Translating Orc Features into Petri nets and the Join Calculus*

Roberto Bruni[1], Hernán Melgratti[2], and Emilio Tuosto[3]

[1] Computer Science Department, University of Pisa, Italy.
[2] IMT, Lucca Institute for Advanced Studies, Italia.
[3] Department of Computer Science, University of Leicester, UK.
bruni@di.unipi.it, hernan.melgratti@imtlucca.it, et52@mcs.le.ac.uk

Abstract. Cook and Misra's Orc is an elegant language for orchestrating distributed services, able to cover e.g. van der Aalst's workflow patterns. We aim to understand the key novel features of Orc by comparing it with variations of Petri nets. The comparison shows that Orc hides powerful mechanisms for name handling (creation and passing) and for atomic distributed termination. Petri nets with static topology can encode Orc under severe restrictions while the full language (up to a more realistic cancellation strategy) can be encoded in Join (that can be seen as a higher-order extension of Petri nets). As an overall result, we promote Join calculus as an elegant language embedding orchestration and computation.

1 Introduction

Service Oriented Computing and its most successful current realisation based on Web Services are challenging science and technology in laying foundations, techniques and engineered development for supporting just-in-time assembly of complex business processes according to the publish-find-bind paradigm. Main issues are concerned with, e.g., security, behavioural description of services with the integration of functional and non-functional requirements, trade-off between network awareness and network transparency, dynamic binding and reconfiguration, model-driven development.

A common theme to all these aspects is service composition. The difference w.r.t. classic program or process composition here is that beside answering the question on "how to compose services", one has to provide languages and logic for "describing composite services" and "use composition as a specification requirement for querying service repositories". Descriptions should be accurate enough to guarantee that dynamically found and bound composite services behave well.

The terms *orchestration* and *choreography* were coined to describe two different flavors of service compositions: orchestration is about describing and executing a single view point model, while choreography is about specifying and guiding a global model. Though the difference between the two terms can be sometimes abused or blurred, substantially orchestration has a more centralised flavor, as opposed to the more distributed vision of choreography. The typical example is that of a ballet: the choreographer fixes

* Research supported by the EU FET-GC2 IST-2004-16004 Integrated Project SENSORIA and by the Italian FIRB Project TOCAI.IT.

the overall scheme for the movements of all dancers, but then each dancer orchestrates her/his own movements. Roughly, from a formal modelling viewpoint, orchestration is mainly concerned with the regulation of control and data flow between services, while choreography is concerned with interaction protocols between single and composite services. In this paper we focus on orchestration, but with an eye left to choreography.

Cook and Misra's Orc [20,19] is a basic programming model for structured orchestration of services, whose primitives meet simplicity with yet great generality. The basic computational entities orchestrated by Orc expressions are *sites*: upon invocation, a site can publish at most one response value. A site call can be an RMI, a call to a monitor procedure, to a function or to a web service. A site computation might itself start other orchestrations, store effects locally and make (or not) such effects visible to clients.

Orc has three composition principles. The first one is the ordinary parallel composition $f|g$ (e.g., the parallel composition of two site calls can produce zero, one or many values). The other two, sequencing and asymmetric parallel composition, take inspiration from universal and existential quantification, respectively. In the sequential expression $f > x > g$, a fresh copy $g[v/x]$ of g is executed for *any* value v returned by f, i.e., a sort of pipeline is established between f and g. The evaluation of the asymmetric parallel expression f **where** $x :\in g$ is lazy: f and g start in parallel, but all sub-expressions of f that depend on the value of x must wait for g to publish *one* value. When g produces a value it is assigned to x and that side of the orchestration is cancelled.

As a workflow language, Orc can encode all most common workflow patterns [11]. Contrary to many other process algebras, Orc neatly separates orchestration from computation: Orc expressions should be considered as scripts to be invoked, e.g., within imperative programming languages using assignments such as $z :\in e$, where z is a variable and the Orc expression e can involve wide-area computation over multiple servers. The assignment symbol $:\in$ (due to Hoare) makes it explicit that e can return zero or more results, one of which is assigned to z.

This papers tries to characterise the distinguishing features of Orc by carrying a comparison with two other main paradigms, namely Petri nets and Join calculus as suitable representatives of workflow and messaging models, respectively. (The basics of Orc, Petri nets and Join are recalled in § 2.) Petri nets are a foundational model of concurrency, hence their choice as a reference model for carrying the comparison is well justified. The choice of Join instead of, e.g., the maybe more popular pi-calculus, might appear less obvious, so it is worth giving some explanation.

First, the multiple input prefix of Join looks more suitable than the single prefix of pi-calculus to smoothly model many orchestration patterns. For example, consider the process that must wait for messages on both x and y or in either one of the two. This is coded in Join as $x\langle u\rangle|tok\langle\rangle \triangleright P_1 \wedge y\langle v\rangle|tok\langle\rangle \triangleright P_2 \wedge x\langle u\rangle|y\langle v\rangle|tok\langle\rangle \triangleright P_3$ and by assuring there is a unique message $tok\langle\rangle$, whereas the pi-calculus expression $x(u).P_1 + y(v).P_2 + x(u).y(v).P_3$ used, e.g., in [23] is a less faithful encoding, because: (i) in the third sub-expression multiple inputs must be arbitrarily sequentialised and (ii) the third alternative can be selected even if a message arrives on x but none arrives on y, causing a deadlock. Of course one can still use the more precise translation $x(u).(P_1 + y(v).P_3) + y(v).(x(u).P_3 + P_2)$ but it is immediately seen that combinatorial explosion would make the encoding unreadable when larger groups of actions and more complex patterns are considered. Second, Join adheres to a locality principle ensuring that extruded names cannot be used in input by the process that received them (they can only output values on such ports). This feature is crucial for deploying distributed implementations [10,7] and it is not enforced in the full pi-calculus. Third, but not last, in [9], Join has been envisaged as some kind of higher-order version of Petri nets making it easier to reconcile all views analysed here.

Our contribution shows that:

- In absence of mobility, P/T nets can encode Orc expressions when mono-sessions are considered.
- Serialised multi-sessions require reset nets [6,12] (as shown in § 3).
- The Join calculus encodes Orc primitives in a rather natural way (as shown in § 4, the only verbosity is due to the encoding of variables, which is also very simple).

The last item shows that Orc primitives can be seen as syntactic sugar for Join processes. Therefore, as an overall result, we would like to promote Join as an elegant language integrating workflow orchestration, messaging, and computation (see § 5).

2 Background

2.1 Orc

This section briefly recaps Orc, borrowing definitions from [20] (apart from minor syntactical differences). Orc relies on the notion of a *site*, an abstraction amenable for being invoked and for publishing values. Each site invocation to s elicits at most one value published by s. Sites can be composed (by means of sequential and symmetric/asymmetric parallel composition) to form expressions. The difference between sites and expressions is that the latter can publish more than one value for each evaluation.

The syntax of Orc is given by the following grammar

$$D ::= E(x_1,\ldots,x_n) \underline{\Delta} f$$

$$e,f,g ::= 0 \;\big|\; M\langle p_1,\ldots,p_n\rangle \;\big|\; E\langle p_1,\ldots,p_n\rangle \;\big|\; f > x > g \;\big|\; f|g \;\big|\; f \textbf{ where } x :\in g$$

where $x_1,\ldots,x_n$ are variables, M stands for site names and E for expression names. We consider a set of constants $\mathcal{C}$ ranged over by c and the special site $let(x_1,\ldots,x_n)$ that publishes the tuple $\langle c_1,\ldots,c_n\rangle$. A value is either a variable, a site name or a constant (values are ranged over by $p_1, p_2, \ldots$).

