the Petri net would be able to mimic it and vice versa, i.e., \( R_P \) is a bisimulation between \( P = 0 \) and its Petri net semantics.

**Case of** \( P = x(v) \): \( (x(v), \text{dec}(x(v), f_\perp)) \in R_P \). The same argument as before. \( R_P \) is a bisimulation between \( P = x(v) \) and its net representation.

**Step:** Let \( R_Q, R_R \) be bisimulations between the processes \( Q, R \) and their net representations \( N(Q), N(R) \).

**Case of** \( P = Q | R \): By definition, \( (Q | R, \text{dec}(Q | R, f_\perp)) \in R_P \). It is sufficient to show, that whenever \( P \rightarrow_D P' \), the Petri net is able to make a step from \( \text{dec}(P, f_\perp) \rightarrow_D \text{dec}(P', f_\perp) \) such that \( (P', \text{dec}(P', f_\perp)) \in R_P \), and vice versa.

We need to show that if \( P \rightarrow_D P' \) in \( \text{LTS}(P) \), then there exists an \( m \) such that \( \text{dec}(P, f_\perp) \rightarrow_D m \) and \( m = \text{dec}(P', f_\perp) \). By the reduction semantics of \( \mathcal{J}_{\text{core}} \), there can only be two cases, either (a) \( Q \rightarrow_D Q' \) and \( P' = Q' | R \) or (b) \( R \rightarrow_D R' \) and \( P' = Q | R' \). By Lemma \( \ref{lemma} \) we only need to consider one of the cases, e.g., case (a). By induction, we know that if \( Q \rightarrow_D Q' \), then \( \text{dec}(Q, f_\perp) \rightarrow_D \text{dec}(Q', f_\perp) \). This means that in \( N(Q) \) there exists a transition \( t \) with \( l(t) = D \) transforming marking \( \text{dec}(Q, f_\perp) \) into \( \text{dec}(Q', f_\perp) \). As \( \text{dec}(P, f_\perp) = \text{dec}(Q, f_\perp) \oplus \text{dec}(R, f_\perp) \), there also exists such a transition \( t \) in \( N(P) \) with \( l(t) = D \). Due to normality of \( P \), \( t \) cannot consume from places of \( \text{dec}(Q, f_\perp) \) and \( \text{dec}(R, f_\perp) \), i.e., there exist only two places in either \( \text{dec}(Q, f_\perp) \) or \( \text{dec}(R, f_\perp) \) such that the transition rule ensures the existence of \( t \). By assumption, \( t \) consumes from places in \( \text{dec}(Q, f_\perp) \), producing the marking \( m \). The firing of \( t \) influences the marking \( \text{dec}(Q, f_\perp) \) only. So, \( m = \text{dec}(Q', f_\perp) \oplus \text{dec}(R, f_\perp) = \text{dec}(P', f_\perp) \). Therefore, \( \text{dec}(P, f_\perp) \rightarrow_D \text{dec}(P', f_\perp) \).

Now we consider the case \( \text{dec}(P, f_\perp) \rightarrow_D \text{dec}(P', f_\perp) \). We need to show that \( P \rightarrow_D P' \). As \( P = Q | R \), we know that \( \text{dec}(P, f_\perp) = m_1 \oplus m_2 \) with \( m_1 = \text{dec}(Q, f_\perp) \) and \( m_2 = \text{dec}(R, f_\perp) \). There are no places \( p \in m_1 \) and \( q \in m_2 \) such that there exists a transition \( t \) consuming from \( p \) and \( q \), due to normality of \( P \). So, if a transition \( t \) is enabled under \( \text{dec}(P, f_\perp) \), it is either enabled by \( m_1 \) or by \( m_2 \). Let \( t \) be enabled under \( m_1 \) with \( l(t) = D \). The successor marking of \( m_1 \) by firing \( t \) is a marking \( m_1' \). By induction, there exists \( Q' \) such that \( Q \rightarrow_D Q' \) and \( (Q', m_1') \in R_Q \), i.e., \( m_1' = \text{dec}(Q', f_\perp) \). Using the reduction semantics of \( \mathcal{J}_{\text{core}} \), we get \( Q | R \rightarrow_D Q' | R \), i.e., \( P \rightarrow_D P' \) where \( P' = Q' | R \). By definition of \( R_P \), we have \( (P', \text{dec}(P', f_\perp)) \in R_P \). The symmetrical case can be omitted, due to commutativity of \( \oplus \).
Case of $P = \text{def } J \triangleright Q \text{in } R$: By definition, $(\text{def } J \triangleright Q \text{in } R, \text{dec}(\text{def } J \triangleright Q \text{in } R, f_\perp)) \in R_P$.

We need to show that if $P \xrightarrow{D} P'$, then there exists a marking $m$, such that $\text{dec}(P, f_\perp) \xrightarrow{D} m$ and $(P', m) \in R_P$. There are two cases to distinguish.

