Orchestrating Tuple-based Languages

Rocco De Nicola[1], Andrea Margheri[1], and Francesco Tiezzi[2]

[1] Univeristà degli Studi di Firenze, Dipartimento di Sistemi e Informatica
Viale Morgagni, 65 – 50134 – Firenze, Italy

[2] IMT - Institutions, Markets and Technologies, Institute for Advanced Studies Lucca
Piazza S. Ponziano, 6 – 55100 – Lucca, Italy

rocco.denicola@unifi.it, margheri.a@alice.it,
francesco.tiezzi@imtlucca.it

Technical report — June 10, 2011

Abstract. The World Wide Web can be thought of as a global computing architecture supporting the deployment of distributed networked applications. Currently, such applications can be programmed by resorting mainly to two distinct paradigms: one devised for orchestrating distributed services, and the other designed for coordinating distributed (possibly mobile) agents. In this paper, the issue of designing a programming language aiming at reconciling orchestration and coordination is investigated. Taking as starting point the orchestration calculus Orc and the tuple-based coordination language Klaim, a new formalism is introduced combining concepts and primitives of the original calculi. To demonstrate feasibility and effectiveness of the proposed approach, a prototype implementation of the new formalism is described and it is then used to tackle a case study dealing with a simplified but realistic electronic marketplace, where a number of on-line stores allow client applications to access information about their goods and to place orders.

Keywords: Global computing, Orchestration, Coordination, Tuple-based languages, Formal methods, Software tools

1 Introduction

In recent years, the increasing success of e-business, e-learning, e-government, and similar emerging models, has led the World Wide Web, initially thought of as a tool supporting humans in looking for information, to evolve towards a service-oriented architecture, where more and more distributed networked applications, the so-called *services*, are deployed. This has promoted the rising of a novel programming paradigm for the *orchestration* of concurrent and distributed services. There are by now some successful and well-developed technologies supporting this paradigm, like e.g. WS-BPEL[33], the standard language for orchestration of web services. However, current software engineering technologies remain at the descriptive level and lack rigorous formal foundations. Hence, many researchers

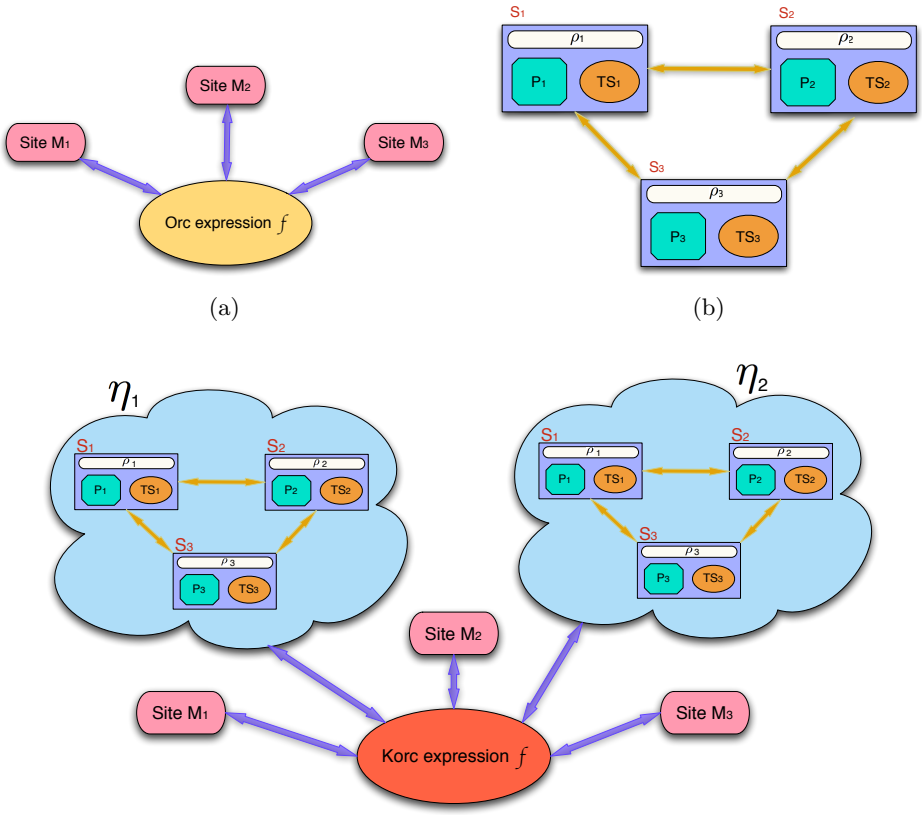

Fig. 1. The ORC (a), KLAIM (b) and KORC (c) approaches

have tackled the problem at a more foundational level, by developing formal languages for designing and programming service orchestrations.

Among the many proposed formalisms (see, e.g., [28, 10, 34, 11, 27, 23, 8, 12, 9]), we will focus on ORC [30, 25, 37], a task orchestration language with applications in workflow, business process management, and web service orchestration. ORC is the result of a tension between simplicity and expressiveness, and its primitives focus on orchestration rather than on communication such as those of most formal languages. An ORC program, graphically depicted in Figure 1(a), is an *expression* that orchestrates the concurrent invocations of a number of services, called *sites* in the ORC's jargon, by means of three operators for sequential and parallel composition.

Although the small numbers of ORC's operators have been proved to be sufficiently expressive to model the most common orchestration patterns (see e.g. those identified in [35]), they do not provide adequate and flexible mechanisms for distributed *coordination*, which may possibly refer and exploit the structures

of the network. In this respect, it has been proved that *tuple-based* languages can be effectively used to implement coordination mechanism in a distributed setting. In particular, here we focus on KLAIM [15, 7, 16], a coordination language specifically designed to program distributed systems consisting of several mobile components that interact through multiple distributed tuple spaces. KLAIM components refer and control the spatial structures of the network at any point of their evolution (i.e. they are *network-aware*). KLAIM's communication model builds over, and extends, Linda's notion of generative communication through a single shared tuple space [21]. KLAIM primitives allow programmers to distribute and retrieve data and processes to and from the (tuple spaces of the) nodes of a net. Moreover, localities are first-class citizens that can be dynamically created and communicated over the network and can be handled via sophisticated scoping rules. A KLAIM specification, graphically depicted in Figure 1(b), can be thought of as a net of interconnected nodes, each of which hosts data tuples and (possibly mobile) processes, and is identified by an unique name.

In this paper, we investigate the issue of designing a programming language aiming at reconciling the orchestration paradigm with the tuple-based coordination one. We tackle the problem first at foundational level, by defining a new formalism, called KORC, that combines the composition patterns and primitives of ORC and KLAIM. Intuitively, a KORC program, graphically depicted in Figure 1(c), consists of an ORC expression and a collection of KLAIM nets, where expressions are extended with primitives for acting on the tuple spaces within the KLAIM nets. To distinguish nets from each other, any net has a name used to refer to it from a KORC expression.

The choice of using ORC and KLAIM has theoretical basis for KORC has been mainly motivated by the fact that they are compact formalisms and, moreover, are already equipped with software tools for programming networked applications. Such tools are both Java-based and, hence, easily integrable. In fact, to demonstrate feasibility and effectiveness of the programming paradigm fostered by KORC and to experiment with it, as a second contribution of this paper we have developed a prototype implementation of the language that build upon the implementations of ORC and KLAIM.

The rest of the paper is structured as follows. Section 2 presents the design and the formal definition of KORC, by exploiting concepts and definitions of ORC and KLAIM. Section 3 introduces an e-commerce scenario used for illustrating the relevant and peculiar aspects of KORC. Section 4 provides an overview of the prototype implementation of KORC and describes an excerpt of the e-commerce scenario written in the syntax accepted by the tool. Finally, Section 5 draws a few conclusions and reviews some strictly related work.

2 From Orc and Klaim to Korc

In this section, we first recap the basic notions of ORC and KLAIM, by borrowing syntax and semantics definitions from [37] and [16]. Then, we exploit such linguistic bases to define our formalism KORC.

(Expressions)	f, g	$::=$	$M(\bar{p})$	$\mid$	$E(\bar{p})$	$\mid$	$f > \bar{p} > g$	$\mid$	$f \mid g$	$\mid$	$f < \bar{p} < g$
(Parameters)	p	$::=$	x	$\mid$	m						

Table 1. Orc syntax

2.1 Orc: an orchestration language

An Orc program consists of a goal *expression* and a set of *definitions*; the goal expression is evaluated in order to run the program. The definitions are used in the expression and in other definitions. Formally, the Orc syntax is defined in Table 1, where M ranges over *site* names, E over *expression* names, x over variables, and m over values. It is assumed that the sets of site names, expression names, variables and values are countable and pairwise disjoint. Notation $\bar{}$ stands for tuples of parameters, e.g. $\bar{m}$ is a compact notation for denoting the tuple of values $\langle m_1, \ldots, m_n \rangle$ (with $n \geq 0$); variables in the same tuple are pairwise distinct. The empty tuple, written $\langle\rangle$, corresponds to a *signal*, the Orc unit value that has no additional information. We shall write $a, \bar{b}$ to denote the tuple obtained by concatenating the element a to the tuple $\bar{b}$.

Expressions can be composed by means of sequential composition $\cdot > \bar{p} > \cdot$, symmetric parallel composition $\cdot \mid \cdot$, and asymmetric parallel composition $\cdot < \bar{p} < \cdot$, starting from the elementary expressions $M(\bar{p})$, i.e. site calls, and $E(\bar{p})$, i.e. expression calls. The variables within $\bar{p}$ are *bound* in g for the expressions $f > \bar{p} > g$ and $g < \bar{p} < f$. We use $\mathrm{fv}(f)$ to denote the set of variables that are not bound (i.e. which occur *free*) in f. Each expression name E has a unique declaration of the form $E(\bar{x}) \triangleq f$, where only the variables $\bar{x}$ are free in f, i.e. $\bar{x} = \mathrm{fv}(f)$.

It is worth noticing that, since we aim at merging Orc with a coordination language exploiting a tuple-based communication mechanism, we consider a polyadic variant of Orc. Thus, site calls, expression calls, sequential composition and asymmetric parallel composition take as argument tuples of parameters rather than single parameters. For the best of our knowledge, this is the first formalization of this (standard) extension of the calculus (which is only informally described in [26]), hence it can be thought of as a minor contribution of this paper.

Evaluation of expressions may call a number of sites and returns a (possibly empty) stream of (tuple of) values. So, in summary, an Orc expression can be either a site call, an expression call or a composition of expressions according to one of the three basic orchestration patterns.

Site call: a site call can have the form $M(\bar{p})$, where the site name is known statically, and $\bar{p}$ are the parameters of the call. A site call returns at most one response and, hence, a site can also never respond to a call. If $\bar{p}$ contains variables, then they must be instantiated before the call is made (i.e. site calls are *strict*).

Expression call: an expression call has the form $E(\bar{p})$ and executes the expression defined by $E(\bar{x}) \triangleq f$ after having replaced $\bar{x}$ by $\bar{p}$ (of course, the

tuples $\bar{x}$ and $\bar{p}$ must have the same length). Here $\bar{p}$ is passed by reference. Note that expression definitions can be recursive.

Symmetric parallel composition: the composition $f \mid g$ executes both f and g concurrently, assuming that there is no interaction between them. It publishes the interleaving of the two streams of tuples published by f and g, in temporal order.

Sequential composition: the composition $f > \bar{p} > g$ executes f, and, for each tuple of values $\bar{m}$ returned by f, it checks if $\bar{p}$ and $\bar{m}$ match: in the positive case, it executes an instance of g with variables in $\bar{p}$ replaced by the corresponding values in $\bar{m}$, otherwise the publication is ignored and no new instance of g is executed. The composition publishes the interleaving (in temporal order) of the streams of tuples published by the different instances of g (tuples published by f are consumed within $f > \bar{p} > g$).