The expressions g **where** $x :\in f$ and $f > x > g$ bind the occurrences of x in g (in g **where** $x :\in f$, the expression g is said to be in the scope of $x :\in f$). The occurrences of variables not bound are free and the set of free variables of an expression f is denoted by $fn(f)$. In the following, all definitions $E(x_1,\ldots,x_n) \underline{\Delta} f$ are well-formed, i.e., $fn(f) \subseteq \{x_1,\ldots,x_n\}$ and $x_1,\ldots,x_n$ are pairwise distinct. We write $\vec{x}$ for $x_1,\ldots,x_n$ and $f[c/x]$ for the expression obtained by replacing the free occurrences of x in f with c.

The operational semantics of Orc is formalised in Figure 1 as a labelled transition system with four kinds of labels: (1) a site call event $M(\vec{c},k)$, representing a call to site M with arguments $\vec{c}$ waiting for response on the dedicated handler k; (2) a response

$$\frac{}{let(c) \xrightarrow{!c} 0}\ (\text{LET}) \qquad \frac{k \text{ globally fresh}}{M\langle\vec{c}\rangle \xrightarrow{M(\vec{c},k)} ?k}\ (\text{SITECALL}) \qquad \frac{}{?k \xrightarrow{k?c} let(c)}\ (\text{SITERET}) \qquad \frac{E(\vec{x}) \underline{\Delta} f}{E\langle\vec{p}\rangle \xrightarrow{\tau} f[\vec{p}/\vec{x}]}\ (\text{DEF})$$

$$\frac{f \xrightarrow{!c} f'}{f > x > g \xrightarrow{\tau} (f' > x > g) \,|\, g[c/x]}\ (\text{SEQPIPE}) \qquad \frac{f \xrightarrow{l} f' \quad l \neq !c}{f > x > g \xrightarrow{l} f' > x > g}\ (\text{SEQ})$$

$$\frac{g \xrightarrow{l} g'}{g \,|\, f \xrightarrow{l} g' \,|\, f}\ (\text{SYML}) \qquad \frac{f \xrightarrow{l} f'}{g \,|\, f \xrightarrow{l} g \,|\, f'}\ (\text{SYMR}) \qquad \frac{g \xrightarrow{l} g'}{g \textbf{ where } x :\in f \xrightarrow{l} g' \textbf{ where } x :\in f}\ (\text{ASYML})$$

$$\frac{f \xrightarrow{l} f' \quad l \neq !c}{g \textbf{ where } x :\in f \xrightarrow{l} g \textbf{ where } x :\in f'}\ (\text{ASYMR}) \qquad \frac{f \xrightarrow{!c} f'}{g \textbf{ where } x :\in f \xrightarrow{\tau} g[c/x]}\ (\text{ASYMPRUNE})$$

Fig. 1. Operational semantics of Orc.

event $k?c$, sending the response c to the call handler k (there is at most one such event for each k); (3) a publish event $!c$; (4) an internal event τ.

A declaration D specifies an expression name E, the formal parameters $x_1, \ldots, x_n$ and the body f, like for usual function or procedure declarations. The body f of an expression declaration can be the expression 0 (i.e., a site which never publishes any value), the invocation of a site $M\langle p_1, \ldots, p_n\rangle$, or an expression call $E\langle p_1, \ldots, p_n\rangle$. Calls to sites are strict (actual parameters are evaluated before the call) while expression calls are non-strict. Expressions f and g can be sequentially composed with $f > x > g$ which first evaluates f and then, for each value v published by f, evaluates a new copy of g where x is replaced with v (if f never publishes any value, no fresh g will ever be evaluated). Expressions can be composed with the symmetric and asymmetric parallel operators. The former is written $f|g$; it evaluates f and g in parallel and publishes the values that f and g publish (we remark that there is no interaction between f and g and that usual monoidal laws for $|$ with 0 as neutral element hold). The latter, called *where-expression*, is written g **where** $x :\in f$. The evaluation of g **where** $x :\in f$ proceeds by evaluating f and g in parallel. Expression f is meant to publish a value to be assigned to x and all the parts of g depending on x must wait until such a value is available. Evaluation of f stops as soon as any value, say v, is published. Then, v is assigned to x so that all the parts in g depending of x can execute, but the residual of f is cancelled.

Example 2.1. We borrow from [20] some of interesting examples of Orc declarations.

- Assume that *CNN* and *BBC* are two sites that return recent news when invoked while site $\mathit{Email}(a,m)$ sends an email containing message m to the address a. (Notice that an invocation to *Email* changes the receiver's mailbox).
- Declaration $\mathit{Notify}(a) \underline{\Delta} (\mathit{CNN}|\mathit{BBC}) > x > \mathit{Email}(x,a)$ specifies a service for notifying last news from *CNN and BBC*. By rule SEQPIPE, the news from both *CNN* and *BBC* are notified in two different emails.
- Another interesting example is $\mathit{MailOnce}(a) \underline{\Delta}\, \mathit{Email}(x,a)$ **where** $x :\in (\mathit{CNN}|\mathit{BBC})$ specifying service $\mathit{MailOnce}(a)$ that notifies address a with only one of the news selected either from *CNN* or from *BBC*.

An Orc program represents an orchestrator $\mathcal{O}$ executed in a host sequential program; $\mathcal{O}$ is a pair $\langle \mathcal{D}, z :\in E(\vec{p})\rangle$ where $\mathcal{D}$ is a set of definitions, z a variable of the host program, $E\langle\vec{c}\rangle$ is an Orc expression call where (i) E is an expression name defined in $\mathcal{D}$ and (ii) $\vec{c}$ are the actual parameters. The notation $z :\in E\langle\vec{c}\rangle$ specifies that even if $E\langle\vec{c}\rangle$ might publish any number of values, z will be bound to just one of them. The types of values published by $E\langle\vec{c}\rangle$ are left unspecified, however it is assumed that they can be dealt with in the hosting program (see § 2.2 of [20]).

2.2 Petri Nets

Petri nets, introduced in [21], have become a reference model for studying concurrent systems, mainly due to their simplicity and the intrinsic concurrent nature of their behaviour. They rely on solid theoretical basis that allows for the formalisation of causality, concurrency, and non-determinism (in terms of non-sequential processes or unfolding constructions). Petri nets are built up from *places* (denoting resources types), which are repositories of *tokens* (representing instances of resources), and *transitions*, which fetch and produce tokens. We assume an infinite set $\mathcal{P}$ of resource names is fixed.

Definition 2.1 (Net). *A* net *N is a 4-tuple $N = (S_N, T_N, \delta_{0N}, \delta_{1N})$ where $S_N \subseteq \mathcal{P}$ is the (nonempty) set of places, $\mathtt{a}, \mathtt{a}', \ldots$, T_N is the set of transitions, $\mathtt{t}, \mathtt{t}', \ldots$ (with $S_N \cap T_N = \emptyset$), and the functions $\delta_{0N}, \delta_{1N} : T_N \to \wp_{\mathrm{f}}(S_N)$ assign finite sets of places, called respectively source and target, to each transition.*

Place / Transition nets (P/T nets) are the most widespread model of nets. The places of a P/T net can hold zero, one or more tokens and arcs are weighted. Hence, the state of the P/T net is described in terms of *markings*, i.e., multisets of tokens available in the places of the net. Given a set S, a *multiset* over S is a function $m : S \to \mathbb{N}$ (where $\mathbb{N}$ is the set of natural numbers). The set of all finite multisets over S is denoted by $\mathcal{M}_S$ and the empty multiset by $\emptyset$.