Case $D = J \triangleright Q$: Then $P'$ is a new process, not covered by $R_P$. As $P \xrightarrow{D} P'$, there is a sub-process of $R$ which is equal to $J$ up to a renaming $\sigma_\nu$ of received variables. Let $P' = \text{def } D \text{in } \bar{R} \mid \hat{Q}$. $\hat{Q}$ is a sub-process of an instantiation of $Q$. As messages of $Q$ can be in the scope of a new definition in $R$, we use $\bar{R}$ instead of $R'$ to denote this situation.

Dually, $\bar{Q}$ is reduced by those messages in the scope of $\bar{R}$. By induction hypothesis, there are at least two places, $p, q$, in $\text{dec}(P, f_\perp)$ having the message scope $J \triangleright Q$, formerly free. Due to the transition rule, there must be a transition labeled by $J \triangleright Q$ consuming from $p$ and $q$, and producing to the decomposition of $Q$ with an appropriate renaming. Thereby, the decomposition function ensures an appropriate skipping. The resulting successor marking $m$ is defined as follows,

$$m = (\text{dec}(P, f_\perp) \setminus \{p, q\}) \uplus \text{dec}(Q, f_t).$$

Due to the definition of $f_t$ which handles the scope of messages in $Q$ -- i.e., resulting in $\bar{Q}$ -- and also serves as a renaming function, $m = \text{dec}(P', f_\perp)$. So, $\text{dec}(P, f_\perp) \xrightarrow{D} \text{dec}(P', f_\perp)$.

Case $D \neq J \triangleright Q$: In this case, $D$ is already defined in $R$, i.e., these steps are covered by $R_R$.

It remains to show that if $\text{dec}(P, f_\perp) \xrightarrow{D} m$, then there exists $P'$ such that $P \xrightarrow{D} P'$ and $m = \text{dec}(P', f_\perp)$. Let $t$ be a transition labeled by $D$ enabled under $\text{dec}(P, f_\perp)$ and producing $m$. From the transition rule, we know that there exist two places $p, q$, from which $t$ consumes.

Case $D = J \triangleright Q$: Then $m = (\text{dec}(P, f_\perp) \setminus \{p, q\}) \uplus \text{dec}(Q, f_t)$. So in $P$, there must be two messages $\pi_t(p), \pi_t(q)$ which may react under $J \triangleright Q$, i.e., they fulfill the join pattern $J$ and by induction, no other join pattern in $R$. From the reduction semantics of $\mathcal{J}_{\text{core}}$, we can follow $P \xrightarrow{D} \text{def } J \triangleright Q \text{in } \bar{R} \mid \hat{Q}$, where $\bar{R}$ and $\hat{Q}$ as before. We set $P' = \text{def } J \triangleright Q \text{in } \bar{R} \mid \hat{Q}$. $m$ corresponds to the decomposition $\text{dec}(P', f_\perp)$. By definition it holds that $(P', \text{dec}(P', f_\perp)) \in R_P$.

Case $D \neq J \triangleright Q$: This case is similar to the one already discussed. Again, we use the induction hypothesis, i.e., $R_R$.

\[\square\]

4 Conclusion

In the last section, we presented a Petri net semantics for the join-calculus. Taking a closer look at the resulting nets, we recognize that the nets describe a kind of unfolding of the join-calculus. These unfolding like structures may be used by unfolding based analysis techniques. Fortunately, the infinite structure of the resulting nets is no problem, because in Petri net unfoldings, it is sufficient to compute a finite representation of the infinite structure to derive, for instance, verification results.

We only considered the core join-calculus here as it is proven to be as expressive as full join. We want to conclude with a discussion of both, the expressive power of the core join calculus and that of full join. We are especially interested in the existence of a confusion structure in the resulting Petri net representations, namely the M being depicted in Fig. 6. The M was identified in [11,16] to make asynchronous implementations impossible.
When we think of core join, we find that an \( M \), in its simple form, is unthinkable. Transitions are represented by definitions and places by messages. Now, we needed two independent definitions representing the \( a \) and \( c \) transitions. These definitions had disjoint join patterns. Now there would be a codependent third definition which needed to consume a message shared with the \( a \)-definition and one message shared with the \( c \)-definition. Messages in the core join calculus only belong to one join definition at most. It is therefore impossible that such three definitions may coexist.

We are currently investigating, whether it is possible to create more complex structures that imply the \( M \) or a similar structure. One way is to look at the following term in \( \pi \).

\[
P = \overline{a} + b.0 + \overline{b} + a.0
\]

\( P \) is not implementable in an asynchronous fashion, although it has an asynchronous counter-part as it only uses input-guarded choices. Fournet gave a fully-abstract encoding from asynchronous \( \pi \) to a core join dialect \[7\] which is perfectly compatible with the presented Petri net semantics. We will apply our semantics to the term and investigate the given encoding of Fournet.

Another branch to tackle distributability of the join-calculus is to look how the core calculus differs from the full one. We can extend our semantics in a natural way to the full join calculus and then show that there is an \( M \). First, we discuss in which points we needed to change our semantics. Then, we give a join-\( M \). Last, we translate it into its Petri net representation.