Asymmetric parallel composition: the composition $g < \bar{p} < f$ starts in parallel both f and the parts of g that do not need the variables in $\bar{p}$. When f publishes a tuple, let say $\bar{m}$, the matching between $\bar{p}$ and $\bar{m}$ is checked: in the positive case, the evaluation of f is immediately terminated and the variables within $\bar{p}$ are replaced by the corresponding values in $\bar{m}$. (in this way, the suspended parts of g can proceed). The composition publishes the stream obtained from g (instantiated with values in $\bar{m}$).

Example 1. Consider a service orchestration that, given a US zip code, first contacts a service for getting the current weather conditions and the temperature in Fahrenheit degrees of the city corresponding to the zip code and, then, contacts a second service for converting Fahrenheit degrees into Celsius ones.

The corresponding Orc specification is as follows:

$$\begin{aligned} CelsiusTemperature(x_{zip}) \triangleq\ & Weather(x_{zip}) \\ & > \langle x_{city}, x_{weather}, x_{temperature} \rangle > \\ & FahrenheitToCelsius(x_{temperature}) \end{aligned}$$

Now, the evaluation of the Orc expression $CelsiusTemperature(10109)$ involves first a call to expression $CelsiusTemperature$ with argument 10109, i.e. the zip code of an area of New York, then a call to site $Weather$ with the same argument and, subsequently, a call to site $FahrenheitToCelsius$ with the temperature value returned by the first call as argument; the expression finally publishes the current temperature of New York converted into Celsius degrees.

The asynchronous operational semantics of Orc[3] is given in terms of a labelled transition relation and, to represent intermediate states in service interactions, is defined over a syntax enriched with auxiliary terms: expressions are

[3] It is worth noticing that the operational semantics of Orc we present here is obtained from the timed semantics introduced in [37] by discharging the timed aspects and adding tuples. We have chosen to rely on the syntax and operational semantics defined in [37], rather than the untimed ones described in [30, 25], only for the sake of update. Indeed, the formal presentation of the calculus in [37] is the latest one and is that used as theoretical base for the implementation of the Orc programming language [26].

$$\mathcal{M}(x,m) = [m/x] \qquad \mathcal{M}(m,m) = \epsilon \qquad \mathcal{M}(\langle\rangle,\langle\rangle) = \epsilon$$

$$\frac{\mathcal{M}(p_1,m_1) = \sigma_1 \quad \mathcal{M}(\bar{p}_2,\bar{m}_2) = \sigma_2}{\mathcal{M}((p_1,\bar{p}_2),(m_1,\bar{m}_2)) = \sigma_1 \circ \sigma_2}$$

Table 2. ORC matching rules

extended with the elementary expression $\mathbf{0}$, which has no observable transitions, and the intermediate expression $?k$, which denotes an instance of a site call that has not yet returned a response. Intuitively, k is an *handle* that describes a possible behavior of a site when it is called with a tuple of values. Formally, an handle is a set of tuples of values that can be returned by a site call as a response. We write $\Sigma(M,\bar{m})$ for the set of handles that correspond to expression $M(\bar{m})$.

To define the labelled transition relation, we need an auxiliary function $\mathcal{M}(\cdot\,,\cdot)$ for performing *pattern-matching* on semi-structured data. The rules defining $\mathcal{M}(\cdot\,,\cdot)$ are shown in Table 2. They state that two tuples match if they have the same number of fields and corresponding fields have matching values/variables. Variables match any value, and two values match only if they are identical. When tuples $\bar{p}$ and $\bar{m}$ do match, $\mathcal{M}(\bar{p},\bar{m})$ returns a substitution for the variables in $\bar{p}$; otherwise, it is *undefined*. *Substitutions* (ranged over by σ) are functions mapping variables to values and are written $[\bar{m}/\bar{x}]$. Application of substitution $[\bar{m}/\bar{x}]$to an expression f, written $[\bar{m}/\bar{x}].f$, has the effect of replacing every free occurrence of variables $\bar{x}$ in f with the corresponding values in $\bar{m}$. We use ϵ to denote the empty substitution (i.e. a substitution $[\bar{m}/\bar{x}]$ where tuples $\bar{m}$ and $\bar{x}$ have length 0), and $\sigma_1 \circ \sigma_2$ to denote the union of σ_1 and σ_2 when they have disjoint domains. With abuse of notation, we shall use $[\bar{p}/\bar{x}].f$ to indicate the replacing of variables $\bar{x}$ in f by parameters $\bar{p}$.

The labelled transition relation $\xrightarrow{a}$ is the least relation over expressions induced by the rules in Table 3, where label a is generated by the following grammar:

$$a \quad ::= \quad \tau \quad \mid \quad !\bar{m}$$

The meaning of labels is as follows: label τ indicates an *internal event* while label $!\bar{m}$ indicates a *publication event* corresponding to the communication to the environment of the tuple of values $\bar{m}$ resulting from the evaluation of an expression.

Let us now comment on the operational rules. A site call can progress only when the actual parameter is a tuple of values $\bar{m}$ (rule *(SiteCall)*); it evolves to an intermediate expression $?k$, where k is an handle for the site call $M(\bar{m})$. The expression $?k$ can then evolve to $\mathbf{0}$ by eliciting one response $\bar{m}$ (rule *(Return)*), with $\bar{m}$ possible response included in k. While site calls use a call-by-value mechanism, expression calls use a call-by-name mechanism (rule *(Def)*), namely the actual parameters replace the formal ones and then the corresponding expression is evaluated. We assume a global set of definitions. Symmetric parallel composition $f \mid g$ consists of concurrent evaluations of f and g (rules *(Sym1)* and *(Sym2)*). Sequential composition $f > \bar{p} > g$ allows f to evolve by performing

$$\frac{k \in \Sigma(M, \bar{m})}{M(\bar{m}) \xrightarrow{\tau} ?k} \textit{ (Call)} \qquad \frac{\bar{m} \in k}{?k \xrightarrow{!\bar{m}} \mathbf{0}} \textit{ (Return)}$$

$$\frac{f \xrightarrow{a} f'}{f \mid g \xrightarrow{a} f' \mid g} \textit{ (Sym1)} \qquad \frac{g \xrightarrow{a} g'}{f \mid g \xrightarrow{a} f \mid g'} \textit{ (Sym2)}$$

$$\frac{f \xrightarrow{\tau} f'}{f > \bar{p} > g \xrightarrow{\tau} f' > \bar{p} > g} \textit{ (Seq1)} \qquad \frac{f \xrightarrow{!\bar{m}} f' \quad \mathcal{M}(\bar{p}, \bar{m}) = \sigma}{f > \bar{p} > g \xrightarrow{\tau} (f' > \bar{p} > g) \mid \sigma.g} \textit{ (Seq2)}$$

$$\frac{E(\bar{x}) \triangleq f}{E(\bar{p}) \xrightarrow{\tau} [\bar{p}/\bar{x}].f} \textit{ (Def)} \qquad \frac{f \xrightarrow{!\bar{m}} f' \quad \mathcal{M}(\bar{p}, \bar{m})\ undefined}{f > \bar{p} > g \xrightarrow{\tau} f' > \bar{p} > g} \textit{ (Seq3)}$$

$$\frac{f \xrightarrow{a} f'}{f < \bar{p} < g \xrightarrow{a} f' < \bar{p} < g} \textit{ (Asym1)} \qquad \frac{g \xrightarrow{\tau} g'}{f < \bar{p} < g \xrightarrow{\tau} f < \bar{p} < g'} \textit{ (Asym2)}$$

$$\frac{g \xrightarrow{!\bar{m}} g' \quad \mathcal{M}(\bar{p}, \bar{m}) = \sigma}{f < \bar{p} < g \xrightarrow{\tau} \sigma.f} \textit{ (Asym3)} \qquad \frac{g \xrightarrow{!\bar{m}} g' \quad \mathcal{M}(\bar{p}, \bar{m})\ undefined}{f < \bar{p} < g \xrightarrow{\tau} f < \bar{p} < g'} \textit{ (Asym4)}$$

Table 3. ORC asynchronous operational semantics

internal transition (rule *(Seq1)*) and, for each matching tuple $\bar{m}$ returned by f, activates a concurrent copy of g instantiated with the substitution generated by the matching function (rule *(Seq2)*). If f publishes a non-matching tuple, the publication is ignored (rule *(Seq3)*). Asymmetric parallel composition $f < \bar{p} < g$ prunes threads selectively. It starts in parallel both g and the part of f that does not need variables within $\bar{p}$ (rules *(Asym1)* and *(Asym2)*). When the first matching tuple is returned by g, the generated substitution is applied to f and the continuation of g and all its descendants are then terminated (rule *(Asym3)*). If g publishes a non-matching tuple, the publication is ignored and g continues to run (rule *(Asym4)*).

2.2 Klaim: a tuple-based language for agent interaction

KLAIM is a formal language equipped with primitives for network-aware programming that combines the process algebra approach with the coordination-oriented one. The syntax of KLAIM is reported in Table 4, where:

- s, s',... range over *locality names*, i.e. network addresses;
- **self**, l, l',... range over *locality variables*, i.e. aliases for addresses;
- ℓ, ℓ',... range over locality names and variables;
- x, y,... range over *value variables*;

(Nets)	N	$::=$	$\mathbf{0} \quad \mid \quad s ::_\rho C \quad \mid \quad N_1 \parallel N_2 \quad \mid \quad (\nu s)N$
(Components)	C	$::=$	$\langle t \rangle \quad \mid \quad P \quad \mid \quad C_1 \mid C_2$
(Processes)	P	$::=$	$\mathbf{nil} \quad \mid \quad \alpha.P \quad \mid \quad P_1 \mid P_2 \quad \mid \quad A(\bar{p})$
(Actions)	α	$::=$	$\mathbf{out}(t)@\ell \quad \mid \quad \mathbf{eval}(P)@\ell$
		$\mid$	$\mathbf{in}(T)@\ell \quad \mid \quad \mathbf{read}(T)@\ell \quad \mid \quad \mathbf{newloc}(s)$
(Tuples)	t	$::=$	$e \quad \mid \quad \ell \quad \mid \quad P \quad \mid \quad t_1, t_2$
(Templates)	T	$::=$	$e \quad \mid \quad \ell \quad \mid \quad !x \quad \mid \quad !l \quad \mid \quad !X \quad \mid \quad T_1, T_2$

Table 4. KLAIM syntax

- X, Y,... range over *process variables*;
- e, e',... range over *expressions*; the precise syntax of expressions is deliberately not specified, but we assume that they contain, at least, basic values (ranged over by v, v',... and whose set is left unspecified too) and variables;
- A, B,... range over *process identifiers*; we assume that each process identifier A with arity n has an unique definition available at any locality of a net of the form $A(f_1, \ldots, f_n) \triangleq P$, where f_i are pairwise distinct ($\bar{p}$ and $\bar{f}$ denote tuples of actual and formal parameters, respectively, consisting of values and variables.)

We assume that the set of variables (i.e. locality, value and process variables) the set of values (locality names and basic values) and the set of process identifiers are countable and pairwise disjoint.

Nets are finite plain collections of nodes where *components* (i.e. evaluated tuples and processes) can be allocated. A *node* is a triple $s ::_\rho C$, where locality s is the address of the node, ρ is the an allocation environment and C are the host components. An *allocation environment* binds the locality variables occurring free in the processes allocated in the corresponding node. Basically, allocation environments provide a name resolution mechanism by mapping locality variables l into localities s. The distinguished locality variable **self** is used by processes to refer to the address of their current hosting node. In the net $(\nu s)N$, the scope of the name s is restricted to N; the intended effect is that if one considers the net $N_1 \parallel (\nu s)N_2$ then locality s of N_2 cannot be immediately referred to from within N_1.

Processes are the KLAIM active computational units that are built up from the special process **nil**, that does not perform any action, and from the basic operations by means of action prefixing $\alpha.P$, parallel composition $P_1 \mid P_2$ and process definition. Process may be executed concurrently either at the same locality or at different localities and can perform five different basic operations, called actions.

