A Formal Basis for Reasoning on Programmable QoS*

Rocco De Nicola[1], Gianluigi Ferrari[2], Ugo Montanari[2], Rosario Pugliese[1], and Emilio Tuosto[2]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze, Via C. Lombroso 6/17, 50134 Firenze (Italy) {denicola,pugliese}@dsi.unifi.it
[2] Dipartimento di Informatica, Università di Pisa, Via M. Buonarroti 2, 56100 Pisa (Italy) {giangi,ugo,etuosto}@di.unipi.it

Abstract. The explicit management of *Quality of Service* (QoS) of network connectivity, such as, e.g., working cost, transaction support, and security, is a key requirement for the development of the novel wide area network applications. In this paper, we introduce a foundational model for specification of QoS attributes at application level. The model handles QoS attributes as semantic constraints within a graphical calculus for mobility. In our approach QoS attributes are related to the programming abstractions and are exploited to select, configure and dynamically modify the underlying system oriented QoS mechanisms.

1 Introduction

Wide-Area Network (WAN) applications have become one of the most important classes of applications in distributed computing. Currently, Internet and World Wide Web are the primary environments for designing, developing and distributing applications. Network services have evolved into self-contained components which inter-operate easily by exploiting WEB-based access protocols [17]. In addition, network services may adapt themselves to match the particular capabilities of a variety of devices ranging from traditional PCs, to Personal Digital Assistants and Mobile Phones having intermittent connectivity to the network.

In this new scenario both final users and WAN application designers put special emphasis on Quality of Service (QoS) issues. For final users, the perceived QoS of their computations is not only dependent on the performance of WEB

* R. De Nicola has been supported by MIUR project **NAPOLI** and EU-FET project **MIKADO** IST-2001-32222. G. Ferrari has been supported by MIUR project **NAPOLI** and EU-FET project **PROFUNDIS** IST-2001-33100. U. Montanari has been supported by MIUR project **COMETA** and EU-FET project **AGILE** IST-2001-32747. R. Pugliese has been supported by MIUR project **NAPOLI** and EU-FET project **AGILE** IST-2001-32747. E. Tuosto has been supported by MIUR project **NAPOLI** and EU-FET project **PROFUNDIS** IST-2001-33100. All authors have been supported by the MIUR project **SP4** "Architetture Software ad Alta Qualità di Servizio per Global Computing su Cooperative Wide Area Networks".

servers but also on the availability of certain resources. Indeed, in addition to access services, users are allowed to control the QoS they receive. Here, QoS is meant as a measure of the *non functional* properties of services along multiple dimensions. For instance, *network bandwidth* is a QoS measure for multimedia services. *Timely response* and *security* are other examples of (higher level) QoS measures.

In general, QoS attributes are special parameters of network services. *Awareness* of these information is crucial for choosing network services to match user requirements. For instance, final users can react to network congestion by binding their network devices to different sites where the requested services are available. Similarly, QoS awareness is exploited by WAN application designers to control resource usage and resource access in order to guarantee and maintain certain security levels and to provide users with differentiated QoS.

The advances in network technologies and the growth of commercial WEB services have prompted questions about suitable mechanisms for providing QoS guarantees. In the last few years, several models have been proposed to meet the demands of QoS. We mention the *Resource Reservation Protocol* (RSVP) [6], *Differentiated Services* [5], *Constraint-based Routing* [26], and we refer to [28] for a detailed discussion of this topic. This stream of research is basically *system-oriented*: it focuses on the lower layers of the Internet protocol stack.

Another significant line of research has dealt with enhancing existing distributed programming middlewares to support QoS features. QoS-aware middlewares allow clients to express their QoS requirements at a higher level of abstraction. In this way the application has good degree of control over QoS without having to deal with low-level details. Examples of QoS-aware middleware are Agilos [22], Mobiware [1], and Globus [13].