Definition 2.2 (P/T net). *A* marked place / transition Petri net *(P/T net) is a tuple $N = (S_N, T_N, \delta_{0N}, \delta_{1N}, m_{0N})$ where $S_N \subseteq \mathcal{P}$ is a set of places, T_N is a set of transitions, the functions $\delta_{0N}, \delta_{1N} : T_N \to \mathcal{M}_{S_N}$ assign respectively, source and target to each transition, and $m_{0N} \in \mathcal{M}_{S_N}$ is the initial marking.*

Given a transition $\mathtt{t} \in T$, ${}^\bullet\mathtt{t} = \delta_0(\mathtt{t})$ is its *preset* and $\mathtt{t}^\bullet = \delta_1(\mathtt{t})$ is its *postset*. Let N be a net and u a marking of N; then a transition $\mathtt{t} \in T_N$ is *enabled at u* iff ${}^\bullet\mathtt{t}(\mathtt{a}) \leq u(\mathtt{a}), \forall \mathtt{a} \in S_N$. We say a marking u evolves to u' under the *firing* of the transition $\mathtt{t}$ written $u[\mathtt{t}\rangle u'$, iff $\mathtt{t}$ is enabled at u and $u'(\mathtt{a}) = u(\mathtt{a}) - {}^\bullet\mathtt{t}(\mathtt{a}) + \mathtt{t}^\bullet(\mathtt{a}), \forall \mathtt{a} \in S$. A *firing sequence* from u_0 to u_n is a sequence of markings and firings s.t. $u_0[\mathtt{t_1}\rangle u_1 \ldots u_{n_1}[\mathtt{t_n}\rangle u_n$.

Reset nets [6] extend P/T nets with special *reset arcs*. A reset arc associating a transition $\mathtt{t}$ with a place $\mathtt{a}$ causes the place $\mathtt{a}$ to reset when $\mathtt{t}$ is fired.

Definition 2.3 (Reset net). *A* reset net *is a tuple $N = (S_N, T_N, \delta_{0N}, \delta_{1N}, m_{0N}, R_N)$, where $(S_N, T_N, \delta_{0N}, \delta_{1N}, m_{0N})$ is a P/T net and $R_N : T_N \to \wp_{\mathrm{f}}(S_N)$ defines reset arcs.*

The condition for the enabling of a reset transition is the same as for ordinary P/T nets, while their firings are defined as follows: u evolves to u' under the *firing* of the reset transition $\mathtt{t}$, written $u[\mathtt{t}\rangle u'$, if and only if $\mathtt{t}$ is enabled at u and $\forall \mathtt{a} \in S_N : u'(\mathtt{a}) = u(\mathtt{a}) - {}^\bullet\mathtt{t}(\mathtt{a}) + \mathtt{t}^\bullet(\mathtt{a})$ if $\mathtt{a} \notin R_N(t)$, and $u'(\mathtt{a}) = 0$ otherwise.

$$
\begin{array}{llll}
(\text{OPEN}) & A,B ::= 0 \mid x\langle\vec{y}\rangle \mid \mathbf{def}_S\ D \ \mathbf{in}\ A \mid A|B & D,E ::= J \triangleright P \mid D \wedge E & (\text{DEF}) \\
(\text{PROC}) & P,Q ::= 0 \mid x\langle\vec{y}\rangle \mid \mathbf{def}\ D \ \mathbf{in}\ P \mid P|Q & J,K ::= x\langle\vec{y}\rangle \mid J|K & (\text{PAT})
\end{array}
$$

(a) Syntax

$$
\begin{array}{rlrl}
rn(x\langle\vec{y}\rangle) &= \{\vec{y}\} & dn(x\langle\vec{y}\rangle) &= \{x\} \\
rn(J|K) &= rn(J) \uplus rn(K) & dn(J|K) &= dn(J) \uplus dn(K) \\
fn(J \triangleright P) &= dn(J) \cup (fn(P) \backslash rn(J)) & dn(J \triangleright P) &= dn(J) \\
fn(D \wedge E) &= fn(D) \cup fn(E) & dn(D \wedge E) &= dn(D) \cup dn(E) \\
fn(0) &= \emptyset & xn(0) &= \emptyset \\
fn(x\langle\vec{y}\rangle) &= \{x\} \cup \{\vec{y}\} & xn(x\langle\vec{y}\rangle) &= \emptyset \\
fn(\mathbf{def}_S\ D \ \mathbf{in}\ A) &= (fn(D) \cup fn(P)) \backslash dn(D) & xn(\mathbf{def}_S\ D \ \mathbf{in}\ A) &= S \uplus xn(A) \\
fn(A|B) &= (fn(A) \backslash xn(B)) \cup (fn(B) \backslash xn(A)) & xn(A|B) &= xn(A) \uplus xn(B)
\end{array}
$$

(b) Free, Defined, Bound and Received names

$$
\begin{array}{lrcll}
(\text{STR-NULL}) & \Vdash_S 0 & \leftrightharpoons & \Vdash_S & \\
(\text{STR-JOIN}) & \Vdash_S P \mid Q & \leftrightharpoons & \Vdash_S P,Q & \\
(\text{STR-AND}) & D \wedge E \Vdash_S & \leftrightharpoons & D,E \Vdash_S & \\
(\text{STR-DEF}) & \Vdash_S \mathbf{def}_{S'}\ D \ \mathbf{in}\ P & \leftrightharpoons & D\sigma \Vdash_{S \uplus S'} P\sigma & \sigma \text{ a globally fresh renaming of } dn(D) \backslash S' \\
(\text{RED}) & J \triangleright P \Vdash_S J\sigma & \xrightarrow{\tau} & J \triangleright P \Vdash_S P\sigma & \\
(\text{EXT}) & \Vdash_S x\langle\vec{u}\rangle & \xrightarrow{S' x\langle\vec{u}\rangle} & \Vdash_{S \uplus S'} & x \text{ is free, and } S' \text{ are the local names in } \vec{u} \text{ not in } S \\
(\text{INT}) & \Vdash_{\{x\} \uplus S} & \xrightarrow{x\langle\vec{u}\rangle} & \Vdash_{\{x\} \uplus S} x\langle\vec{u}\rangle & \vec{u} \text{ contains free, extruded and fresh names}
\end{array}
$$

(c) Semantics

Fig. 2. Open-join Calculus.

2.3 Join **Calculus**

This section summarises the basics of the Open-join [15], a conservative extension of Join [14] equipped with the notion of weak bisimulation used in § 4. We rely on an infinite set of names $x, y, u, v, \dots$ each carrying fixed length tuple of names (denoted as $\vec{u}$). A sorting discipline that avoids arity mismatch is implicitly assumed and only well-sorted terms are considered. Open processes A, processes P, definitions D and patterns J are defined in Figure 2(a). A Join process is either the inert process 0, the asynchronous emission $x\langle\vec{y}\rangle$ of message on port x that carries a tuple of names $\vec{y}$, the process **def** D **in** P equipped with local ports defined by D, or a parallel composition of processes $P|Q$. An open process A is like a Join process, except that it has open definitions at top-level. The open definition $\mathbf{def}_S$ D **in** P exhibits a subset S of names defined by D that are visible from the environment: the *extruded names*. Open processes are identified with ordinary Join processes when the set S of extruded names is empty. A *definition* is a conjunction of elementary reactions $J \triangleright P$ that associate *join-patterns* J with *guarded processes* P.