Actions **out**, **in** and **read** manage data repositories by adding/withdrawing/accessing data to/from node repositories. Action **eval** activates a new thread of execution, i.e. a process, in a (possibly remote) node. Action **newloc** permits creation of new network nodes. The latter action is not indexed with an address because it always acts locally; all other actions indicate explicitly the (possibly remote) locality where they will take place. **in** and **read** are (possibly) blocking actions and exploit templates as patterns to select data in shared repositories. *Templates* are sequences of actual and formal fields, where the latter are written $!\,x$, $!\,l$ or $!\,X$ and are used to bind variables to basic values, locality names or processes, respectively. **out** and **eval** are non-blocking actions and implement *static* and *dynamic scoping* disciplines, respectively (see below).

A net is *well-formed* if for each node $s ::_\rho C$, we have that $\rho(\mathbf{self}) = s$, and, for any pair of nodes $s ::_\rho C$ and $s' ::_{\rho'} C'$, we have that $s = s'$ implies $\rho = \rho'$. Hereafter,we will only consider well-formed nets.

Names and variables occurring in KLAIM processes and nets can be *bound*. More precisely, prefix $\mathbf{newloc}(s).P$ binds name s in P, and, similarly, net restriction $(\nu s)N$ binds s in N. The sets $fn(\cdot)$ and $bn(\cdot)$ of, respectively, free and bound locality names of a term are defined accordingly. Prefixes $\mathbf{in}(\ldots,!\_,\ldots)@\ell.P$ and $\mathbf{read}(\ldots,!\_,\ldots)@\ell.P$ binds variable $\_$ in P. A name/variable that is not bound is called *free*. As usual, we say that two terms are alpha-equivalent, written $=_\alpha$, if one can be obtained from the other by renaming bound names/variables. In the sequel, we shall work with terms whose bound variables are all distinct and whose bound names are all distinct and different from the free ones.

Example 2. Consider an application that converts a list of some famous US places into the corresponding zip code by filtering out the places that are not located in the State of New York[4].

The KLAIM net corresponding to this scenario is as follows:

$$\begin{array}{rl} s_{zip} ::_{\{\mathbf{self}\mapsto s_{zip}\}} & \ldots \mid \langle 10109, \text{“}Times\ Square\text{”}\rangle \mid \ldots \mid \langle 10451, \text{“}Bronx\text{”}\rangle \mid \ldots \\ & \mid \langle 20500, \text{“}White\ House\text{”}\rangle \mid \ldots \mid \langle 89144, \text{“}Las\ Vegas\text{”}\rangle \mid \ldots \\ \parallel\ s_{places} ::_{\{\mathbf{self}\mapsto s_{places}, l_{zip}\mapsto s_{zip}\}} & Filter(10001, 14905) \mid \langle \text{“}Times\ Square\text{”}\rangle \\ & \mid \langle \text{“}White\ House\text{”}\rangle \mid \langle \text{“}Bronx\text{”}\rangle \mid \ldots \end{array}$$

where locality s_{zip} represents a US zip codes database, locality s_{places} contains a list of places, and $Filter$ is a process defined as follows:

$$\begin{array}{l} Filter(zip_{min}, zip_{max}) \triangleq \\ \mathbf{in}(!place)@\mathbf{self}.\,\mathbf{read}(!zip, place)@l_{zip}. \\ \quad (\,\mathbf{if}\ (zip_{min} \leqslant zip \leqslant zip_{max})\ \mathbf{then}\ \mathbf{out}(zip)@\mathbf{self}.\,\mathbf{nil} \\ \quad \mid Filter(zip_{min}, zip_{max})\,) \end{array}$$

Basically, the above process cyclically reads (and consumes) the name of a place locally and uses it to retrieve the corresponding zip code from s_{zip}, then checks if

[4] The geographical area of the State of New York is identified by the set of corresponding zip codes, which range from 10001 to 14905.

(Monoid) $N \parallel \mathbf{0} \equiv N$	$N_1 \parallel N_2 \equiv N_2 \parallel N_1$ $\quad (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$
(RCom) $(\nu s_1)(\nu s_2)N \equiv (\nu s_2)(\nu s_1)N$	*(PDef)* $s ::_\rho A(\bar{p}) \equiv s ::_\rho P[\bar{p}/\bar{f}]$ if $A(\bar{f}) \triangleq P$
(Ext) $N_1 \parallel (\nu s)N_2 \equiv (\nu s)(N_1 \parallel N_2)$ if $s \notin fn(N_1)$	*(Alpha)* $N \equiv N'$ if $N =_\alpha N'$
(Abs) $s ::_\rho C \equiv s ::_\rho (C \mid \mathbf{nil})$	*(Clone)* $s ::_\rho C_1 \mid C_2 \equiv s ::_\rho C_1 \parallel s ::_\rho C_2$

Table 5. KLAIM structural congruence

the zip code is within the given range and, in the positive case, produces a tuple containing the zip code. For the sake of simplicity, in defining the *Filter* process we have used a conditional construct, which can be easily programmed in KLAIM by exploiting the dynamic creation of new nodes and the parallel composition operator.

The operational semantics of KLAIM is given in terms of a structural congruence relation and a reduction relation expressing the evolution of a net. The structural congruence $\equiv$ identifies syntactically different terms that intuitively represent the same term. It is defined as the least congruence closed under the equational laws shown in Table 5. Most of the laws are standard, while laws *(Abs)* and *(Clone)* are peculiar to this setting. The first one states that **nil** is the identity for $\cdot \mid \cdot$; the second one turns a parallel between co-located components into a parallel between nodes (thus, it is also used to achieve commutativity and associativity of $\cdot \mid \cdot$).

To define the reduction relation, we need two auxiliary functions. First, we exploit a function $\mathcal{E}[\![\_]\!]_\rho$ for *evaluating tuples/templates*, which evaluates the expressions within $\_$ and, in case of success, returns an evaluated tuple/template. In particular, $\mathcal{E}[\![\_]\!]_\rho$ replaces the free locality variable occurring in $\_$ according to the allocation environment ρ of the node performing the action whose argument is $\_$. Notice that the free occurrences of variables within the argument of **eval** actions are not involved by such replacement. However, $\mathcal{E}[\![\_]\!]_\rho$ cannot be explicitly defined because the exact syntax of expressions is deliberately not specified.

Then, through the rules in Table 6, we define a *pattern-matching* function $match(\cdot, \cdot)$, to verify the compliance of a tuple w.r.t. a template and to associate basic values, locality names and processes to variables bound in templates. Intuitively, a tuple matches against a template if they have the same number of fields, and corresponding fields match (where a bound name matches any value, while two names match only if they are identical). Notably, differently from ORC where constructs $\cdot > \bar{p} > \cdot$ and $\cdot < \bar{p} < \cdot$ bind all variables within $\bar{p}$, in KLAIM variables can be either bound or not (in the former case they are preceded by the symbol '!'). Thus, bound variables match any (corresponding) value, while the matching always fails for free variables.

The reduction relation is given in Table 7. The intuition beyond the operational rules of KLAIM is the following. In rule *(Out)*, the local allocation environment ρ of the node performing the action $\mathbf{out}(t)@\ell$ is used both to evaluate ℓ, in

$$match(!x, v) = [v/x] \qquad match(!l, s) = [s/l] \qquad match(!X, P) = [P/X]$$

$$match(v, v) = \epsilon \qquad \frac{match(T_1, t_1) = \sigma_1 \qquad match(T_2, t_2) = \sigma_2}{match((T_1, T_2), (t_1, t_2)) = \sigma_1 \circ \sigma_2}$$

$$match(s, s) = \epsilon$$

Table 6. KLAIM matching rules

$$\frac{\rho(\ell) = s' \qquad \mathcal{E}[\![\, t \,]\!]_\rho = t'}{s ::_\rho \mathbf{out}(t)@\ell.P \;\parallel\; s' ::_{\rho'} \mathbf{nil} \longmapsto s ::_\rho P \parallel s' ::_{\rho'} \langle t' \rangle} \; (Out)$$

$$\frac{\rho(\ell) = s'}{s ::_\rho \mathbf{eval}(Q)@\ell.P \;\parallel\; s' ::_{\rho'} \mathbf{nil} \longmapsto s ::_\rho P \parallel s' ::_{\rho'} Q} \; (Eval)$$

$$\frac{\rho(\ell) = s' \qquad match(\mathcal{E}[\![\, T \,]\!]_\rho, t) = \sigma}{s ::_\rho \mathbf{in}(T)@\ell.P \;\parallel\; s' ::_{\rho'} \langle t \rangle \longmapsto s ::_\rho P\sigma \parallel s' ::_{\rho'} \mathbf{nil}} \; (In)$$

$$\frac{\rho(\ell) = s' \qquad match(\mathcal{E}[\![\, T \,]\!]_\rho, t) = \sigma}{s ::_\rho \mathbf{read}(T)@\ell.P \;\parallel\; s' ::_{\rho'} \langle t \rangle \longmapsto s ::_\rho P\sigma \parallel s' ::_{\rho'} \langle t \rangle} \; (Read)$$

$$s ::_\rho \mathbf{newloc}(s').P \longmapsto (\nu s')(s ::_\rho P \parallel s' ::_{\rho[s'/\mathbf{self}]} \mathbf{nil}) \; (New)$$

$$\frac{N_1 \longmapsto N_1'}{N_1 \parallel N_2 \longmapsto N_1' \parallel N_2} \; (Par) \qquad \frac{N \longmapsto N'}{(\nu s)N \longmapsto (\nu s)N'} \; (Res)$$

$$\frac{N \equiv M \longmapsto M' \equiv N'}{N \longmapsto N'} \; (Str)$$

Table 7. KLAIM operational semantics

order to determine the location that is the target of the action, and to evaluate the argument tuple t, in order to evaluate the expressions within t and, hence, to replace the locality variables occurring free in t. This way, if the argument tuple contains a field with a process, the corresponding field of the evaluated tuple contains the process resulting from the evaluation of its free variables w.r.t. the environment ρ (*static scoping* discipline). A *dynamic linking* strategy is adopted in rule *(Eval)* for the action $\mathbf{eval}(P)@\ell$, since the local allocation environment is used only to evaluate ℓ, while the argument process P is not interpreted (i.e. free variables in P will be translated according to the remote allocation environment). Rules *(In)* and *(Read)* require existence of a matching datum in the target node. The tuple is then used to replace the free occurrences of the variables bound by the template in the continuation of the process performing the

(Extended expressions)	$f \mathrel{+}= \mathbf{out}(t)@\eta : \ell$	$\mid$	$\mathbf{eval}(P)@\eta : \ell$	
	$\mid \mathbf{in}(T)@\eta : \ell$	$\mid$	$\mathbf{read}(T)@\eta : \ell$	
(Named nets)	$\mathcal{K} ::= \{\eta_i ::_{\rho_i} N_i\}_{i \in I}$			

Table 8. KORC syntax

actions. With action **in**, the matched datum is consumed while with action **read** it is not. In rule *(New)*, the environment of a new node is derived from that of the creating one with the obvious update for the **self** variable (notice that the environment update $\rho[s'/\mathbf{self}]$ is defined only if s' is not in the codomain of ρ). Therefore, the new node inherits all the bindings of the creating node. The last three rules are standard: rule *(Par)* states that if a part of a larger net evolves, the whole net evolves accordingly; rule *(Res)* states that reductions are not inhibited even if they arise from the scope of a restriction; finally, rule *(Str)* states that structural congruent nets have the same reductions.

2.3 Korc: a language for orchestrating Klaim agents

We now show how the orchestration approach of ORC and the network-aware one of KLAIM can be combined in order to define a new formalism for orchestrating concurrent processes coordinated via distributed tuple spaces. In particular, in this section we present the syntax and the operational semantics of the resulting calculus, which we have called KORC.

A KORC program consists of a configuration $(f, \mathcal{K})$, where f is an extended ORC expression (possibly coming with a set of expression definitions) and $\mathcal{K}$ is a set of *named* KLAIM *nets*. To execute a program, its expression is evaluated while the nets concurrently run. Formally, the KORC syntax is defined in Table 8, where f is an ORC expression (as in Table 1) extended with KLAIM actions, η ranges over *net names*, and nets N, processes P, tuples t and templates T are defined in Table 4. We assume that the KORC set of values, ranged over by m, includes the KLAIM set of values (containing locality names, basic values and processes). As a matter of notation, given a grammar such as $e ::= p_1 \mid \ldots \mid p_m$, we write $e+ = p_{m+1} \mid \ldots \mid p_{m+n}$ as a shorthand for $e ::= p_1 \mid \ldots \mid p_{m+n}$. Moreover, in the sequel, we will use $\uplus$ to denote the disjoint union operator between sets.