Novel computational models exploit the idea of *network awareness* to manage the dynamic nature of network infrastructures. This has led to the development of models and foundational calculi where *mobility* is the basic notion and applications have control over localities where computation progresses. These computational models do not provide natural and expressive QoS mechanisms. Some preliminary results in this direction can be found in [7]; there a calculus is introduced which incorporates a notion of communication rate (bandwidth) and describe some programming constructs based on this calculus.

We believe that the ability of identifying and managing QoS requirements at the early stages of software development (*programmable QoS*) is a key issue and the *added-value* of the evolutionary paradigms for WAN programming. Indeed, programmable QoS allows one to evaluate the impact of QoS requirements on the overall software architectures without committing to specific low level technological details. Moreover, when implementing a specific application, this information can be used to select and configure the primitives of the underlying QoS-aware middleware.

Our research goal is to contribute at a formal understanding of programmable QoS, as a step toward the development of proof techniques and tools for the automated verification and certification of properties of WAN applications. To

achieve this, we provide an appropriate level of abstraction to describe programmable QoS together with a semantic model which can be used to experiment programmable QoS without relying on the current complex network technologies.

In particular, we have abstracted the basic features of the problem in a calculus with primitives which control explicitly QoS attributes. This calculus, that we call KAOS (KLAIM-*based calculus for Application Oriented QoS*), is a first contribution of the paper. KAOS builds on KLAIM (*Kernel Language for Agent Interaction and Mobility*) [9]. KLAIM is an experimental kernel programming language specifically designed to model and program WAN applications with located services and mobility. KLAIM naturally supports a *peer-to-peer* programming model where interacting peers (network nodes or sites, in KLAIM terminology) cooperate to provide common sets of services. KAOS enriches KLAIM networking constructs [2] with attributes which are used to specify the QoS properties of peer groups. This QoS attributes may be seen, e.g., as a value specifying the *abstract* cost c of using a given connection, or as a pair $\langle c, \pi \rangle$ that additionally specifies the set π of *access rights* (in the sense of [10]), or, more generally, as a vector whose components represent different QoS aspects.

Although KAOS provides semantic idioms to deal with programmable QoS, it does not directly handle how QoS attributes can be effectively enforced and guaranteed. To this purpose, we introduce a formal model that enables us to describe and reason about QoS guarantees. Our model is based on (hyper-)graphs and local graph synchronization and extends the graphical calculus for mobility introduced in [15]. Graphs naturally provide the capabilities to describe internetworking systems: edges represent network components and vertices model the network environments. If some edges share a vertex then the corresponding components may interact by exploiting the underlying network communication infrastructure. Graph synchronization is purely local and it is obtained by the combination of graph rewriting with constraint solving. The intuitive idea is that properties of components are specified as constraints over their local resources. Hence, local evolutions of components depend on the outcome of a (possibly distributed) constraint satisfaction algorithm. In other words, the actual behaviour is the result of a constraint solving algorithm [25, 29].

As a second contribution of the paper we provide an operational semantics for KAOS in terms of the graphical calculus. The key issue of the approach is that QoS attributes become semantic constraints of the graphical calculus. In other words, the model fosters a declarative approach by identifying the points where satisfaction of QoS requirements has a strong impact on behaviours. Hence, the goal of finding a connection between two KAOS peers with a certain QoS level corresponds to find the optimal path with respect to the QoS constraints. The main result of the paper proves that the graphical semantics provides the desired properties. In particular, we show that whenever a remote operation is performed, the graphical calculus always select the optimal path with respect to the QoS constraints set up by the application.

Throughout the paper we focus on KAOS since the specification of QoS attributes and their enforcement in the graphical semantics are specifically designed for KLAIM. However, similar techniques could be applied to other process languages in which network infrastructures can be specified declaratively.

Structure of the paper. Syntax and semantics of KAOS are given in Sections 2 and 3, respectively. An example of how an application can be modeled with KAOS is presented in Section 4. Section 5 defines hypergraphs and their semantics in terms of hyperedge replacement. A mapping of KAOS into hypergraphs and their semantics is defined in Section 6, while productions for edges used in translations of KAOS nets are detailed in Section 7. Section 8 shows how path reservation can be obtained by simple changes of productions for hypergraphs of KAOS nets. Section 9 applies the translation of Section 6 to the example in Section 4 and shows that the paths reserved by using the productions in Section 8 are the optimal paths computed by the Floyd-Warshall algorithm applied to graphs representing KAOS nets. Finally, some concluding remarks are reported in Section 10.