The sets of defined names dn, received names rn, free names fn and extruded names xn are shown in Figure 2(b) ($\uplus$ denotes the disjoint union of sets). Note that the extruded names of two parallel processes are required to be disjoint because they are introduced by different definitions. Similarly, the extruded names S of $\mathbf{def}_S\ D$ **in** A are disjoint from the extruded names of A. As usual, patterns are required to be disjoint.

The semantics of the Open-join calculus relies on the *open reflexive chemical abstract machine* model (*Open* RCHAM) [15]. A solution of an Open RCHAM is a triple $(\mathcal{R},\mathcal{S},\mathcal{A})$, written $\mathcal{R} \Vdash_{\mathcal{S}} \mathcal{A}$, where $\mathcal{A}$ is a multiset of open processes with disjoint sets of extruded names, $\mathcal{R}$ is a multiset of active definitions s.t. $dn(\mathcal{R}) \cap xn(\mathcal{A}) = \emptyset$, and $\mathcal{S} \subseteq dn(\mathcal{R})$ is a set of extruded names (*fn*, *dn* and *xn* lift to multisets in the obvious way). Moves are distinguished between *structural* $\rightleftharpoons$ (or heating/cooling), which stand for the syntactical rearrangement of terms, and reductions $\rightarrow$, which are the basic computational steps. The multiset rewriting rules for Open-join are shown in Figure 2(c). Rule STR-NULL states that 0 can be added or removed from any solution. Rules STR-JOIN and STR-AND stand for the associativity and commutativity of $|$ and $\wedge$, because $\_\,,\_$ is such. STR-DEF denotes the activation of a local definition, which implements a static scoping discipline by properly renaming defined ports by *globally fresh* names.

Reduction rules are labelled either by (i) internal reduction τ, (ii) output messages $Sx\langle\vec{u}\rangle$ on the free port x that extrude the set S of local names, or (iii) $x\langle\vec{u}\rangle$ denoting the intrusion of a message on the already extruded local name x. Rule RED describes the use of an active reaction rule $(J \triangleright P)$ to consume messages forming an instance of J (for a suitable substitution σ, with $dom(\sigma) = rn(J)$), and to produce a new instance $P\sigma$ of its guarded process P. Rule (EXT) consumes messages sent on free names; these messages may extrude some names S' for the first time, thus increasing the set of extruded names. Rule (INT) stands for the intrusion of a message on a defined-extruded name. We remark that rules are local and describe only the portion of the solution that actually reacts. Hence, any rule can be applied in a larger context.

3 Orc vs Petri Nets

In this section we sketch an intuitive explanation of Orc basic orchestration primitives in terms of Petri nets. At first glance, the composition patterns available in Orc can seem easily representable using (workflow) Petri nets. Assume that each Orc expression f is represented by a suitable net N_f with two distinguished places in_f (for getting tokens in input that activate the net) and out_f for publishing tokens. A pipeline between the nets N_f and N_g can be obtained by adding just one transition from out_f to in_g. Similarly, the parallel composition of N_f and N_g can be obtained by adding places $in_{f|g}$ and $out_{f|g}$ with three transitions: (i) from $in_{f|g}$ to in_f and in_g, (ii) from out_f to $out_{f|g}$, and (iii) from out_g to $out_{f|g}$. Finally asymmetric parallel composition can be obtained by adding a place $wh_{f,g}$ with just one token in it and no incoming arc, together with a transition from out_f and $wh_{f,g}$ to in_g (so that such transition can be fired at most once).

However it is easy to realise that the modelling is not as simple as above. Take Example 2.1, where two instances of $Email(\_,a)$ can concurrently run when $Notify(a)$ is invoked. If site invocation is modelled by passing the control-flow token to the net representing *Email*, then the tokens of two different sessions can be mixed! Apart from the cumbersome solution of representing sessions identifiers within tokens, there are two possible solutions to the multi-session problem (i.e., the possibility of re-using parts of the net when for different invocations). The first is to replicate the net corresponding to

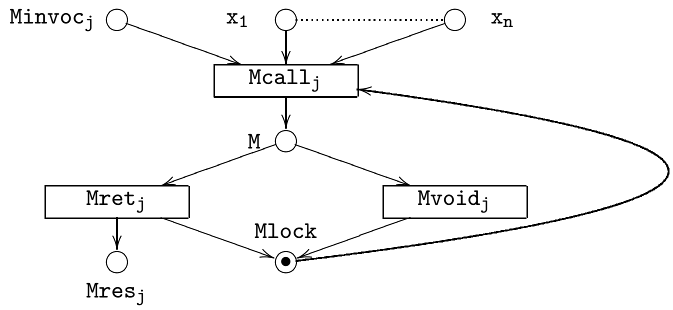

Fig. 3. Net for the invocation of a site M

the body of an expression for each invocation, while the second is using dynamic nets, where fresh ports of the net can be released during the execution. The first alternative is considered here, while § 4 provides an encoding of Orc in Join, as a linguistic counterpart of dynamic nets [9]. Another problem is that expressions can carry arguments, so that more than one input place can be needed (e.g., one for each variable).

In this section we shall focus on Orc$^-$, a simplified fragment of Orc, where recursion is avoided and values are not considered. Avoiding recursion is necessary in order to have finite nets, indeed each invocation will result in a new instance of the body of the defined expression. Petri nets can encode Orc$^-$ expressions in absence of mobility and when each expression is evaluated at most once (i.e., when mono-session are considered). Multi-sessions require reset arcs and can only be dealt with by serialising the accesses to the re-used part of the net. We prefer to keep the presentation of the Petri net semantics at an informal level. A more technical presentation is postponed to § 4, where the concurrent multi-session problem is tackled by establishing a strong formal correspondence between observational semantics of Orc and its encoding in Join.

Since recursion is banned from Orc$^-$ programs, invocations to site (resp. expressions) can be enumerated and we write $M_j\langle x_1,\ldots,x_n\rangle$ (resp. $E_j\langle x_1,\ldots,x_n\rangle$) to denote the j-th invocation to site M (resp. expression E). The main difference w.r.t. Orc, is that Orc$^-$ uses names (i.e., variables) only for passing signals. For instance, the sequential operator of Orc$^-$ is simplified as $f >> g$. This implies that variables are only required for site invocations and asymmetric parallel composition, say $\_$ **where** $z :\in g$. In the former case, variables are used to render the strict policy of site invocation in the Petri net encoding. In the latter case, z will be used as the output place for the net representing g.

Let $\langle \mathcal{D}, z :\in E(\vec{p})\rangle$ be an Orc$^-$ program. The encoding of an Orc$^-$ expression f into Petri nets is denoted as $[\![\, f \,]\!]_{\mathcal{D}}^{\mathtt{f_i},\mathtt{f_o}}$ where $\mathtt{f_i}$ and $\mathtt{f_o}$ are distinguished places (entry and exit points of f). The idea is that $\mathrm{f_i}$ is the place for the activation of f and $\mathrm{f_o}$ is the place for returning the control. Data dependencies/flows due to asymmetric parallelism are rendered by places associated with variable names, which may coincide with output places of other parts of the net (to store results).