A KORC expression can interact with different KLAIM nets that can be referred (and distinguished) by means of net names. Specifically, a *named net* is a triple $\eta ::_\rho N$, where η is the name of the net, ρ is the allocation environment used for evaluating the add/withdrawal/access actions performed by the KORC expression with target the named net, and N is a KLAIM net. Besides site and expression calls, a KORC expression can perform **out**, **eval**, **in** and **read** actions over named nets within the associated set $\mathcal{K}$. Such actions, in KORC, have a further argument η that explicitly indicates the target net. Actions **newloc** cannot be directly executed by a KORC expression, since they only act locally in

a KLAIM node. Anyway, they can be indirectly performed by resorting to **eval** actions.

A KORC program $(f, \{\eta_i ::_{\rho_i} N_i\}_{i \in I})$ is *well-formed* if names η_i are pairwise distinct and for each $i \in I$ we have that **self** is not in the domain of ρ_i and N_i is a well-formed net (see Section 2.2). Hereafter, we will only consider well-formed programs. Notably, we consider named nets, rather than unnamed ones, to avoid requiring locality names of all nets to be pairwise distinct. In fact, while this is reasonable at level of a single net, it becomes, from a practical point of view, a requirement too strong at level of a distributed and loosely coupled environment where some different and independent subnets cohabit. Notice also that the requirement about the **self** variable is due to the fact that ρ_i are used to evaluate actions executed by a KORC expression and that, hence, are not hosted by any KLAIM node.

Example 3. Consider an application that combines the functionalities of the applications described in Examples 1 and 2 to calculate the current temperature of a given set of places of interest within the State of New York. We suppose here that the localities s_{zip} and s_{places} of Example 2 belong to a KLAIM net η_{zip}, while the locality $s_{temperature}$, where the results will be stored, belongs to another net η_{ny}, which is dedicated to applications concerning the State New of York.

The whole scenario is rendered in KORC as

$$
\begin{array}{l}
(\, f, \{\eta_{zip} ::_{\{l_{places} \mapsto s_{places}\}} \\
\qquad (\nu s_{zip})(\, s_{zip} ::_{\{\mathbf{self} \mapsto s_{zip}\}} \ldots \mid \langle 10109, \text{“}\mathit{Times\ Square}\text{”}\rangle \mid \ldots \\
\qquad\qquad \parallel \ s_{places} ::_{\{\mathbf{self} \mapsto s_{places}, l_{zip} \mapsto s_{zip}\}} \langle \text{“}\mathit{Times\ Square}\text{”}\rangle \mid \ldots)\,, \\
\quad \eta_{ny} ::_{\{l_{nyTemp} \mapsto s_{temperatures}\}} \\
\qquad\qquad (\, s_{temperatures} ::_{\{\mathbf{self} \mapsto s_{temperatures}\}} \mathbf{0} \ \parallel \ \ldots \,) \,\}\,)
\end{array}
$$

where locality s_{zip} here is assumed restricted in order to disallow processes external to the net to directly access to the zip codes database.

The expression f is as:

$$\mathbf{eval}(\mathit{Filter}(10001, 14905))@\,\eta_{zip} : l_{places} >> f_{calculateTemperature}$$

Basically, it spawns the process Filter on the node s_{places} within η_{zip} and than activates the expression $f_{calculateTemperature}$, which is defined as follows:

$$
\begin{array}{l}
f_{calculateTemperature} \triangleq \\
\mathbf{in}(!zip)@\,\eta_{zip} : l_{places} > \langle x_{zip}\rangle > \\
\quad \mathit{Weather}(x_{zip}) > \langle x_{city}, x_{weather}, x_{temperature}\rangle > \\
\qquad \mathit{FahrenheitToCelsius}(x_{temperature}) > \langle x_{celsius}\rangle > \\
\qquad\quad \mathbf{out}(x_{zip}, x_{celsius})@\,\eta_{ny} : l_{nyTemp} >> f_{calculateTemperature}
\end{array}
$$

This expression cyclically reads a zip code generated by the process Filter, invokes the sites $\mathit{Weather}$ and $\mathit{FahrenheitToCelsius}$ and inserts the result into the localities $s_{temperatures}$ of η_{ny}.

$$\frac{\rho(\ell) = s' \qquad \mathcal{E}[\![\, t \,]\!]_\rho = t' \qquad (fn(t') \cup \{s'\}) \not\subseteq (bn(N) \cup \bar{s})}{(\mathbf{out}(t)@\,\eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \mathbf{nil})\}) \xrightarrow{!\langle\rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \langle t'\rangle)\})} \textit{(Ko-out)}$$

$$\frac{\rho(\ell) = s' \qquad (fn(P) \cup \{s'\}) \not\subseteq (bn(N) \cup \bar{s})}{(\mathbf{eval}(P)@\,\eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \mathbf{nil})\}) \xrightarrow{!\langle\rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} P)\})} \textit{(Ko-eval)}$$

$$\frac{\rho(\ell) = s' \qquad match(\mathcal{E}[\![\, T \,]\!]_\rho, t) = \sigma \qquad (fn(T) \cup \{s'\}) \not\subseteq (bn(N) \cup \bar{s})}{(\mathbf{in}(T)@\,\eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \langle t\rangle)\}) \xrightarrow{!\langle t\rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu\, \bar{s}\backslash fn(t))(N \parallel s' ::_{\rho'} \mathbf{nil})\})} \textit{(Ko-in)}$$

$$\frac{\rho(\ell) = s' \qquad match(\mathcal{E}[\![\, T \,]\!]_\rho, t) = \sigma \qquad (fn(T) \cup \{s'\}) \not\subseteq (bn(N) \cup \bar{s})}{(\mathbf{read}(T)@\,\eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \langle t\rangle)\}) \xrightarrow{!\langle t\rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu\, \bar{s}\backslash fn(t))(N \parallel s' ::_{\rho'} \langle t\rangle)\})} \textit{(Ko-read)}$$

$$\frac{N \longmapsto N'}{(f, \mathcal{K} \uplus \{\eta ::_\rho N\}) \xrightarrow{\tau} (f, \mathcal{K} \uplus \{\eta ::_\rho N'\})} \textit{(Ko-net)}$$

Table 9. KORC operational semantics (additional rules)

The operational semantics of KORC is given in terms of a labelled transition relation $\xrightarrow{a}$ over configurations, which relies on the reduction relation $\longmapsto$ over KLAIM nets defined by the rules in Table 7. As in the semantics of ORC, labels a indicate either internal events τ or publication events $!\bar{m}$. The operational rules defining the labelled transition relation are those in Table 9 together with those in Table 3 extended to KORC configurations in standard way. For example, rule *(Sym1)* in Table 3 extends to configurations as follows:

$$\frac{(f, \mathcal{K}) \xrightarrow{a} (f', \mathcal{K}')}{(f \mid g, \mathcal{K}) \xrightarrow{a} (f' \mid g, \mathcal{K}')}$$

Notably, site and expression calls cannot modify the set $\mathcal{K}$, only the KORC actions **out**, **eval**, **in** and **read** can. Notice also that, since KORC inherits pairwise disjoint variables sets from KLAIM, the rule $\mathcal{M}(x, m) = [m/x]$ of the pattern-matching function $\mathcal{M}(\cdot, \cdot)$ must be replaced by the following ones:

$$\mathcal{M}(x, v) = [v/x] \qquad \mathcal{M}(l, s) = [s/l] \qquad \mathcal{M}(X, P) = [P/X]$$

Let us now comment on the operational rules in Table 9. All actions evolve to expression $\mathbf{0}$, act on a net named η, require the existence of the target

node s' (which must not be restricted in η) and exploit the allocation environment ρ for evaluating their arguments. As a matter of notation, we abbreviate $(\nu s_1) \ldots (\nu s_n)$ to $(\nu \bar{s})$ with $\bar{s} = \langle s_1, \ldots, s_n \rangle$. Actions **out** and **eval**, rules *(Ko-out)* and *(Ko-eval)*, can be performed only if the components they intend to insert in s' (i.e. the evaluated tuple $\langle t' \rangle$ or the process P) do not contain locality names restricted in η. If such actions can be performed, a signal $\langle\rangle$ is published. Notice that a signal is produced, rather than a τ event, to enable possible sequential or asymmetric parallel compositions (see rules *(Seq1)* and *(Asym2)* in Table 3). Similarly, actions **in** and **read**, rules *(Ko-in)* and *(Ko-read)*, can be performed only if the template T does not contain locality names restricted in η, because a private name cannot be matched by any name used outside the net (i.e. private names cannot be 'guessed'). Instead, these actions can be performed if the tuple t they intend to withdraw/read contains some locality names restricted in η; in this case, the restriction of such names is removed. If a matching datum t exists in the target node, actions **in** and **read** can proceed and publish the withdrawn/read tuple $\langle t \rangle$. Notice that, in order to properly integrate **in** and **read** actions with the binding operators of Orc, in rules *(Ko-in)* and *(Ko-read)* the generated substitution σ is not applied and the complete withdrawn/read tuple is published (rather than, e.g., only the values matching with the template variables). The values in the returned tuple can be then caught via pattern-matching through sequential or asymmetric parallel compositions. Finally, rule *(Ko-net)* says that Klaim nets in $\mathcal{K}$ can freely evolve w.r.t. the evolution of expression f.

Remark 1 (Actions à la Klaim*).* In Korc, as previously pointed out, execution of actions **in** and **read** does not involve a direct application of a substitution, but simply the publication of the withdrawn/read tuple. Anyway, **in** and **read** actions à la Klaim can be easily expressed: e.g. expression $\mathbf{in}(5, !x)@\,\eta : \ell\,.f$, where a substitution for x is applied to f, can be rendered in Korc as $\mathbf{in}(5, !y)@\,\eta : \ell > \langle 5, x \rangle > f$, where y is a variable never used elsewhere (i.e. it can be thought of as a wildcard).

Remark 2 (Dealing with private names). In Klaim, locality names can be private, i.e. restricted by means of the operator (νs). In fact, the freshness of a private name can be reasonable ensured by a middleware supporting the execution of a Klaim net (such as, e.g., Klava [5]). Instead, the loosely coupled nature of the service-oriented architecture at the base of Korc implies that names freshness cannot be guaranteed over a whole global net, which can be potentially composed of many independent Klaim subnets. Therefore, in Korc, when a private name is extracted from a Klaim net, through an **in**/**read** action, the name becomes public (somehow, the effect is similar to that of rule *(open)* in π-calculus [29]).

Of course, other different policies for dealing with private names can be considered. For example, we could prevent the access to all Klaim tuples containing private names. This can be realized by simply adding the premise $fn(t) \not\subseteq (bn(N) \cup \bar{s})$ to rules *(Ko-in)* and *(Ko-read)*. However, this requirement

could turn out to be too strong. Another possibility could be to extend KORC expressions with a restriction operator (νs) but this, as already explained above, would violate the loosely coupling principle underlying KORC.

3 Korc at work on an e-commerce case study

In this section, we illustrate an application of KORC to a simplified but realistic electronic marketplace scenario, where a number of on-line stores allow client applications to (read-only) access their data about items availability and to place orders. We suppose that each store has one on-line portal and relies on many 'realworld' stores, each of which with its own warehouse. Specifically, here we consider a client application that aims at finding a store that has in stock a given quantity of a specific item, by concurrently accessing the stores data, and then placing an order to the first store found. For the sake of presentation, hereafter we consider a scenario consisting of only three on-line stores.