2 KAOS: a Calculus for Programmable QoS

This section presents the syntax of KAOS (KLAIM-*based calculus for Application Oriented QoS*) as reported in Table 1. We assume a set $\mathcal{N}$ of *names* ranged over by metavariables r, s, t, ... Names provide the abstract counterpart of the set of *communicable* values. Generally speaking, communicable values consist of expressions, processes, tuples (ordered sequences of values), and so on. For the sake of simplicity, here communicable values are simply *localities*. These are the syntactic ingredient used to express the idea of administration domain: computations at a certain locality are under the control of a specific authority. We also assume a set of *process variables* ranged over by $X, Y, \ldots$.

Finally, we assume a set of *costs* which are special values used to measure and manage QoS attributes. Cost values (ranged over by κ) are equipped with two binary operations (an additive and a multiplicative operation) together with a partial order relation $\sqsubseteq$. Formally, the algebraic structure of cost values is a *constraint semiring* [3] (or *c-semiring*, for short). An algebraic structure $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ is a constraint semiring if A is a set ($\mathbf{0}, \mathbf{1} \in A$), and $+$ and $\times$ are binary operations on A that satisfy the following properties:

- $+$ is commutative, associative, idempotent, $\mathbf{0}$ is its unit element and $\mathbf{1}$ is its absorbing element;
- $\times$ is commutative, associative, distributes over $+$, $\mathbf{1}$ is its unit element, and $\mathbf{0}$ is its absorbing element.

The additive operation of a c-semiring induces a partial order on A defined as $a \leq_A b \iff \exists c : a + c = b$. The minimal element is thus $\mathbf{0}$ and the maximal $\mathbf{1}$. Hence, $a \leq_A b$ means that a is more constrained than b.

N	::=	NETS	l	::=	LINKS
	$s ::^{L} P$	*Single node*		$\langle s, \kappa \rangle$	*Incoming link*
	$(\nu\, s)N$	*Node restriction*		$\langle \kappa, s \rangle$	*Outgoing link*
	$N_1 \parallel N_2$	*Net composition*			
			P	::=	PROCESSES
γ	::=	ACTIONS		$\mathbf{0}$	*Null process*
	(s)	*Input*		$\gamma.P$	*Action prefixing*
	$\nu(s \cdot \kappa)$	*Node creation*		$\langle t \rangle$	*Output*
	$\overset{\kappa}{\frown} s$	*Login*		$\varepsilon(P)@s$	*Process spawning*
	$s \overset{\kappa}{\smile}$	*Accept*		$P_1 \mid P_2$	*Process composition*
	$\delta\, l$	*Disconnect*		X	*Process variables*
				$rec\ X.P$	*Recursion*

Table 1. KAOS Syntax

The actual definitions of the operations of the constraint semiring depend on the notion of costs we intend to capture. For instance, the constraint semiring of truth values (the structure $(\{T, F\}, \vee, \wedge, F, T)$) allows reasoning on the availability of network connections.

The following examples give an intuition of some c-semiring structures that will be exploited in next sections.

Example 1. An example of c-semiring is the structure $\langle N, min, +, +\infty, 0 \rangle$, the c-semiring of natural numbers N where the additive operation is min (which induces the obvious order) and the multiplicative operation is the sum over natural numbers. Notice that in this case the partial order induced by the additive operations (i.e. min) is the inverse of the ordinary total order on natural numbers.

Another example is given by $\langle \wp(\{A\}), \cup, \cap, A, A\} \rangle$ where $\wp(A)$ is the powerset of a set A, and $\cup$ and $\cap$ are the usual set union and intersection operations. Notice that in the latter example also the multiplicative operation is idempotent. When this is the case, additional constraint satisfaction algorithms apply (e.g. local propagation).