We first consider the translation $[\![\, M_j\langle x_1,\ldots,x_n\rangle \,]\!]_{\mathcal{D}}^{\mathtt{Minvoc_j},\mathtt{Mres_j}}$ of the j-th invocation to site M (see Figure 3). The places `M` and `Mlock` are shared by all the invocations and `Mlock` is meant as a lock mechanism for serialising multiple concurrent invocations to

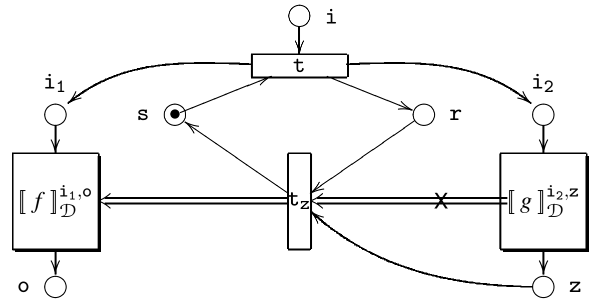

Fig. 4. A schema of net for $[\![f \textbf{ where } z :\in g]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}}$.

M. Indeed, $\mathtt{Mcall_j}$ is enabled only if `Mlock` contains a token as it initially does. The invocation takes place when $\mathtt{Minvoc_j}$ and all places $\mathtt{x}_1, \dots, \mathtt{x}_n$ contain a token (i.e., the actual parameters have been evaluated). Moreover, both $\mathtt{Mret_j}$ and $\mathtt{Mvoid_j}$ put a token into `Mlock` once they are fired so that the next invocation can proceed. Remarkably, $\mathtt{Mret_j}$ and $\mathtt{Mvoid_j}$ are mutually exclusive: $\mathtt{Mret_j}$ models the case in which M returns a value, while $\mathtt{Mvoid_j}$ the case in which no value is returned to the invoker.

If $E\langle z_1, \dots, z_n \rangle \underline{\Delta} f \in \mathcal{D}$, the j-th invocation of E, say $E_j\langle x_1, \dots, x_n \rangle$, is translated as $[\![E_j\langle x_1, \dots, x_n \rangle]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}} = [\![f[x_1/z_1, \dots, x_n/z_n]]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}}$. For the remaining constructs of Orc^-, the simplest case is $[\![0]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}}$ which is the net with a single transition whose preset is the place i and whose postset is empty. For the sequential operator, $[\![f >> g]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}} = [\![f]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}'} \cup [\![g]\!]_{\mathcal{D}}^{\mathtt{o}',\mathtt{o}}$ where o' is a fresh place and where given two nets N_1 and N_2, we write $N_1 \cup N_2$ for the net whose set of places (resp. transitions) is the union of the places (resp. transitions) of N_1 and N_2 and, for each transition `t` in N_1 and N_2, the preset (resp. postset) of `t` is the union of the presets (resp. postset) of `t` in N_1 and N_2. For the parallel operator, given two fresh places $\mathtt{i}_1$ and $\mathtt{i}_2$, $[\![f|g]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}} = [\![f]\!]_{\mathcal{D}}^{\mathtt{i}_1,\mathtt{o}} \cup [\![g]\!]_{\mathcal{D}}^{\mathtt{i}_2,\mathtt{o}} \cup N$ where N is the net made by a single transition with preset `i` and postset $\{\mathtt{i}_1, \mathtt{i}_2\}$.

Where-expressions, say f **where** $z :\in g$, require some subtlety, because their evaluation requires that g terminates when a value for z is available. In our encoding, this is modelled by resetting all the places of g.

$$[\![f \textbf{ where } z :\in g]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}} = [\![f]\!]_{\mathcal{D}}^{\mathtt{i}_1,\mathtt{o}} \cup [\![g]\!]_{\mathcal{D}}^{\mathtt{i}_2,\mathtt{z}} \cup R \tag{1}$$

where $\mathtt{i}_1, \mathtt{i}_2$ are two new places and R is a net for connecting $[\![f]\!]_{\mathcal{D}}^{\mathtt{i}_1,\mathtt{o}}$ and $[\![g]\!]_{\mathcal{D}}^{\mathtt{i}_2,\mathtt{z}}$ and for resetting the places of $[\![g]\!]_{\mathcal{D}}^{\mathtt{i}_2,\mathtt{z}}$. More precisely, R contains

1. the place `i` together with two fresh places `s` and `r` and a token in `s`;
2. a fresh transition `t` such that $^\bullet\mathtt{t} = \{\mathtt{i}, \mathtt{s}\}$ and $\mathtt{t}^\bullet = \{\mathtt{i}_1, \mathtt{i}_2, \mathtt{r}\}$;
3. a fresh transition $\mathtt{t_z}$ such that $^\bullet\mathtt{t_z} = \{\mathtt{z}, \mathtt{r}\}$ and $\mathtt{t_z}^\bullet$ includes `s` and the set of all the places in $[\![f]\!]_{\mathcal{D}}^{\mathtt{i}_1,\mathtt{o}}$ corresponding to the occurrences of z in f, moreover, for each place `p` in $[\![g]\!]_{\mathcal{D}}^{\mathtt{i}_2,\mathtt{z}}$ (including $\mathtt{i}_2$ and `z`), there is a reset arc from `p` to $\mathtt{t_z}$.

A pictorial representation of $[\![f \textbf{ where } z :\in g]\!]_{\mathcal{D}}^{\mathtt{i},\mathtt{o}}$ is given in Figure 4 where the bold boxes represent the nets for f and g; the double arrow is the set of arcs described in 3 and the crossed double arrow is the set or reset arcs described in 3. Places $\mathtt{s}$ and $\mathtt{r}$ serialise the activation of f and g. When a token is available on $\mathtt{z}$, then $\mathtt{t_z}$ can be fired: it distributes the token to all the occurrences of z in f, resets g and enables further activation of the net by restoring the token in $\mathtt{s}$.

Reset arcs are not needed if just mono-sessions are considered. As an alternative to reset arcs, inhibitor arcs [13] could have been used. However, the net R in (1) would have been more complex.

4 Encoding Orc in the Join Calculus

When encoding Orc$^-$ either into reset or inhibitor nets, different evaluations of certain expressions are computed sequentially (e.g., site calls, where-expressions). Since in general it is possible to write expressions involving an unbounded number of concurrently executing sessions, it is evident that any static net topology will either introduce some serialisation or mix tokens from different sessions.

In this section, we propose an encoding of full Orc into Join in which different evaluations of the same expression can be computed concurrently. This is achieved by taking advantage of the reflexive mechanism provided by Join and dynamic nets that allows for the dynamic creation of places and transitions. The main strategy of the encoding is to associate a fresh portion of a net to any evaluation of an Orc expression. That is, if the evaluation of an expression f can be represented by a net N_f, we assure that any evaluation of f is performed by a fresh copy of N_f. In this way confusions among concurrent evaluations of the same expression are avoided.