The whole scenario, graphically depicted in Figure 2, can be rendered in KORC as

$$(f, \{\eta_{store1} ::_{\rho} N_1,\ \eta_{store2} ::_{\rho} N_2,\ \eta_{store3} ::_{\rho} N_3,\ \eta_{client} ::_{\rho} (s ::_{\{\mathbf{self} \mapsto s\}} \mathbf{nil})\})$$

where ρ stands for $\{l \mapsto s\}$ and each net N_i has the following form

$$\begin{array}{l} s_1 ::_{\{\mathbf{self}\mapsto s_1, l\mapsto s, l_{next}\mapsto s_2, l_{end}\mapsto s_e\}} \langle t^1_{i1}\rangle \mid \ldots \mid \langle t^{k_i}_{i1}\rangle \mid \langle \text{“}next\text{”}, l_{next}\rangle \\ \parallel\ s_2 ::_{\{\mathbf{self}\mapsto s_2, l\mapsto s, l_{next}\mapsto s_3, l_{end}\mapsto s_e\}} \langle t^1_{i2}\rangle \mid \ldots \mid \langle t^{w_i}_{i2}\rangle \mid \langle \text{“}next\text{”}, l_{next}\rangle \\ \parallel\ \ldots\ \parallel\ s_{m_i} ::_{\{\mathbf{self}\mapsto s_{m_i}, l\mapsto s, l_{next}\mapsto s_e, l_{end}\mapsto s_e\}} \langle t^1_{im}\rangle \mid \ldots \mid \langle t^{r_i}_{im}\rangle \mid \langle \text{“}next\text{”}, l_{next}\rangle \\ \parallel\ s ::_{\{\mathbf{self}\mapsto s, l_1\mapsto s_1\}} \langle \text{“}start\text{”}, l_1\rangle \end{array}$$

Intuitively, each store *store i* is modelled by a site $M_{store\,i}$, representing its on-line portal, and a net named $\eta_{store\,i}$, whose nodes s_j represent the data storages of its warehouses, while node s is used for computation support. Instead, a client application consists of an expression f and a net named η_{client}, which contains a node, initially empty, used to elaborate the retrieved data. Finally, each tuple t^u_{ij} represents the data of a specific item stored inside the warehouse s_j of the store *store i*. Specifically, an item tuple has the form $\langle id, q, p\rangle$, where *id* is the item identifier, q (with $q > 0$) is the quantity available at the warehouse, and p is the price (which can be different from a warehouse to another). We assume that each node contains at most one tuple for each item identifier. Notice that the item tuples have been thought to be read only data for client processes; this could be guaranteed by resorting to, e.g., the tuple access control systems like that presented in [22].

The expression f is defined as follows:

$$\begin{array}{l} f \triangleq \\ \mathbf{eval}(\mathit{FindItem}(\text{“}\mathit{itemId}3\text{”}, 20, \text{“}\mathit{reqId}12\text{”}))@\,\eta_{store1} : l \\ \mid\ \mathbf{eval}(\mathit{FindItem}(\text{“}\mathit{itemId}3\text{”}, 20, \text{“}\mathit{reqId}12\text{”}))@\,\eta_{store2} : l \\ \mid\ \mathbf{eval}(\mathit{FindItem}(\text{“}\mathit{itemId}3\text{”}, 20, \text{“}\mathit{reqId}12\text{”}))@\,\eta_{store3} : l \\ \mid\ f_{moveFromStore1} \mid f_{moveFromStore2} \mid f_{moveFromStore3} \mid g \end{array}$$

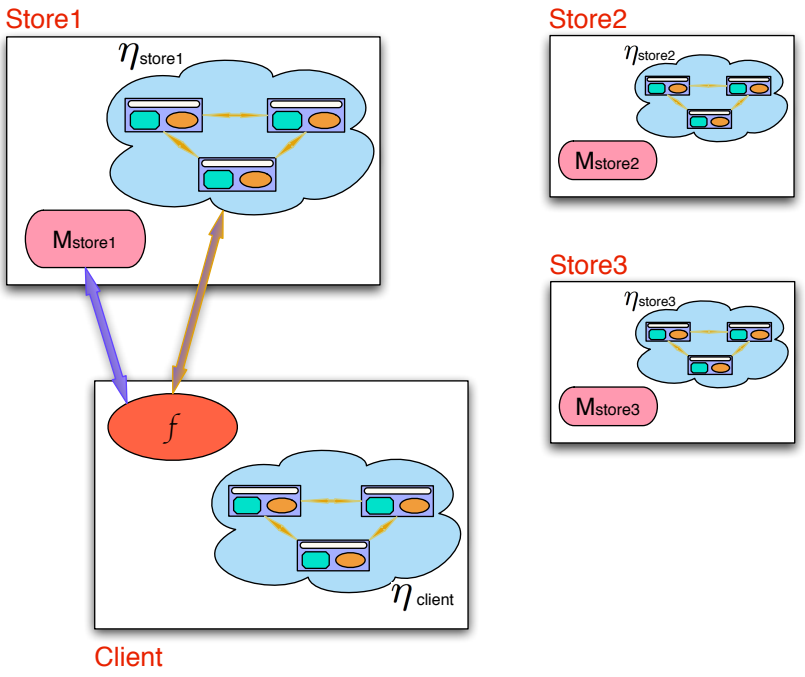

Fig. 2. The e-commerce scenario in KORC

Basically, it represents a client's search request for 30 items of type "$itemId3$" whose maximum price per item that the client is willing to pay is less or equal to 20. To avoid that data of different search requests are erroneously mixed together, a request identifier, say "$reqId12$", is provided by the client and inserted into each tuple. Of course, the above expression could be parameterized w.r.t. item identifier, price, request identifier and quantity, but we prefer to leave it as it is for the sake of presentation.

Specifically, by means of three **eval** actions, the client expression spawns three copies of the process $FindItem$ into the locality s of each store net. Such process looks for tuples having as arguments the item identifier "$itemId3$" and a price less or equal to 20. A copy of each tuple (extended with the request identifier "$reqId12$") that meets this requirement is stored in the locality s of the net. Then, by means of three expression calls $f_{moveFromStore\,i}$, as tuples are inserted into the node s of each store's net, they are moved to the node s of the client's net. Each expression $f_{moveFromStore\,i}$ is defined as a recursive expression performing a sequence of **in** and **out** actions:

$$
\begin{array}{l}
f_{moveFromStore\,i} \triangleq \\
\mathbf{in}(\text{“}itemId3\text{”}, !x_q, !x_p, \text{“}reqId12\text{”})@\,\eta_{store\,i} : l \\
\quad > \langle \text{“}itemId3\text{”}, x_q, x_p, \text{“}reqId12\text{”} \rangle > \\
\qquad \mathbf{out}(\text{“}store\,i\text{”}, \text{“}reqId12\text{”}, x_q)@\,\eta_{client} : l >> f_{moveFromStore\,i}
\end{array}
$$

where we have used the short-hand expression $f >> g$ to denote the expression $f > \langle\cdot\rangle > g$. Notably, in performing such movements, information about prices and item identifiers are left out, while information about the source stores are added.

The KLAIM process $FindItem$ is defined as follows:

$$FindItem(itemId, maxPrice, reqId) \triangleq$$
$$\mathbf{read}(\text{“}start\text{”}, !l_{start})@\mathbf{self}.\ \mathbf{eval}(Find(itemId, maxPrice, reqId))@l_{start}$$

It locally reads the 'start' locality of the hosting net and then activates there a mobile process $Find$ (i.e. an *agent*) that will visit all net nodes to find the availability of the wanted item. The mobile process is defined as follows:

$$\begin{array}{l} Find(itemId, maxPrice, reqId) \triangleq \\ \mathbf{read}(itemId, !q, !p)@\mathbf{self}. \\ \quad \mathbf{if}\ (p \leqslant maxPrice)\ \mathbf{then}\ \mathbf{out}(itemId, q, p, reqId)@l \\ |\ \mathbf{read}(\text{“}next\text{”}, !l_n)@\mathbf{self}. \\ \quad \mathbf{if}\ (l_n \neq l_{end})\ \mathbf{then}\ \mathbf{eval}(Find(itemId, maxPrice, reqId))@l_n \end{array}$$

The process simply checks if a tuple for the given item is present locally; if the item per price is not greater than the maximum price then it add a corresponding tuple to the node s of the hosting net (referred by means of the locality variable l). Moreover, it reads the locality of the next node and, if it there exists, than a new copy of the process is spawned on such node. This second check exploits the locality variable l_{end} that is properly bound by the allocation environment of each net node. For the sake of simplicity, in defining the above agent we have used a conditional construct (which can be easily programmed by exploiting the dynamic creation of new nodes and the parallel composition operator) and omitted trailing occurrences of **nil**.

The expression g is defined as follows:

$$\begin{array}{l} g \triangleq \\ \mathbf{out}(\text{“}sum\text{”}, \text{“}store1\text{”}, \text{“}reqId12\text{”}, 0)@\,\eta_{client} : l \\ |\ \mathbf{out}(\text{“}sum\text{”}, \text{“}store2\text{”}, \text{“}reqId12\text{”}, 0)@\,\eta_{client} : l \\ |\ \mathbf{out}(\text{“}sum\text{”}, \text{“}store3\text{”}, \text{“}reqId12\text{”}, 0)@\,\eta_{client} : l \\ |\ (\ (\ if(x = \text{“}store1ok\text{”})\ >>\ M_{store1}(\text{“}itemId3\text{”}, 30)\ ) \\ \quad |\ (\ if(x = \text{“}store2ok\text{”})\ >>\ M_{store2}(\text{“}itemId3\text{”}, 30)\ ) \\ \quad |\ (\ if(x = \text{“}store3ok\text{”})\ >>\ M_{store3}(\text{“}itemId3\text{”}, 30)\ )\ ) < \langle x \rangle < (\ g_1 \mid g_2 \mid g_3\ ) \end{array}$$

It adds to the node s of the client's net three tuples containing the partial sum of the quantity of the requested item available at each store (initially set to 0). It also starts the concurrent evaluation of three expressions g_i, each of which computes the sum of the item quantity for a store and publishes the string "*store i ok*" if the store has in stock at least 30 items of the requested type. The asymmetric parallel composition operator is used here to bind the variable x with the string "*store i ok*" and to terminate the evaluation of the other functions g_j, with $j \neq i$. Then, according to the published string, the corresponding site

$M_{store\,i}$ is called to place an order. We have exploited here the fundamental[5] Orc site $if(b)$, which returns a signal $\langle\cdot\rangle$ if b evaluates to $true$, otherwise it does not respond.

Finally, an expression g_i is defined as follows:

$$
\begin{array}{l}
g_i \triangleq \\
\mathbf{in}(\text{“}store\,i\text{”}, \text{“}reqId12\text{”}, !y_q)@\,\eta_{client} : l \\
> \langle \text{“}store\,i\text{”}, \text{“}reqId12\text{”}, y_q \rangle > \\
\mathbf{in}(\text{“}sum\text{”}, \text{“}store\,i\text{”}, \text{“}reqId12\text{”}, !y_{sum})@\,\eta_{client} : l \\
> \langle \text{“}sum\text{”}, \text{“}store\,i\text{”}, \text{“}reqId12\text{”}, y_{sum} \rangle > \\
(\ (\ if(y_q + y_{sum} \geqslant 30)\ >>\ let(\text{“}store\,i\,ok\text{”})\ ) \\
\quad |\ (\ if(y_q + y_{sum} < 30)\ >> \\
\qquad \mathbf{out}(\text{“}sum\text{”}, \text{“}store\,i\text{”}, \text{“}reqId12\text{”}, y_q + y_{sum})@\,\eta_{client} : l\ )\ >>\ g_i\ )
\end{array}
$$

basically, this a recursive expression that, at each recursive call, consumes a tuple containing an item availability and a tuple containing the actual sum, computes the sum between the read values and, if the sum is less than the desired number (i.e. 30) it produces a new 'sum' tuple and calls itself, otherwise publishes the string "$store\,i\,ok$" and terminates. Notably, to publish the string "$store\,i\,ok$", expression g_i exploits the fundamental Orc site $let(x, y, \ldots)$, which returns the argument values as a tuple.

4 Implementation issues

In this section, we first provide a brief overview of the implementations of the programming languages derived from Orc and Klaim, then we give a glimpse of the proof-of-concept implementation of Korc.