KAOS programs, called nets, are the parallel composition of a set of *nodes*. A node is characterized by a unique name, representing its locality, and it is a container of resources (data) and active computational entities (processes). Syntactically, a node is written

$$s ::^{L} P$$

where s is the locality of the node, P is the process running at s, and L is the *network interface* of the node, i.e. a set of *links*. Links (ranged over by l) are pairs either of the form $\langle s, \kappa \rangle$ or of the form $\langle \kappa, s \rangle$, where s is a locality and κ is a cost. Pair $\langle s, \kappa \rangle$ ($\langle \kappa, s \rangle$) represents a link from (to) node s with costs κ. Restriction $(\nu\, s)N$ is a binder for s and is used to express the lexical scope of location s in net N. Intuitively, it is similar to local declarations inside a "block",

namely, s is a variable whose scope is local to N. Differently from declarations of usual programming languages, the scope of s can dynamically change. This is a key feature of name passing process calculi (the well known example is the π-calculus [24]) formally defined as *scope extrusion*.

KAOS processes are built up from the special process $\mathbf{0}$, that does not perform any action, and from a set of basic actions by using action prefixing, parallel composition and recursion. Inter-processes communication is local. The output process $\langle s \rangle$ makes available name s in the local repository. Intuitively, the output operation abstracts the idea of *publishing* a service (a data) into a directory (the repository). The input action (s) withdraws a name from the local repository and uses it to replace the formal name s in the rest of the process; if no name is available, the executing process is blocked. In our analogy, input abstracts the idea of resource *discovery*. The operation for creating a new node $\nu(s \cdot \kappa)$ has the effect of establishing a link with cost attribute κ between the creating node and the fresh node (otherwise, the created node would be unreachable, i.e. completely useless). The only possibly remote operation is $\varepsilon(P)@s$ that provides code mobility. The execution of $\varepsilon(P)@s$ has the effect of sending process P for execution at the node s. Process $P_1 \mid P_2$ is the parallel composition of processes P_1 and P_2, namely P_1 and P_2 are executed concurrently. Process $rec\, X.\, P$ is a recursive process. It is equivalent to execute the process obtained from P once process variable X has been replaced by its definition. Variable X can be renamed without affecting the behaviour of $rec\, X.\, P$. Indeed, if Y does not occur in P, $rec\, Y.\, P[^Y/_X]$ is equivalent to $rec\, X.\, P$ ($[^Y/_X]$ denotes the substitution that replace occurrences of X with Y in P if they are not in the scope of a $rec\,.$ binder).

Actions *login*, *accept* and *disconnect* allow specifying the characteristic of connections among nodes. A login operation sends to the receiver a message detailing the QoS attributes of the required connection. The receiver may accept a login request by performing the accept operation. If a login request is accepted a link with the specified QoS attributes is set up among the two nodes involved. The disconnect operation removes the link with the specified QoS attributes from the network interface of a node.

Names occurring in KAOS processes and nets can be *bound*. More precisely, prefix $(s).P$ binds s in P; this prefix is similar to the λ-abstraction of the λ-calculus. Prefix $\nu(s \cdot \kappa).P$ binds s in P and, similarly, $(\nu\, s)N$ binds s in N. A name that is not bound is called *free*. The sets $\mathrm{bn}(\cdot)$ and $\mathrm{fn}(\cdot)$ (respectively, of bound and free names of a term) are defined in Table 2 (where $\mathrm{fn}(L)$ denotes the names occurring in the set of links L). The set $\mathrm{n}(\cdot)$ of names of a term is the union of its sets of free and bound names. We say that two terms are *α-equivalent*, $\equiv_\alpha$, if one can be obtained from the other by renaming bound names.