Definition 4.1. *Let $\mathcal{O} = \langle \mathcal{D}, z :\in E(P) \rangle$ be an* Orc *orchestrator. Then, the corresponding* Join *process is $P_{\mathcal{O}} = \{[\mathcal{O}]\}$, where $\{[\_]\}$ is inductively defined in Figure 5.*

We comment on the definitions in Figure 5. Any Orc definition $D \in \mathcal{D}$ becomes a local definition $\{[D]\}$ of the corresponding Join process $P_{\mathcal{O}}$, while the initial expression $z :\in E(p_1,\ldots,p_n)$ becomes the active process $\{[E(p_1,\ldots,p_n)]\}_z$. Note that the initial expression $E(p_1,\ldots,p_n)$ is encoded by considering a context z (i.e., the channel z encodes the homonymous variable). In this way, $P_{\mathcal{O}}$ will send a message $z\langle v\rangle$ for any value v obtained during the evaluation of $E(p_1,\ldots,p_n)$. Any Orc definition $E(x_1,\ldots,x_n) \underline{\Delta} f$ is translated as a basic rule $E(q_1,\ldots,q_n,z) \triangleright \{[f]\}_z$, where z is a fresh name used for returning the values produced during the evaluation of f, i.e., z is used for implementing the usual continuation passing style of Join.

All remaining rules define the translation of expressions. In particular, the inert Orc expression 0 is translated as the inert Join process 0, while the constant expression c is encoded as $z\langle c\rangle$, i.e., as the assignment of the unique value c to z. Differently, the encoding of an expression consisting of a variable x is translated as a message $x\langle z\rangle$. In fact, as we will see later, we associate any Orc variable with a basic Join definition that answers any message $x\langle z\rangle$ with $z\langle v\rangle$ if x has been assigned a value v. Moreover, any request $x\langle z\rangle$ will be blocked until a value is assigned to x.

The invocation $M(p_1,\ldots,p_n)$ of a service M is translated as a process that starts by evaluating all actual parameters p_i. Since actual parameters can be only constants or variables, the evaluation of $\{[p_i]\}_{y_i}$ will finish by producing messages $y_i\langle x_i\rangle$ on fresh

$$
\begin{array}{rcl}
\{[\mathcal{O} = \langle \mathcal{D}, z :\in E(P)\rangle]\} & = & \mathbf{def} \bigwedge_{D\in\mathcal{D}} \{[D]\} \ \mathbf{in}\ \{[E(P)]\}_z \\
\{[E(q_1,\ldots,q_n) \,\underline{\Delta}\, f]\} & = & E(q_1,\ldots,q_n,z) \triangleright \{[f]\}_z \quad \text{with } z \notin \{q_1,\ldots,q_n\} \\
\{[0]\}_z & = & 0 \\
\{[c]\}_z & = & z\langle c\rangle \\
\{[x]\}_z & = & x\langle z\rangle \\
\{[M(p_1,\ldots,p_n)]\}_z & = & \mathbf{def}\ y_1\langle x_1\rangle|\ldots|y_n\langle x_n\rangle \triangleright \mathbf{def}\ k\langle v\rangle|tok\langle\rangle \triangleright z\langle v\rangle \\
 & & \qquad\qquad\qquad\qquad \mathbf{in}\ M\langle x_1,\ldots,x_n,k\rangle \mid tok\langle\rangle \\
 & & \mathbf{in}\ \{[p_1]\}_{y_1}|\ldots|\{[p_n]\}_{y_n} \\
\{[X(p_1,\ldots,p_n)]\}_z & = & \mathbf{def}\ y_1\langle x_1\rangle|\ldots|y_n\langle x_n\rangle|y\langle M\rangle \triangleright \mathbf{def}\ k\langle v\rangle|tok\langle\rangle \triangleright z\langle v\rangle \\
 & & \qquad\qquad\qquad\qquad \mathbf{in}\ M\langle x_1,\ldots,x_n,k\rangle \mid tok\langle\rangle \\
 & & \mathbf{in}\ \{[p_1]\}y_1|\ldots|\{[p_n]\}y_n|\{[X]\}y \\
\{[E(p_1,\ldots,p_n)]\}_z & = & \mathbf{def}\ \wedge_{p\in\{p_1,\ldots,p_n\}\cap\mathcal{C}}\, fwd_p\langle k\rangle \triangleright k\langle p\rangle \\
 & & \mathbf{in}\ E((\!|p_1|\!),\ldots,(\!|p_n|\!),z) \\
 & & \text{where } (\!|p_i|\!) = p_i \ \textit{if}\ p_i \notin \mathcal{C} \ \textit{and}\ (\!|p_i|\!) = fwd_{p_i}\ \textit{otherwise} \\
\{[f \mid g]\}_z & = & \{[f]\}_z \mid \{[g]\}_z \\
\{[f > x > g]\}_z & = & \mathbf{def}\ w\langle v\rangle \triangleright \mathbf{def}\ x\langle y\rangle \mid val_x\langle u\rangle \triangleright y\langle u\rangle \mid val_x\langle u\rangle \\
 & & \qquad\qquad \mathbf{in}\ \{[g]\}_z \mid val_x\langle v\rangle \\
 & & \mathbf{in}\ \{[f]\}_w \\
\{[g\ \mathbf{where}\ x :\in f]\}_z & = & \mathbf{def}\ x\langle y\rangle \mid val_x\langle u\rangle \triangleright y\langle u\rangle \mid val_x\langle u\rangle \\
 & & \wedge\ w\langle v\rangle \mid tok\langle\rangle \triangleright val_x\langle v\rangle \\
 & & \mathbf{in}\ \{[g]\}_z \mid \{[f]\}_w \mid tok\langle\rangle
\end{array}
$$

Fig. 5. Encoding of an Orc Orchestrator in Join.

names y_i. Hence, the unique local definition is enabled only when all actual parameters have been completely evaluated. Moreover, the firing of the local rule creates two fresh ports: k and tok and a unique firing rule. Channel k indicates the port where the orchestrator awaits the answers of the invoked service (We assume the definition of any site to be extended in order to receive this extra parameter.) Channel tok assures that just one answer is considered for any invocation. In fact, there is only one message $tok\langle\rangle$, which is consumed (and it is not generated anymore) when the first message on k is received.

In case the name of the invoked service is the variable X, then X has to be evaluated before the invocation, just like any other actual parameter. The name of the site M will be returned as he value of X on port y.

The use of an Orc definition $E(p_1,\ldots,p_n)$ differs from the invocation of a service in the fact that definitions are called by following a lazy evaluation, i.e., parameters are not evaluated before the call. Hence, invocations of E can take place even though some of the formal parameters $p_1,\ldots,p_n$ have not been initialised. The local ports fwd_p introduced by the encoding if $p \in \mathcal{C}$ allow the constant parameters to be used as variables inside the expression defining E (see Example 4.1).

The encoding of a parallel composition $f|g$ corresponds to the parallel composition of the encodings of f and g. Note that both encoded expressions produce results on the same channel z. On the other hand, the sequential composition $f > x > g$ is translated as a process that starts by evaluating $\{[f]\}_w$ (i.e., the encoding of f) whose values will be sent as messages to the local port w. Hence, any message on w corresponds to the activation of a new evaluation of g. In fact, the local definition, which is enabled with a message $w\langle v\rangle$, will create a fresh copy of the encoding of g, which will evaluate g by considering the particular value v produced by f.

The last rule handles the translation of asymmetric parallel composition. Note that the encodings of f and g are activated concurrently. Unlike sequential composition, there is a unique copy of $\{[g]\}$ and a unique instance of the variable x. In fact, asymmetric composition requires to evaluate g just for one value of f. The unique message *tok* assures that only one value produced by $\{[f]\}$ will be set to the variable x.