4.1 The Orc programming language

Although Orc was originally conceived as a process calculus, then it has evolved into a complete language for programming orchestration-based concurrent applications [26]. Such a programming language provides the Orc's orchestration operators and the site call construct with their original syntax, while expression definitions take the form `def f(x1,...,xn)` $= f_{body}$. The language is also equipped with arithmetic and logical operators, data structures, a conditional construct, and a variable binder construct `val` (e.g., `val x = 5` binds `x` to `5`). Moreover, the language supporting libraries provides predefined functions and fundamental sites.

A feature of the Orc implementation, particularly relevant for our purposes, is its capability of integrating with Java applications. In fact, Java classes can be accessed by an Orc expression as sites. To make a class available to an

[5] To effective programming in Orc, the language is equipped with a few 'fundamental' sites (e.g. $if(b)$, $let(x, y, \ldots)$) that have to be considered local and whose behavior is predefined and predictable [37].

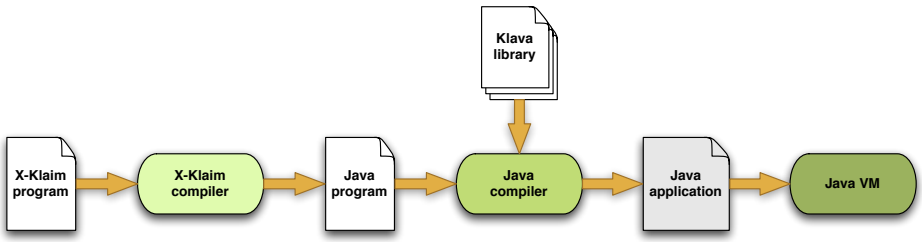

Fig. 3. X-KLAIM framework

expression, a `site` declaration and a variable binding must be used like in the following example

```
site WebServiceFactory = orc.lib.net.Webservice
val service = WebServiceFactory(...)
```

where the variable `service` can be then used for invoking a web service defined in the Java class `orc.lib.net.Webservice`. To be accessed as an ORC site, a Java class must extended one of the specific classes provided with the ORC's libraries (e.g. `EvalSite`).

We refer the interested reader to [1, 2] for a complete account of the ORC programming language and its supporting libraries. The source code and binaries of the ORC's implementation are released under the BSD License and can be downloaded from http://orc.csres.utexas.edu.

4.2 The X-Klaim programming language

Similarly to the evolution of ORC to a full programming language, also the process calculus KLAIM has been extended, to effectively program distributed networked applications, with high-level features, such as variable declarations, assignments, and (standard) control flow constructs. As shown in Figure 3, the implementation of the resulting programming language, called X-KLAIM (eXtended KLAIM, [6]), is based on a compiler, which generates Java code, and on the Java library KLAVA [5], which provides the run-time support for X-KLAIM actions.

For example, the KLAIM net N_1 belonging to the *store*1 of the e-commerce case study introduced in Section 3 can be rendered in X-KLAIM as follows:

```
nodes
  shop11::succ ~ localhost:11002,finish ~ localhost:11005, s ~ localhost:11004
    port 11001
    begin
        out("next", succ)@self;
        out("id1",10,11)@self;
        ...
    end;

  shop12::succ ~ localhost:11003,finish ~ localhost:11005, s ~ localhost:11004
    port 11002
    begin
        out("next", succ)@self;
```

```
        out("id1",6,15)@self;
        ...
    end;

  ...

  l1::startS ~ localhost:11001
    port 11004
    begin
        out("start",startS)@self
    end
endnodes
```

A net, as expected, is a collection of node definitions, which must be included within `nodes` and `endnodes`. A node, e.g. the first one in the net above, is defined by specifying its name (`shop11`), its allocation environment (containing, e.g., the mapping from the locality variable `succ` to the locality `localhost : 11002`), the port (`11001`) where it is listening, and a set of processes running on it (`out("next", succ)@self`; ...). It is worth noticing that the (physical) locality of a node is not defined by its name, but by the IP address of the computer where the node will run (in our example, this always is `localhost`) together with its port number.

The process definition is rendered in X-KLAIM through the construct

`rec` *Proc_name* [*parameters_decl*] `declare` *variables_decl* `begin ... end`

For example, the definition of the process *FindItem* exploited in the e-commerce case study is as follows:

```
rec FindItem [itemId: str, maxPrice:int, reqId:str ]
  declare
        var locstart: loc
  begin
        read("start",!locstart)@self;
        eval(Find(itemId,maxPrice,reqId))@locstart
  end
```

Notice that `str` and `int` are the standard base types for strings and integers, while `loc` is the type for locality variables.

A complete documentation of X-KLAIM and KLAVA, together with source and binary files (distributed under the GNU General Public License), can be found at http://music.dsi.unifi.it/klaim.html.

4.3 The Korc implementation

To speed up the experimentation with the programming paradigm fostered by KORC, we have exploited the compile- and run-time support tools for ORC and KLAIM presented above to implement a prototype implementation of KORC. The underlying idea is as follows: KORC expressions are rendered as standard ORC expressions that rely on ad-hoc sites for performing the KLAIM actions. Specifically, we have developed a Java class `com.orcNode`, extending `EvalSite`, that can be used to define a new type of ORC site and that relies on the KLAVA library for performing the KLAIM actions. Since KLAVA uses types for values

different from those of Orc, i.e. `KString`, `KInteger`, etc., and allows patterns to use both actual and formal parameters, we have also developed another kind of Orc site, `com.orcTuple`, that can be used to create objects having the correct types for invoking the Klava methods.

An example of a Korc expression rendered in our implementation is as follows

```
site orcNode = com.orcNode
site orcTuple = com.orcTuple

val client = orcNode("client",15000,"localhost",9999)
val store1 = orcNode("store1",15001,"localhost",9998)
val store2 = orcNode("store2",15002,"localhost",9997)
val store3 = orcNode("store3",15003,"localhost",9996)
val c = orcTuple()

def addLocality() =
   client.addEnv("l",14000)
   >> store1.addEnv("l",11004)
   >> ...

def moveToStore1() =
   store1.in(c.tuple("id3",c.intFormal(),c.intFormal(),"reqId12"),c.locality("l"))
   > x > c.get(x,1) > z >
   client.out(c.tuple("store1","reqId12",z),c.locality("l")) >> moveToStore1()

def ...

addLocality()
>> (startSearch()
    >> (moveToStore1() | moveToStore2() | moveToStore3() | g()) )
```

At the beginning, our sites `com.orcNode` and `com.orcTuple` are declared and assigned to some variables. Each `com.orcNode` site permits interfacing with a Klaim net; thus, the corresponding variable can play the role of net name in Korc actions. For example, the action $\mathbf{out}(\text{“}store1\text{”}, \text{“}reqId12\text{”}, z)@\,\eta_{client} : l$ is rendered as

```
client.out(c.tuple("store1","reqId12",z),c.locality("l"))
```

where `client` represents η_{client}.

It is worth noticing that a `com.orcNode` site corresponds to a node belonging to the corresponding Klaim net (in the example above, for the `client` net such node has name `client` and locality `localhost : 9999`). Thus, specific methods have been provided to set the allocation environment of such nodes and to load processes into them: `addEnv` and `loadProcess`, respectively. Notice also that formal parameters are unnamed in `com.orcTuple` tuples and, hence, a `get` method has to be used after *in*/*read* actions to extract the values associated to the formal parameters by pattern-matching.

We refer the interested reader to Appendix A for the Java code of classes `com.orcNode` and `com.orcTuple`. Such classes can be downloaded from http://rap.dsi.unifi.it/korc/korc.zip and can be installed in Orc as any other Java class defining an external site. The Korc implementation has been tested with Orc 1.1.0, X-Klaim 2.b9 and Klava 2.b1.

The complete specification of the e-commerce case study, written in the syntax accepted by our tool, is relegated to Appendix B. A run of the program consists on the activation of the KLAIM nets and on the evaluation of the KORC expression.

5 Concluding Remarks

We have introduced KORC, a formalism aiming at reconciling the orchestration paradigm of ORC with the coordination one of KLAIM. Specifically, we have formally defined the syntax and operational semantics of KORC, and developed a prototype implementation supporting KORC programming.

Related work. From the theoretical point of view, the formalisms closest to ours are ORC and KLAIM. In fact, to define KORC, we have chosen them as representative of the broader classes of orchestration calculi (as, e.g., [28, 11, 27, 23, 8, 12, 9]) and coordination calculi for network-aware and mobility programming (as, e.g., [20, 24, 13, 36]). Although KORC, w.r.t. the above calculi, does not provide any new primitive or mechanism, since it results from the combination of ORC and KLAIM, it enables experimenting and reasoning on a novel programming paradigm that allows the combined use of orchestration and coordination operators.

From the technological point of view, our work falls within a main line of research that aims at developing programming frameworks based on, or strongly inspired to, process calculi. Among the several proposals, we want to mention below those designed for programming distributed networked applications. JCaSPiS [4] is a Java implementation of the service-oriented calculus CaSPiS (Calculus of Sessions and Pipelines, [9]) that, as well as KORC, takes inspiration by ORC (in particular, for the use of the sequential composition operator, called *pipeline*, over value streams). CaSPiS's implementation is based on the generic Java framework IMC (Implementing Mobile Calculi, [3]) that provides recurrent mechanisms for network applications and, hence, can be used as a middleware for the implementation of different process calculi. JOLIE [31, 32] is an interpreter written in Java for a programming language based on the process calculus SOCK (Service Oriented Computing Kernel, [23]), which is a formalism inspired by the WS-BPEL language for formalizing some fundamental concepts of Service-Oriented Computing, such as the design of a service behaviour, its deployment in an executing enviroment and the composition of services within a system. JSCL (Java Signal Core Layer, [19]) is a Java-based coordination middleware for services based on the event notification paradigm of the Signal Calculus [18], a variant of the π-calculus with explicit primitives to deal with event notification and component distribution. Finally, `PiDuce` [14] is a distributed run-time environment that implements a variant of the asynchronous π-calculus extended with native XML values, datatypes and patterns. The environment also permits interacting and experimenting with web services technologies.

Anyway, our work differs from all ones above because we have not developed an interpreter for KORC from scratch, but we have exploited and extended the established implementation of ORC and KLAIM.

Ongoing and future work. At foundational level, we intend to investigate the extension of KORC with name passing communication. Indeed, currently ORC does not allow expressions to receive site names and use them in site calls, e.g. the term $M() > x > x(5)$ is not an ORC expression since the variable x cannot occur as a site name in the call $x(5)$ (see ORC's syntax in Table 1). In KORC, besides site name passing, also net name passing is disallowed. In fact, on the one hand, a language that permits passing net names but not site ones would not be particularly meaningful for programming networked applications. On the other hand, the KORC's implementation exploits site calls for executing KLAIM actions over nets (see Section 4.3), hence net name passing cannot be realized without site name passing in the current implementation. We also plan to investigate extension of KORC with other orchestration/coordination primitives, e.g. the 'otherwise' operation introduced in [26].

At support tool level, we intend revise the programming language based on KORC presented in Section 4.3. Tto make it more usable by programmers, for example, KLAIM actions should have a syntax more similar to that shown in Table 8 and allows the direct use of named formal parameters. This could be realized, e.g., by means of a pre-compiling step. To further simplifying the task of writing KORC programs, we also intend to provide programmers with an Eclipse-based development environment relying on the Xtext framework [17]. Finally, while KORC is basically an extension of ORC with KLAIM actions and nets, we are also currently investigating a sort of reverse extension, i.e. KLAIM with mechanisms for calling sites (specifically, web services via SOAP over HTTP). Such extension mainly involves the KLAIM middleware (i.e. X-KLAIM and KLAVA) rather than the process calculus itself. In fact, we rely on standard out/in actions for interacting with web services.