Hereafter, we will identify KAOS nets which intuitively represent the same net. We therefore define *structural congruence* $\equiv$, namely an equivalence relation over nets that equates terms denoting the same net and differ only for meaningless syntactic details. Relation $\equiv$ relates nets and relies on a relation $\equiv_p$ which defines structural congruence over processes and is defined as:

γ	fn()	bn()
(s)	$\emptyset$	$\{s\}$
$\nu(s \cdot \kappa)$	$\emptyset$	$\{s\}$
$\overset{\kappa}{\frown} s$	$\{s\}$	$\emptyset$
$s \overset{\kappa}{\smile}$	$\{s\}$	$\emptyset$
$\delta \langle \kappa, s \rangle$	$\{s\}$	$\emptyset$
$\delta \langle s, \kappa \rangle$	$\{s\}$	$\emptyset$

P	fn()	bn()
$\mathbf{0}$	$\emptyset$	$\emptyset$
$\gamma.P$	$\text{fn}(P) \setminus \text{bn}(\gamma) \cup \text{fn}(\gamma)$	$\text{bn}(P) \cup \text{bn}(\gamma)$
$\langle s \rangle$	$\{s\}$	$\emptyset$
$\varepsilon(P)@s$	$\text{fn}(P) \cup \{s\}$	$\text{bn}(P)$
$P_1 \mid P_2$	$\text{fn}(P_1) \cup \text{fn}(P_2)$	$\text{bn}(P_1) \cup \text{bn}(P_2)$
X	$\emptyset$	$\emptyset$
$rec\, X.\, P$	$\text{fn}(P)$	$\text{bn}(P)$

N	fn()	bn()
$s ::^L P$	$\{s\} \cup \text{fn}(L) \cup \text{fn}(P)$	$\text{bn}(P)$
$N_1 \parallel N_2$	$\text{fn}(N_1) \cup \text{fn}(N_2)$	$\text{bn}(N_1) \cup \text{bn}(N_2)$
$(\nu\, s)N$	$\text{fn}(N) \setminus \{s\}$	$\text{bn}(N) \cup \{s\}$

Table 2. Free and bound names

- $P \mid \mathbf{0} \equiv_p P$, for any P;
- $P \mid Q \equiv_p Q \mid P$, for any P and Q;
- $P \mid (Q \mid R) \equiv_p (P \mid Q) \mid R$, for any P, Q and R.

The axioms above state that $(P, \mid, \mathbf{0})$ is a *commutative monoid*.

We can define net structural equivalence as the smallest relation over nets such that

1. $\parallel$ is a commutative and associative;
2. $\dfrac{N \equiv_\alpha N'}{N \equiv N'}$;
3. $\dfrac{P \equiv_p Q}{s ::^L P \equiv s ::^L Q}$.

Notice that $\equiv$ identifies only nets whose equality derives from their syntactical structure and has nothing to do with the semantics of nets (which has still to be introduced and shall rely on structural congruence). With a slight abuse of notation, in the following we write $P \equiv Q$ instead of $P \equiv_p Q$; the context will always clarify whether $\equiv$ is the relation on nets or on processes.

A net $s_1 ::^{L_n} P_1 \parallel ... \parallel s_n ::^{L_n} P_n$ is *well-formed* if, and only if, for any i, j, if $i \neq j$ then $s_i \neq s_j$ and if $\langle \kappa, s_j \rangle \in L_i$ ($\langle s_j, \kappa \rangle \in L_i$) then $\langle s_i, \kappa \rangle \in L_j$ ($\langle \kappa, s_i \rangle \in L_j$). Notice that this definition implies that, in well-formed nets, a connection from s to t costing κ is possible only if two links, $\langle t, \kappa \rangle$ and $\langle \kappa, s \rangle$ are in the network interfaces of nodes s and t, respectively. We shall only consider well-formed nets.

3 An LTS Semantics for KAOS

This section presents the operational semantics of KAOS as a standard labeled transition system semantics. For simplicity, we write $M \cup e$ (resp. $M \setminus e$) instead of $M \cup \{e\}$ (resp. $M \setminus \{e\}$).

The semantics is given in terms of a transition relation that describes possible net evolutions and the corresponding abstract costs (we omit the cost when it is negligible).