The first extension is that of polyadic messages. Here, our Petri net semantics does not need to be changed as it translates whole messages into places and does not separate channel names from its values or variables. The only difference is that the place labels are getting longer.

The next extension is that we can join more than two messages in a join-pattern. In our semantics, we only need to adjust the transition rule to a join with \( n \) messages, i.e., transitions are consuming from \( n \) places.

The last difference is a crucial one. A definition process does not need to consist of only one join definition \( \triangleright JQ \), but \( n \). The definitions are separated by \( \land \). Here, we need to extend the place labels and the transition rule again. Places not only hold a stack of definitions, but each stack element is a set of possible definitions for a message. Now the transition rule joins such messages that have the same stack through one definition in the set, on top of their stacks.

Now the described join-\( M \) above is not as impossible as in the core join case. We just specify all three definitions in one definition process and add the necessary messages. A join-\( M \) would be a process like the following.

\[
Q = \text{def } a\langle u \rangle \triangleright u\langle \rangle \land a\langle u \rangle \triangleright c\langle v \rangle \triangleright u\langle v \rangle \land c\langle v \rangle \triangleright v\langle \rangle a\langle x \rangle \mid c\langle y \rangle
\]

Let us call \( a\langle u \rangle \triangleright u\langle \rangle \ D_a, a\langle u \rangle \triangleright c\langle v \rangle \triangleright u\langle v \rangle \ D_b, \) and \( c\langle v \rangle \triangleright v\langle \rangle \ D_c. \) \( D_a \) and \( D_c \) can be applied independently while they are mutually excluded by \( D_b. \) The Petri net representation of \( Q \) resembles the Petri net in Fig.\[6\]. It just produces to the places on the left hand-side of the \( \triangleright \)-operators.
There are arguments that this \( M \) is desired, as we do not want to distribute definitions of one definition process\(^1\). Those three definitions shall run on one machine. Nevertheless, it is interesting that there is a fully abstract encoding from full join to core join, but an \( M \) in core join, if it exists, is much harder to find.

References


\(^1\)From a personal talk with Uwe Nestmann.


<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-07</td>
<td>M. Huhn, C. Knieke</td>
<td>Semantic Foundation and Validation of Live Activity Diagrams</td>
</tr>
<tr>
<td>2010-01</td>
<td>A. Litvinenko and H. G. Matthies</td>
<td>Sparse data formats and efficient numerical methods for uncertainties quantification in numerical aerodynamics</td>
</tr>
<tr>
<td>2010-02</td>
<td>D. Grunwald, M. Lochau, E. Börger, U. Goltz</td>
<td>An Abstract State Machine Model for the Generic Java Type System</td>
</tr>
<tr>
<td>2010-03</td>
<td>M. Krosche, R. Niekamp</td>
<td>Low-Rank Approximation in Spectral Stochastic Finite Element Method with Solution Space Adaption</td>
</tr>
<tr>
<td>2011-02</td>
<td>B. V. Rosić, A. Litvinenko, O. Pajonk, H. G. Matthies</td>
<td>Direct Bayesian update of polynomial chaos representations</td>
</tr>
<tr>
<td>2011-04</td>
<td>O. Pajonk, B. V. Rosić, A. Litvinenko, and H. G. Matthies</td>
<td>A Deterministic Filter for non-Gaussian Bayesian Estimation</td>
</tr>
<tr>
<td>2011-05</td>
<td>H. G. Matthies</td>
<td>A Hitchhiker’s Guide to Mathematical Notation and Definitions</td>
</tr>
<tr>
<td>2011-12</td>
<td>S. Oster</td>
<td>A Semantic Preserving Feature Model to CSP Transformation</td>
</tr>
<tr>
<td>2012-01</td>
<td>O. Pajonk, B. V. Rosić and H. G. Matthies</td>
<td>Deterministic Linear Bayesian Updating of State and Model Parameters for a Chaotic Model</td>
</tr>
<tr>
<td>2012-02</td>
<td>B. V. Rosić and H. G. Matthies</td>
<td>Stochastic Plasticity - A Variational Inequality Formulation and Functional Approximation Approach I: The Linear Case</td>
</tr>
<tr>
<td>2012-03</td>
<td>J. Rang</td>
<td>An analysis of the Prothero–Robinson example for constructing new DIRK and ROW methods</td>
</tr>
<tr>
<td>2012-04</td>
<td>S. Kolatzki, M. Hagner, U. Goltz and A. Rausch</td>
<td>A Formal Definition for the Description of Distributed Concurrent Components - Extended Version</td>
</tr>
<tr>
<td>2012-05</td>
<td>M. Espig, W. Hackbusch, A. Litvinenko, H. G. Matthies and P. Wichert</td>
<td>Efficient low-rank approximation of the stochastic Galerkin matrix in tensor formats</td>
</tr>
<tr>
<td>2012-06</td>
<td>S. Mennike</td>
<td>A Petri Net Semantics for the Join-Calculus</td>
</tr>
</tbody>
</table>