Example 4.1. Let $\mathcal{O} = \langle\{d\}, z :\in \mathit{Invoke}(\mathit{StockQuote}, \mathit{Sun})\rangle$, with $d : \mathit{Invoke}(m,n) \,\underline{\Delta}\, m(n)$. The corresponding Join process is as follows.

$$
\begin{array}{l}
\{[\mathcal{O}]\} = \textbf{def}\ \ \mathit{Invoke}\langle m,n,z\rangle \triangleright \textbf{def}\ y_1\langle x_1\rangle | y\langle M\rangle \triangleright \textbf{def}\ k\langle v\rangle \mid \mathit{tok}\langle\rangle \triangleright z\langle v\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ M\langle x_1, k\rangle \mid \mathit{tok}\langle\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ n\langle y_1\rangle \mid m\langle y\rangle \\
\qquad\qquad\quad \textbf{in def}\ \mathit{fwd}_{\mathit{StockQuote}}\langle k\rangle \triangleright k\langle \mathit{StockQuote}\rangle \\
\qquad\qquad\qquad \wedge\ \mathit{fwd}_{\mathit{Sun}}\langle k\rangle \triangleright k\langle \mathit{Sun}\rangle \\
\qquad\qquad\qquad \textbf{in}\ \mathit{Invoke}\langle \mathit{fwd}_{\mathit{StockQuote}}, \mathit{fwd}_{\mathit{Sun}}, z\rangle
\end{array}
$$

Note the difference when calling a local definition (i.e., *Invoke*) and when invoking a service (i.e., *StockQuote*). In particular, actual parameters are not evaluated when calling a local definition. Moreover, a new forwarder is created for any constant parameter. In this case, the ports $\mathit{fwd}_{\mathit{StockQuote}}$ and $\mathit{fwd}_{\mathit{Sun}}$ are introduced and are used as actual parameters. In this way the definition of *Invoke* may handle all its parameters as if they were variables. In fact, when rule $\mathit{Invoke}\langle m,n,z\rangle \triangleright \ldots$ is fired by consuming the token $\mathit{Invoke}\langle \mathit{fwd}_{\mathit{StockQuote}}, \mathit{fwd}_{\mathit{Sun}}, z\rangle$, then the arguments m and n are evaluated by sending the messages $\mathit{fwd}_{\mathit{Sun}}\langle y_1\rangle$ and $\mathit{fwd}_{\mathit{StockQuote}}\langle y\rangle$, which will return the corresponding constants, i.e., the messages $y_1\langle \mathit{Sun}\rangle$ and $y\langle \mathit{StockQuote}\rangle$ will be produced.

The remaining part of this section is devoted to show the correspondence among Orc processes and their encoded form. The following definition introduces the equivalence notion we will use to compare Orc processes with their encoded form, which is a kind of weak bisimulation. In the following, given an Orc label α, the corresponding Join label is denoted with $\overline{\alpha}$ and it is defined as $\overline{M_k(v)} = \{k\}M\langle v,k\rangle$, $\overline{k?v} = k\langle v\rangle$, $\overline{!v} = \emptyset z\langle v\rangle$.

Definition 4.2 (Weak Bisimulation). *Let $\mathcal{O} = \langle \mathcal{D}, z :\in E(p_1,\ldots,p_n)\rangle$ be an orchestrator, and P be a* Join *process. We call* weak bisimulation *any relation $\mathcal{R}$ satisfying the following condition: $\mathcal{O}\,\mathcal{R}\,P$ iff*

1. $\mathcal{O} \xrightarrow{\alpha} \mathcal{O}'$ *and* $\alpha \neq \tau$ *then* $P \rightarrow^{*} \xrightarrow{\overline{\alpha}} P'$ *and* $\mathcal{O}'\,\mathcal{R}\,P'$
2. $\mathcal{O} \xrightarrow{\tau} \mathcal{O}'$ *then* $P \xrightarrow{\tau}{}^{*} P'$ *and* $\mathcal{O}'\,\mathcal{R}\,P'$
3. $P \xrightarrow{\overline{\alpha}} P'$ *and* $\alpha \neq k\langle v\rangle$ *then* $\mathcal{O} \rightarrow^{*} \xrightarrow{\alpha} \mathcal{O}'$ *and* $\mathcal{O}'\,\mathcal{R}\,P'$
4. $P \xrightarrow{k\langle v\rangle} P'$ *then either (i)* $\mathcal{O} \xrightarrow{k?v} \mathcal{O}'$ *and* $\mathcal{O}'\,\mathcal{R}\,P'$*, or (ii)* $\mathcal{O} \stackrel{k?v}{\not\longrightarrow}$ *and* $\mathcal{O}\,\mathcal{R}\,P'$
5. $P \xrightarrow{\tau} P'$ *then* $\mathcal{O} \xrightarrow{\tau}{}^{*} \mathcal{O}$ *and* $\mathcal{O}\,\mathcal{R}\,P'$

The largest relation $\mathcal{R}$ is said the weak bisimilarity and it is written $\approx$.

All rules but the fourth one are quite standard. In fact, rule 4 handles the case in which a Join process performs an intrusion on an already extruded name. The only possibility is when the process receives an answer for a site call. Hence, such step should be mimicked by the orchestrator (i.e., the condition $\mathcal{O} \xrightarrow{k?v} \mathcal{O}'$). Nevertheless, this situation may take place only when the first answer is received. In fact, the Join encoding of an Orc site call ignores all the answers following the first one. On the other end, the open semantics of Join allows for the intrusion of those messages (even if they cannot be exploited). Hence, the weak bisimulation says that the intrusion of extra answers does not change the behaviour of the encoded form (i.e., $\mathcal{O}\not\xrightarrow{k?v}$ and $\mathcal{O}\ \mathcal{R}\ P'$).

In the following, we show that there exists a weak bisimulation among Orc orchestrators and their encoded form when considering a non-killing version of Orc, that is, a version in which asymmetric composition does not imply the killing of the residual of f. In fact we consider the following version of the rule (ASYMPRUNE).

$$\frac{f \xrightarrow{!c} f'}{g \textbf{ where } x :\in f \xrightarrow{\tau} g[c/x] \mid (0 \textbf{ where } z :\in f')} \ (\text{NOTKILL-WHERE})$$

Note that g is evaluated as in ordinary Orc just for one value produced by f. Nevertheless, the residual f' of f is allowed to continue its execution, but the obtained values are thrown away since 0 appears as the left-hand-side of the clause **where**. We remark that (NOTKILL-WHERE) does not significantly alters Orc's semantics and it can be envisaged as an implementation of the g **where** $x :\in f$ construct that simply ignores all values published by f but the first one.

Lemma 4.1 (Correspondence). *When considering rule* NOTKILL-WHERE*,* $\mathcal{O} \approx \{[\mathcal{O}]\}$.