Definition 1 ($\xrightarrow[\rho]{\alpha}\!\!\!\!\rightarrow$). *The LTS semantics of* KAOS *is the minimal relation* $\longrightarrow \subseteq N \times \langle \alpha, \rho \rangle \times N$ *closed under the inference rules of Tables 3 and 4 and rule*

$$\text{(struct)} \qquad \frac{N_1 \equiv N' \xrightarrow[\rho]{\alpha}\!\!\!\!\rightarrow N'' \equiv N_2}{N_1 \xrightarrow[\rho]{\alpha}\!\!\!\!\rightarrow N_2}.$$

Labels of transition $\longrightarrow$ *are defined as follows:*

$$\begin{array}{l} \alpha ::= \tau \;\mid\; s \lhd t \;\mid\; s \rhd t \;\mid\; s(\eta,\, P)@t \;\mid\; s \,\varepsilon\, t \;\mid\; X@s \\ \quad \mid\; s \overset{\kappa}{\frown} t \;\mid\; s \overset{\kappa}{\smile} t \;\mid\; \delta(s, \langle t, \kappa \rangle) \;\mid\; \delta(s, \langle \kappa, t \rangle) \end{array}$$

$$\rho ::= \epsilon \;\mid\; s, \kappa$$

In the operational rules we adopt a notational convention borrowed from a similar notation for terms. In particular,

- $\mathrm{bn}(\alpha)$ is equal to η if $\alpha = s(\eta,\, P)@t$, $\emptyset$ otherwise,
- $\mathrm{n}(\alpha)$ ($\mathrm{n}(\rho)$ resp.) denotes the set of all names (free and bound) occurring in α (ρ resp.).

Finally, we will write $\kappa \models T(P)$ to indicate that the behaviour of P is compatible with the cost κ. Intuitively, $T(P)$ represents the *type* or *capabilities* of process P. Like for abstract costs, we intentionally do not specify what $T(P)$ exactly is. For example, it could be determined by using a type system like the one in [10]. From a pragmatic point of view, $T(P)$ is a parameter used to discriminate between processes that can be executed at a site and processes that cannot.

Let us now comment on the semantics. Labels α are used to describe process activities.

- τ describes internal activity.
- $s \lhd t$ ($s \rhd t$) says that a process located at s aims at receiving (sending) a name t.
- $s(\eta,\, P)@t$ says that a process at s intends spawning process P for execution at t. Set η contains the localities occurring in P that must be restricted upon migration (because their scope must also include the target node t).
- $s \,\varepsilon\, t$ says that a process from s is migrated to t.
- $X@s$ says that node s does exist and can accept a process for execution at s; variable X is the placeholder where the migrating process that reaches s will be substituted for.
- $s \overset{\kappa}{\frown} t$ says that a process at s intends to establish a link between s and t with cost κ.
- $s \overset{\kappa}{\smile} t$ says that a process at s accepts the establishment of a link between s and t with cost κ.