References

1. Orc Reference Manual. Technical report, University of Texas at Austin, 2011. Available at http://orc.csres.utexas.edu/documentation.shtml.
2. Orc User Guide. Technical report, University of Texas at Austin, 2011. Available at http://orc.csres.utexas.edu/documentation.shtml.
3. L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, and M. Loreti. A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In *Proc. of 5th International Conference on Distributed Applications and Interoperable Systems (DAIS)*, volume 3543 of *Lecture Notes in Computer Science*, pages 181–193. Springer, 2005.
4. L. Bettini, R. De Nicola, M. Lacoste, and M. Loreti. Implementing Session Centered Calculi. In *Proc. of 10th international conference on Coordination Models and Languages (COORDINATION)*, volume 5052 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2008.

5. L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.
6. L. Bettini, R. De Nicola, and R. Pugliese. X-Klaim and Klava: Programming Mobile Code. In *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
7. Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gian Luigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The Klaim Project: Theory and Practice. In *Global Computing*, volume 2874 of *LNCS*, pages 88–150. Springer, 2003.
8. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V.T. Vasconcelos, and G. Zavattaro. SCC: a Service Centered Calculus. In *Proc. of 3rd International Workshop on Web Services and Formal Methods (WS-FM))*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.
9. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In *Proc. of 10th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2008.
10. M. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *Proc. of 16th European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2007.
11. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of 8th international conference on Coordination Models and Languages (COORDINATION'06)*, volume 4038 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2006.
12. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *Proc. of 16th European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.
13. L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177213, 2000.
14. S. Carpineti, C. Laneve, and L. Padovani. PiDuce - a project for experimenting Web services technologies. Submitted to Science of Computer Programming. Available at http://www.cs.unibo.it/PiDuce/, 2006.
15. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *Transactions on Software Engineering*, 24(5):315–330, 1998.
16. R. De Nicola, D. Gorla, and R. Pugliese. On the expressive power of KLAIM-based calculi. *Theor. Comput. Sci.*, 356(3):387–421, 2006.
17. H. Behrens et al. Xtext 1.0, 2010. http://www.eclipse.org/xtext.
18. G. Ferrari, R. Guanciale, and D. Strollo. Event based service coordination over dynamic and heterogeneous networks. In *Proc. of 4th International Conference on Service Oriented Computing (ICSOC)*, volume 4294 of *Lecture Notes in Computer Science*, pages 453–458. Springer, 2006.
19. G. Ferrari, R. Guanciale, and D. Strollo. JSCL: A middleware for service coordination. In *Proc. of 26th International Conference on Formal Methods for Networked and Distributed Systems (FORTE)*, volume 4229 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2006.

20. C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *CONCUR*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
21. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
22. D. Gorla and R. Pugliese. Dynamic management of capabilities in a network aware coordination language. *Journal of Logic and Algebraic Programming*, 78:665–689, 2009.
23. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *Proc. of 4th International Conference on Service Oriented Computing (ICSOC)*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.
24. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.
25. D. Kitchin, W.R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, volume 4137 of *LNCS*, pages 477–491. Springer, 2006.
26. D. Kitchin, A. Quark, W.R. Cook, and J. Misra. The Orc Programming Language. In *FMOODS/FORTE*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.
27. C. Laneve and L. Padovani. Smooth Orchestrators. In *Proc. of 9th International Conference Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 3921 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2006.
28. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of 16th European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.
29. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.
30. J. Misra and W.R. Cook. Computation Orchestration: A Basis for Wide-Area Computing. *Journal of Software and Systems Modeling*, 6(1):83–110, 2007.
31. F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. In *Proc. of 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord)*, volume 181 of *Electronic Notes in Theoretical Computer Science*, pages 19–33. Elsevier, 2007.
32. F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *ECOWS*, pages 13–22. IEEE Computer Society Press, 2007.
33. OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007. Available at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
34. D. Prandi and P. Quaglia. Stochastic COWS. In *Proc. of 5th International Conference on Service Oriented Computing (ICSOC)*, volume 4749 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 2007.
35. 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.
36. M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *POPL*, pages 291–302. ACM, 1999.
37. I. Wehrman, D. Kitchin, W.R. Cook, and J. Misra. A timed semantics of orc. *Theoretical Computer Science*, 402(2-3):234–248, 2008.

A Korc implementation: com.orcNode and com.orcTuple classes

We report in this appendix the Java code of the classes com.orcNode and com.orcTuple (Listings 1.1 and 1.2), which can be used to create Orc sites for accessing Klaim nets.

Listing 1.1. Java class com.orcNode

```
1  package com;
2
3  import orc.error.runtime.TokenException;
4  import orc.runtime.sites.EvalSite;
5  import orc.runtime.sites.DotSite;
6  import orc.runtime.sites.Site;
7  import orc.runtime.sites.ThreadedSite;
8  import orc.runtime.*;
9  import Klava.*;
10
11
12 public class orcNode extends EvalSite{
13         public Object evaluate (Args args) throws TokenException{
14                 return new orcNodeInstance(args.stringArg(0),
15                                            args.intArg(1),
16                                            args.stringArg(2),
17                                            args.intArg(3));
18         }
19 }
20
21 class orcNodeInstance extends DotSite {
22         private String loc;
23         private NetNode klavaNode;
24         private Integer nport;
25         private String server;
26
27
28         orcNodeInstance (String loc,Integer nport,String server,Integer port){
29            try{
30                 this.loc = loc;
31                 this.nport = nport;
32                 this.server = server;
33
34                 System.out.println("Create Node" + server+":"+loc+"-"+port);
35                 klavaNode = new Node( loc, nport, server, port );
36
37                 addMember("in", new InMethod());
38                 addMember("out", new OutMethod());
39                 addMember("read", new ReadMethod());
40                 addMember("inp",new InpMethod());
41                 addMember("readp",new ReadpMethod());
42                 addMember("eval", new EvalMethod());
43                 addMember("getLocality", new GetLocalityMethod());
44                 addMember("getPort",new GetPortMethod());
45                 addMember("addEnv",new AddEnvironmentMethod());
46
47                 klavaNode.start();
48
49            } catch(Exception e) {
50                 System.out.println("Error in the Klava node creation");
51                 e.printStackTrace();
52            }
53         }
54
55
56         private class InMethod extends ThreadedSite{
57                 public Object evaluate(Args args) throws TokenException{
58                         Tuple t= null;
59                         try {
60                                 t = (Tuple) args.getArg(0);
61                                 Locality l = (Locality) args.getArg(1);
62                                 klavaNode.in(t,l);
63                         } catch (KlavaException e) {
64                                 e.printStackTrace();
65                         }
66                         return t;
67                 }
68         }
69
70         private class InpMethod extends ThreadedSite{
71                 public Object evaluate(Args args) throws TokenException{
72                         Tuple t = null;
73                         try {
74                                 t = (Tuple) args.getArg(0);
75                                 Locality l = (Locality) args.getArg(1);
```

```
76                      Integer timeout = args.intArg(2);
77
78                              boolean a = klavaNode.in_t(t,l,timeout);
79                      if (a == false){
80                              return new Tuple(new KBoolean(false));
81                      }
82
83                      } catch (KlavaException e) {
84                              e.printStackTrace();
85                      }
86                      return t;
87              }
88      }
89
90      private class ReadMethod extends ThreadedSite {
91              public Object evaluate(Args args)throws TokenException{
92                      Tuple t = null;
93                      try {
94                              t = (Tuple) args.getArg(0);
95                      Locality l = (Locality) args.getArg(1);
96
97                      klavaNode.read(t,l);
98
99                      } catch (KlavaException e) {
100                             e.printStackTrace();
101                     }
102                     return t;
103             }
104     }
105
106     private class ReadpMethod extends ThreadedSite {
107             public Object evaluate(Args args)throws TokenException{
108                     Tuple t = null;
109                     try {
110                             t = (Tuple) args.getArg(0);
111                     Locality l = (Locality) args.getArg(1);
112                     Integer timeout = args.intArg(2);
113
114                             boolean a = klavaNode.read_t(t,l,timeout);
115                     if (a == false){
116                             return new Tuple(new KBoolean(false));
117                     }
118                     } catch (KlavaException e) {
119                             e.printStackTrace();
120                     }
121                     return t;
122             }
123     }
124
125     private class OutMethod extends EvalSite{
126             public Object evaluate(Args args) throws TokenException{
127                 try {
128                     Tuple t = (Tuple) args.getArg(0);
129                     Locality l = (Locality) args.getArg(1);
130
131                     klavaNode.out(t,l);
132
133                     } catch (KlavaException e) {
134                             e.printStackTrace();
135                     }
136                     return signal();
137             }
138     }
139
140     private class EvalMethod extends EvalSite{
141             public Object evaluate(Args args) throws TokenException{
142                     try {
143
144                             KlavaProcess p = (KlavaProcess) args.getArg(0);
145
146                             Locality loc = (Locality) args.getArg(1);
147                             klavaNode.eval(p,loc);
148
149
150                     } catch (KlavaException e) {
151                             e.printStackTrace();
152                     }
153                     return signal();
154             }
155     }
156
157     private class GetLocalityMethod extends Site{
158             public void callSite(Args args, Token token) throws TokenException{
159                     token.resume(loc);
160             }
161     }
162
163     private class GetPortMethod extends Site{
164             public void callSite(Args args, Token token) throws TokenException{
165                     token.resume(nport);
```

```
166                 }
167         }
168
169         private class AddEnvironmentMethod extends EvalSite{
170                 public Object evaluate(Args args) throws TokenException{
171                         try{
172                         klavaNode.addToEnv(args.stringArg(0),
173                                         NetUtils.createNodeAddress(server+":"+args.intArg(1)));
174                         }catch(KlavaException e){
175                                 e.printStackTrace();
176                         }
177                         return signal();
178                 }
179         }
180
181         protected void addMembers() {
182         }
183
184 }
```