Proof (Sketch). The proof follows by coinduction, showing that the following relation R is a weak bisimulation.

$$R = \{(\mathcal{O},P) | \{[\mathcal{O}]\} \xrightarrow{\tau} {}^*P\} \cup \{(\mathcal{O}',P') | \mathcal{O} \xrightarrow{\alpha} \mathcal{O}' and \{[\mathcal{O}]\} \xrightarrow{\tau} {}^* \xrightarrow{\overline{\alpha}} \xrightarrow{\tau} {}^*P'\}$$

Actually the proof is up-to strong-bisimulation [15] on Join processes, since we consider terms up-to the relation $\equiv_e$ defined below

1. if $P \rightleftharpoons^* Q$ then $P \equiv_e Q$, i.e., P and Q are structural equivalent;
2. $P \equiv_e P|\textbf{def } D \textbf{ in } 0$, i.e., useless definitions are removed; and
3. if $Q \equiv \textbf{def}_S\ D \textbf{ in } \textbf{def}_{\{k\}}\ k\langle\vec{v}\rangle|tok\langle\rangle \triangleright z\langle v\rangle \textbf{ in } R \mid k\langle\vec{u}\rangle \rightarrow^* Q'$ implies $Q' \equiv \textbf{def}_S\ D' \textbf{ in } \textbf{def}_{\{k\}}\ k\langle\vec{v}\rangle|tok\langle\rangle \triangleright z\langle\vec{v}\rangle \textbf{ in } R' \mid k\langle\vec{u}\rangle$ and $P \equiv \textbf{def}_S\ D \textbf{ in } R \rightarrow^* \textbf{def}_S\ D' \textbf{ in } R'$ and $tok \notin fn(R)$, then $P \equiv_e Q$, i.e., intruded messages that do not alter the behaviour of the process can be removed.

Note that $\equiv_e$ is a strong bisimulation (proved by standard coinduction).

Finally, we show that the computed values of ordinary Orc orchestrators corresponds with the computed values of their encoded form.

Theorem 4.1. $\mathcal{O} \rightarrow^* \xrightarrow{!v} \mathcal{O}'$ *iff* $\{[\mathcal{O}]\} \rightarrow^* \xrightarrow{z\langle v\rangle} P$

Proof (Sketch). The proof follows by (i) showing that the results computed by Orc and its not killing version are the same and (ii) by using Lemma 4.1.

5 Concluding remarks

Orchestration paradigms can be roughly categorised into three key trends:

- technology-driven languages: all XML dialects and standardisation efforts (e.g., WS-BPEL [8], XLANG [27], WSFL [16]);
- model oriented: workflow aspects are prominent (e.g., Petri nets [24,3], YAWL [4]);
- process algebraic or messaging-based: the orchestration is ruled by communication primitives (e.g., CCS [17], pi-calculus [18], and Join calculus [14]).

A few years ago, when the series of WS-FM Workshop started, each trend contained several proposals substantially separated from the other two trends, with different background, scope and applications. For example, a still ongoing debate [25,1,2] adverses the use of workflow to that of pi-calculus and it has led to the establishment of an expert forum (the Process Modelling Group [22]) to investigate how the two different approaches can solve typical service composition challenges, like van der Aalst et al.'s *workflow patterns* [28,5], and compare the solutions. Workflow enthusiasts advocate that name mobility and message passing are not really necessary, while pi-calculus enthusiasts are confident that mobility aspects play a prominent role in dynamic assembling of services. The discussion has led also to the combined use of ideas from both world, like in the case of SMAWL [26], a CCS variant.

We have investigated the modelling of the orchestration language Orc in Petri nets and the Join calculus. Orc is an interesting proposal that can hardly fit in the orchestration categories discussed above. Our comparisons have allowed us to identify some key features of Orc, that are not so evident from its original definition. First, pipelining, site calls and asymmetric parallel composition involve dynamic creation of names and links, that cannot find a natural encoding in Petri nets with static topology, unless seriously restricting Orc. Second, the pruning associated with asymmetric conflict is a rather peculiar and powerful operation not common in process calculi. In fact, one can argue that it is also not very realistic to impose atomic cancelling of complex activities in a distributed setting (especially when side effects due to e.g. name passing and extrusion could have taken place). Nevertheless, from the point of view of process calculi, cancelling can be rendered as equivalent to the disabling of the input ports where the cancelled activities could send their data. In Petri nets and Join the disabling is modelled by void tokens that enable just one occurrence of certain events, but Join has the advantage of not introducing cleaning activities and serialisation of site calls, which are instead necessary for dealing with multiple invocations in the Petri net encodings of § 3.

Finally, we mention that Join appears to be adequate as coordination language since it can suitably encode Orc. Remarkably, Join, despite its thinness, also results a respectable language for choreography and computing. Finally, Join is perhaps also more suitable as coordination/orchestration language than e.g. pi-calculus because its join-pattern construct yields more flexible and convenient communication patterns.

Acknowledgement. The authors thank the anonymous reviewers for their valuable comments and suggestions that contributed to improve this work.

References

1. W.M.P. van der Aalst. Why workflow is NOT just a pi process. *BPTrends*, pages 1–2, 2004.
2. W.M.P. van der Aalst. Pi calculus versus Petri nets. *BPTrends*, pages 1–11, 2005.
3. W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow patterns: On the expressive power of (Petri-net-based) workflow languages. *Proc. of CPN'02*, volume 560 of *DAIMI*, pages 1–20. University of Aarhus, 2002.
4. W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
6. T. Araki and T. Kasami. Some decision problems related to the reachability problem for Petri nets. *TCS*, 3(1):85–104, 1976.
7. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for $C^\sharp$. *Proc. of ECOOP'02*, volume 2374 of *LNCS*, pages 415–440. Springer, 2002.
8. BPEL Specification (v. 1.1). `http://www.ibm.com/developerworks/library/ws-bpel`.
9. M. Buscemi and V. Sassone. High-level Petri nets as type theories in the join calculus. *Proc. of FoSSaCS'01*, volume 2030 of *LNCS*, pages 104–120. Springer, 2001.
10. S. Conchon and F. Le Fessant. Jocaml: Mobile agents for Objective-Caml. *Proc. of ASA/MA'99*, pages 22–29. IEEE Computer Society, 1999.
11. W.R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc, 2006. Submitted.
12. C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. *Proc. of ICALP'98*, volume 1443 of *LNCS*, pages 103–115. Springer, 1998.
13. M. J. Flynn and T. Agerwala. Comments on capabilities, limitations and correctness of Petri nets. *SIGARCH Computer Architecture News*, pages 81–86, 1973.
14. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join calculus. *Proc. of POPL'96*, pages 372–385. ACM Press, 1996.
15. C. Fournet and C. Laneve. Bisimulations in the join calculus. *TCS*, 266:569–603, 2001.
16. F. Leymann. WSFL Specification (v. 1.0). `http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.
17. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.
18. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992.
19. J. Misra and W. R. Cook. Orc - An orchestration language. `http://www.cs.utexas.edu/~wcook/projects/orc/`.
20. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 2006. To appear.
21. C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.
22. The Process Modelling Group web site. `http://www.process-modelling-group.org/`.
23. F. Puhlmann and M. Weske. Using the pi-calculus for formalising workflow patterns. *Proc. of BPM'05*, volume 3649 of *LNCS*, pages 153–168. Springer, 2005.
24. W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1985.
25. H. Smith and P. Fingar. Workflow is just a pi process. *BPTrends*, pages 1–36, 2004.
26. C. Stefansen. SMAWL: A small workflow language based on CCS. *CAiSE'05 Short Paper Proceedings*, volume 161 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.
27. S. Thatte. XLANG: Web Services for Business Process Design. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`, 2001.
28. Workflow Patterns web site. `http://is.tm.tue.nl/research/patterns/`.