(out)	$s ::^L \langle t \rangle \xrightarrow{s \rhd t} s ::^L \mathbf{0}$
(in)	$s ::^L (x).P \xrightarrow{s \lhd t} s ::^L P[^t/_x]$
(leval)	$s ::^{L \cup \langle s,\kappa \rangle} \varepsilon(P)@s \xrightarrow{\tau} s ::^{L \cup \langle s,\kappa \rangle} P, \quad$ if $\kappa \models T(P)$
(eval)	$s ::^L \varepsilon(P)@t \xrightarrow[s,1]{s(\emptyset,\ P)@t} s ::^L \mathbf{0}, \quad$ if $s \neq t$
(new)	$s ::^L (\nu(x \cdot \kappa).P) \mid Q \xrightarrow{\tau} (\nu\, x)(s ::^{L \cup \langle \kappa,x \rangle} P \mid Q \parallel x ::^{\langle s,\kappa \rangle} \mathbf{0})$, if $x \notin \mathrm{n}(L) \cup \{s\} \cup \mathrm{fn}(Q)$
(llogin)	$s ::^{L\ \kappa\frown} s.P \xrightarrow{\tau} s ::^{L \cup \{\langle s,\kappa \rangle, \langle \kappa,s \rangle\}} P$
(login)	$s ::^{L\ \kappa\frown} t.P \xrightarrow{s\ \kappa\frown\ t} s ::^{L \cup \langle \kappa,t \rangle} P, \quad$ if $s \neq t$
(accept)	$s ::^L t\ \overset{\kappa'}{\smile}.P \xrightarrow{t\ \overset{\kappa}{\smile}\ s} s ::^{L \cup \langle t,\kappa \rangle} P, \quad$ if $\kappa \leq \kappa'$
(ldisc)	$s ::^L \delta\langle s,\kappa \rangle.P \xrightarrow{\tau} s ::^{L \setminus \langle s,\kappa \rangle \setminus \langle \kappa.s \rangle} P$
(idisc)	$s ::^L \delta\,\langle t,\kappa \rangle.P \xrightarrow{\delta(s,\langle t,\kappa \rangle)} s ::^{L \setminus \langle t,\kappa \rangle} P, \quad$ if $t \neq s$
(odisc)	$s ::^L \delta\,\langle \kappa,t \rangle.P \xrightarrow{\delta(s,\langle \kappa,t \rangle)} s ::^{L \setminus \langle \kappa,t \rangle} P, \quad$ if $t \neq s$
(node)	$s ::^{L \cup \langle r,\kappa \rangle} P \xrightarrow[r,\kappa]{X@s} s ::^{L \cup \langle r,\kappa \rangle} P \mid X, \quad$ if X fresh
(rec)	$s ::^L rec\ X.P \xrightarrow{\tau} s ::^L P[^{rec\ X.P}/_X]$

Table 3. Axioms of KAOS interactive semantics

– $\delta(s, \langle t, \kappa \rangle)$ ($\delta(s, \langle \kappa, t \rangle)$) says that a process running at s wants to remove the link $\langle t, \kappa \rangle$ ($\langle \kappa, t \rangle$) from the network interface of node s.

Labels ρ can be either the empty label ϵ (that carries no information and will be omitted) or the pair $\langle s, \kappa \rangle$ which point out that s is used as gateway for a remote action at node s and κ is the cost of the path from the source node to s.

Axioms in Table 3 describe process activities. The informal semantics has been explained in the previous section. Here we only add a few comments.

– Axiom (in) gives an "early" flavor to the semantics (in the sense of the π-calculus [24]).
– Axioms (leval) and (eval) deal with process spawning. The first one says that local evaluation is authorized only if the type of the process is compatible

with the cost of performing a local spawning. The second axiom accounts for remote evaluation.
- Axioms (llogin) and (login) deal with the establishment of a new link. A local link can always be established, while a remote one needs the authorization of the target node. In both cases, the node network interface is enriched with an outgoing link.
- Axiom (accept) says that any connection request having a cost κ less than κ' can be accepted. The node network interface is enriched with an incoming link.
- Axioms (ldisc), (idisc) and (odisc) handle requests for removing local, incoming and outgoing links, respectively. Removing local links simply updates the network interface of a node, whereas, removing a link from/to a remote node t requires to signal node t that the corresponding link must be removed from its network interface (see rule (remlink) in Table 4).
- Axiom (node) says that a node s dynamically creates an execution context named (by the process variable) X where a (possibly remote) process can be placed. Other than X, the transition label contains the locality r of a node that can be used as an intermediate node (a sort of *gateway*) to reach s (because there exists a link from r to s).

Rules in Table 4 coordinate the behaviour of processes in a net. Most of the rules (e.g. (par1), (par2) and (res)) are standard. Here, we comment on a few peculiarities.