Listing 1.2. Java class `com.orcTuple`

```
1  package com;
2
3  import java.lang.reflect.Constructor;
4
5  import orc.error.runtime.TokenException;
6  import orc.runtime.sites.EvalSite;
7  import orc.runtime.sites.DotSite;
8  import orc.runtime.sites.Site;
9  import orc.runtime.*;
10 import Klava.*;
11
12 public class orcTuple extends EvalSite{
13         public Object evaluate (Args args) throws TokenException{
14                 return new orcTupleInstance();
15         }
16 }
17
18
19 class orcTupleInstance extends DotSite{
20
21         orcTupleInstance(){
22                 addMember("tuple", new TupleMethod() );
23                 addMember("stringFormal", new StringFormalMethod());
24                 addMember("intFormal", new IntFormalMethod());
25                 addMember("boolFormal", new BooleanFormalMethod());
26                 addMember("get",new getMethod());
27                 addMember("loadProcess", new loadProcessMethod());
28                 addMember("locality", new localityMethod());
29                 addMember("equals", new EqualsMethod() );
30         }
31
32         private class localityMethod extends Site{
33                 public void callSite(Args args, Token token)throws TokenException{
34                         try{
35                                 token.resume(new LogicalLocality(args.stringArg(0)));
36                         }catch(Exception e){
37                                 e.printStackTrace();
38                         }
39                 }
40         }
41
42         private class TupleMethod extends Site{
43                 public void callSite(Args args, Token token) throws TokenException{
44                         try {
45                                 Tuple t = new Tuple();
46                                 for (int i = 0; i< args.size(); i++){
47                                         if (args.getArg(i) instanceof java.lang.String){
48                                                 KString g = new KString(args.stringArg(i));
49                                                 t.add(g);
50                                         }
51                                         if (args.getArg(i) instanceof java.lang.Integer ){
52                                                 KInteger g = new KInteger(args.intArg(i));
53                                                 t.add(g);
54                                         }
55                                         if (args.getArg(i)instanceof java.math.BigInteger){
56                                                 KInteger g = new KInteger(((java.math.BigInteger)
57                                                                                 args.getArg(i)).intValue());
58                                                 t.add(g);
59                                         }
60                                         if (args.getArg(i) instanceof java.lang.Boolean){
61                                                 KBoolean g = new KBoolean(args.boolArg(i));
62                                                 t.add(g);
63                                         }
64                                         if (args.getArg(i)instanceof Klava.TupleItem){
```

```
                                        t.add(args.getArg(i));
                                }
                        }
                        token.resume(t);
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
    }

    private class StringFormalMethod extends Site{
        public void callSite(Args args, Token token) throws TokenException{
                try {
                        token.resume(new KString());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
    }
    private class BooleanFormalMethod extends Site{
        public void callSite(Args args, Token token) throws TokenException{
                try {
                        token.resume(new KBoolean());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
    }
    private class IntFormalMethod extends Site{
        public void callSite(Args args, Token token) throws TokenException{
                try {
                        token.resume(new KInteger());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
    }

    private class loadProcessMethod extends Site{
        public void callSite(Args args, Token token)throws TokenException{
                try{
                        Object o;
                        if (args.size() == 1){
                                Class process = Class.forName(args.stringArg(0));
                                o = process.newInstance();
                        }else{
                                Class process = Class.forName(args.stringArg(0));
                                Class paramtype[] = new Class[args.size()-1];
                                Object arguments[] = new Object[args.size()-1];
                                for (int i = 1; i < args.size(); i++){
                                        if (args.getArg(i) instanceof java.math.BigInteger ){
                                                paramtype[i-1] = KInteger.class;
                                                arguments[i-1] = new KInteger(
                                                                        ((java.math.BigInteger)
                                                                         args.getArg(i)).intValue());
                                        }
                                        if (args.getArg(i) instanceof java.lang.Integer ){
                                                paramtype[i-1] = KInteger.class;
                                                arguments[i-1] = new KInteger(args.intArg(i));
                                        }
                                        if (args.getArg(i) instanceof java.lang.String ){
                                                paramtype[i-1] = KString.class;
                                                arguments[i-1] = new KString(args.stringArg(i));
                                        }
                                        if (args.getArg(i) instanceof java.lang.Boolean ){
                                                paramtype[i-1] = KBoolean.class;
                                                arguments[i-1] = new KBoolean(args.boolArg(i));
                                        }
                                        if (args.getArg(i) instanceof KInteger
                                            || args.getArg(i) instanceof KBoolean
                                            || args.getArg(i) instanceof KString){
                                                paramtype[i-1] = args.getArg(i).getClass();
                                                arguments[i-1] = args.getArg(i);
                                        }
                                }
                                Constructor ct = process.getConstructor(paramtype);
                                o = ct.newInstance(arguments);
                        }
                        token.resume(o);

                }catch(Exception e){
                        e.printStackTrace();
                }
        }
    }

    private class EqualsMethod extends Site{
        public void callSite (Args args, Token token) throws TokenException{
                try{
                        TupleItem tupleitem = (TupleItem)args.getArg(0);
```

```
155                             Object val = null;
156                             if (args.getArg(1) instanceof java.lang.Integer
157                                 || args.getArg(1) instanceof java.math.BigInteger){
158                                     val = new KInteger(args.intArg(1));
159                             }
160                             if (args.getArg(1) instanceof java.lang.Integer
161                                 || args.getArg(1) instanceof java.math.BigInteger){
162                                     val = new KInteger(args.intArg(1));
163                             }
164                             if (args.getArg(1) instanceof java.lang.String){
165                                     val = new KString(args.stringArg(1));
166                             }
167                             if (args.getArg(1) instanceof java.lang.Boolean){
168                                     val = new KBoolean(args.boolArg(1));
169                             }
170                             token.resume(tupleitem.equals(val));
171                     }catch(Exception e){
172                             e.printStackTrace();
173                     }
174             }
175     }
176
177     private class getMethod extends Site {
178             public void callSite (Args args, Token token) throws TokenException{
179                     try{
180                             Tuple tuple = (Tuple)args.getArg(0);
181                             token.resume(tuple.getItem(args.intArg(1)));
182                     }catch(Exception e){
183                             e.printStackTrace();
184                     }
185             }
186     }
187
188     protected void addMembers() {
189     }
190 }
```

B E-commerce case study implementation

We report in this appendix the KORC code of the e-commerce case study introduced in Section 3. We first show the KORC code of the main function f (Listing 1.3) and then the X-KLAIM code of the nets η_{store1}, η_{store2}, η_{store3} and η_{client} (Listings 1.4, 1.5, 1.6, 1.7, 1.8) Notably, for simplifying the experimentation with the scenario, all KLAIM nets here are set to be run over the local machine (identified by *localhost*).

Listing 1.3. Function f

```
1  site orcNode = com.orcNode
2  site orcTuple = com.orcTuple
3
4  val client = orcNode("client",15000,"localhost",9999)
5  val store1 = orcNode("store1",15001,"localhost",9998)
6  val store2 = orcNode("store2",15002,"localhost",9997)
7  val store3 = orcNode("store3",15003,"localhost",9996)
8
9  val c = orcTuple()
10
11
12 def addLocality() =
13    client.addEnv("lClient",14000)
14    >> store1.addEnv("l",11004)
15    >> store2.addEnv("l",12004)
16    >> store3.addEnv("l",13004)
17
18
19 def moveFromStore1() =
20    store1.in(c.tuple("id3",c.intFormal(),c.intFormal(),"reqId12"),c.locality("l"))
21    > x > c.get(x,1) > z >
22    client.out(c.tuple("store1","reqId12",z),c.locality("lClient")) >> moveFromStore1()
23
24
25 def moveFromStore2() =
26    store2.in(c.tuple("id3",c.intFormal(),c.intFormal(),"reqId12"),c.locality("l"))
27    > x > c.get(x,1) > z >
28    client.out(c.tuple("store2","reqId12",z),c.locality("lClient")) >> moveFromStore2()
29
30
```

```
def moveFromStore3() =
   store3.in(c.tuple("id3",c.intFormal(),c.intFormal(),"reqId12"),c.locality("l"))
   > x > c.get(x,1) > z >
   client.out(c.tuple("store3","reqId12",z),c.locality("lClient")) >> moveFromStore3()


def startSearch() =
   c.loadProcess("FindItem","id3",200,"reqId12")
   > proc > store1.eval(proc,c.locality("l"))
            >> store2.eval(proc,c.locality("l"))
            >> store3.eval(proc,c.locality("l"))

def g() =
   client.out(c.tuple("sum","store1","reqId12",0),c.locality("lClient")) |
   client.out(c.tuple("sum","store2","reqId12",0),c.locality("lClient")) |
   client.out(c.tuple("sum","store3","reqId12",0),c.locality("lClient"))
   |
   (if x = "store1ok" then print("\n \n Order placed at store 1 \n \n") >> stop
    else (if x = "store2ok" then print("\n \n Order placed at store 2 \n \n") >> stop
          else (if x = "store3ok" then print("\n \n Order placed at store 3 \n \n") >> stop ) ) )
      < x < (g1() | g2() | g3())


def g1() =
   client.in(c.tuple("store1","reqId12",c.intFormal()),c.locality("lClient"))
   > x > c.getInt(x,2) > y >
   client.in(c.tuple("sum","store1","reqId12",c.intFormal()),c.locality("lClient"))
   > x > c.getInt(x,3) > sum >
   (if (y + sum ≥ 30) then
      let("store1ok")
    else
      (client.out(c.tuple("sum","store1","reqId12",y + sum),c.locality("lClient")) >> g1()) )


def g2() =
   client.in(c.tuple("store2","reqId12",c.intFormal()),c.locality("lClient"))
   > x > c.getInt(x,2) > y >
   client.in(c.tuple("sum","store2","reqId12",c.intFormal()),c.locality("lClient"))
   > x > c.getInt(x,3) > sum >
   (if (y + sum ≥ 30) then
      let("store2ok")
    else
      (client.out(c.tuple("sum","store2","reqId12",y + sum),c.locality("lClient")) >> g2()) )


def g3() =
   client.in(c.tuple("store3","reqId12",c.intFormal()),c.locality("lClient"))
   > x > c.getInt(x,2) > y >
   client.in(c.tuple("sum","store3","reqId12",c.intFormal()),c.locality("lClient"))
   > x > c.getInt(x,3) > sum >
   (if (y + sum ≥ 30) then
      let("store3ok")
    else
      (client.out(c.tuple("sum","store3","reqId12",y + sum),c.locality("lClient")) >> g3()) )


addLocality()
>> (startSearch()
    >> (moveFromStore1() | moveFromStore2() | moveFromStore3() | g()) )
```

Listing 1.4. Net η_{store1}

```
nodes
shop11::{succ ~ localhost:11002,finish ~ localhost:11005, s ~ localhost:11004}
  port 11001
  begin
        out("next", succ)@self;
        out("id1",10,11)@self;
        out("id2",15,15)@self;
        out("id3",15,8)@self
  end;

shop12::{succ ~ localhost:11003,finish ~ localhost:11005, s ~ localhost:11004}
  port 11002
  begin
        out("next", succ)@self;
        out("id1",6,15)@self;
        out("id2",15,15)@self;
        out("id3",13,9)@self;
        out("id4",16,6)@self
  end;

shop13::{finish ~ localhost:11005, s ~ localhost:11004}
  port 11003
  begin
        out("next", finish)@self;
```

```
        out("id2",15,10)@self;
        out("id3",12,8)@self;
        out("id4",11,6)@self
end;

l1::{startS ~ localhost:11001}
  port 11004
  begin
        out("start",startS)@self
  end

endnodes
```

Listing 1.5. Net η_{store2}

```
nodes
shop21::{succ ~ localhost:12002,finish ~ localhost:12005, s ~ localhost:12004}
  port 12001
  begin
        out("next", succ)@self;
        out("id1",10,12)@self;
        out("id2",15,10)@self;
        out("id3",15,9)@self
  end;

shop22::{succ ~ localhost:12003,finish ~ localhost:12005, s ~ localhost:12004}
  port 12002
  begin
        out("next", finish)@self;
        out("id2",15,19)@self;
        out("id3",12,29)@self;
        out("id4",11,17)@self
end;

shop23::{finish ~ localhost:12005, s ~ localhost:12004}
  port 12003
  begin
        out("next", finish)@self;
        out("id2",15,21)@self;
        out("id3",12,10)@self;
        out("id4",11,27)@self
end;

l2::{startS ~ localhost:12001}
  port 12004
  begin
        out("start",startS)@self
  end

endnodes
```

Listing 1.6. Net η_{store3}

```
nodes
shop31::{succ ~ localhost:13002,finish ~ localhost:13005, s ~ localhost:13004}
  port 13001
  begin
        out("next", succ)@self;
        out("id1",10,12)@self;
        out("id2",15,18)@self;
        out("id3",15,15)@self
  end;

shop32::{succ ~ localhost:13003,finish ~ localhost:13005, s ~ localhost:13004}
  port 13002
  begin
        out("next", succ)@self;
        out("id1",6,11)@self;
        out("id2",15,15)@self;
        out("id3",13,29)@self;
        out("id4",16,6)@self
  end;

shop33::{finish ~ localhost:13005, s ~ localhost:13004}
  port 13003
  begin
        out("next", finish)@self;
        out("id2",15,17)@self;
        out("id3",12,23)@self;
        out("id4",11,16)@self
end;

```

```
l3::{startS ~ localhost:13001}
  port 13004
  begin
        out("start",startS)@self
  end

endnodes
```

Listing 1.7. Net η_{client}

```
nodes
lclient::{}
 port 14000
 begin
        out("client")@self
 end
endnodes
```

Listing 1.8. KLAIM processes

```
rec Find  [itemId: str, maxPrice:int, reqId:str]
  declare
        locname s,finish;
        var q,p:int;
        var locnext: phyloc
  begin
        p := 0;
        q := 0;
        read(itemId,!q,!p)@self within 200;
        if (p ≤ maxPrice AND p > 0) then
                out(itemId,q,p,reqId)@s
        endif;
        read("next",!locnext)@self;
        if (locnext != *(finish)) then
                eval(Find(itemId,maxPrice,reqId))@locnext
        endif;
        out("end search")@self
  end;


rec FindItem [itemId: str, maxPrice:int, reqId:str ]
  declare
        var locstart: loc
  begin
        read("start",!locstart)@self;
        eval(Find(itemId,maxPrice,reqId))@locstart
  end
```