- Rule (com) says that communication is always local and asynchronous.
- Rule (connect) says that in order to establish a link, a synchronization must occur between the requiring process and the accepting one.
- Rule (remlink) says that to remove a link, a synchronization between the two nodes connected by the link is necessary in order to guarantee that the network interfaces of these two nodes will be updated coherently. Notice that this synchronization occurs provided that the link actually exists.
- Scope extrusion of bound names that are exported by migrating processes is implemented by rules (open) and (close) in the style of the π-calculus [24].
- Rules (route) and (close) check step-by-step the existence of a path of links from the node s (performing the remote spawning of process P) to the target node t. In particular, the first premise of these rules checks the existence of a path from s to an intermediate gateway r'. The second premise checks the existence of a link starting from r'. If both checks succeed, and if the type of P complies with the cost of each link along the path, the path can be extended by including the link from r' (which is used as an intermediate node). Additionally, in rule (route) the obtained path connects s to an $r \neq t$ (see the reduction labels in the conclusion of the rule), hence the free context X of r is not used to execute P and must be removed. This is done by simply plugging process $\mathbf{0}$ inside X and by exploiting the fact that $Q \mid \mathbf{0} \equiv \mathbf{0}$ and rule (struct). On the other hand, rule (close) obtains a path from s to t, hence P is now plugged in the execution context X of t (i.e. P is sent for execution

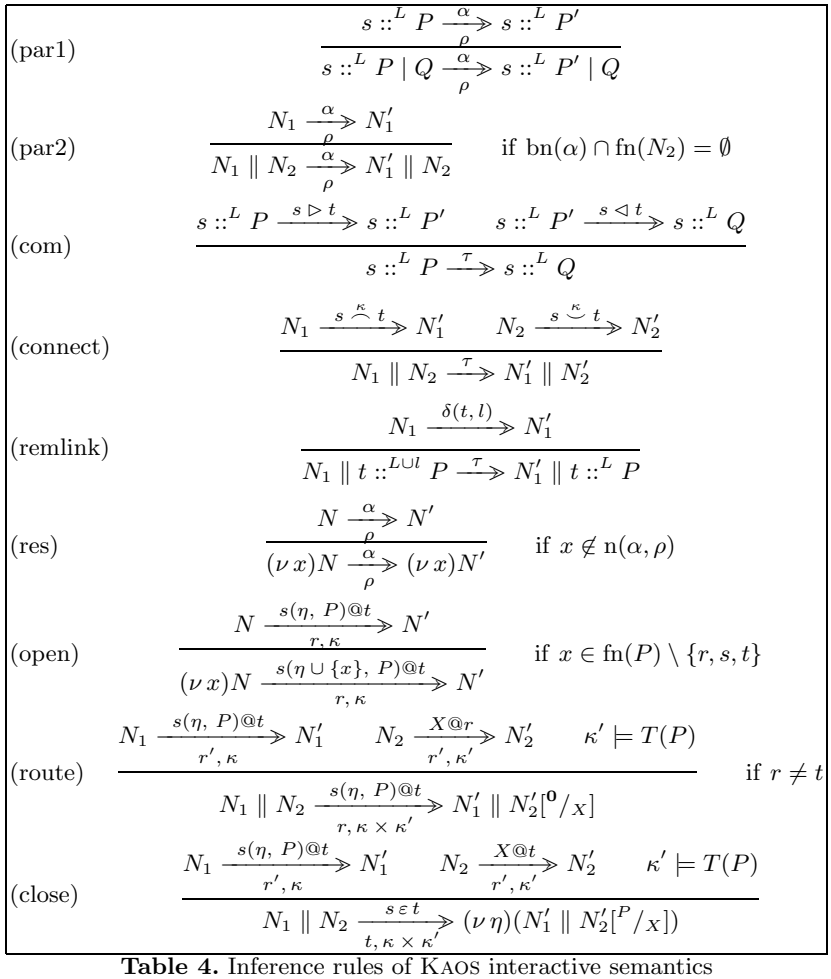


Table 4. Inference rules of KAOS interactive semantics

at t). Moreover, the restrictions over the bound names carried along with P are restored to extend the scope of these names to the resulting net as a whole. An example of use of these two rules is given in Figure 3.

4 A KAOS Application

This section shows how KAOS can be used to model QoS requirements at application level. The scenario is a messaging application. A group of nodes are connected and have to exchange messages (or files). When a message must be sent from node s to node t a notification message is first sent to t. At this point, node t spawns an agent on s that will examine the message and, if all checks are satisfied, the message is effectively moved on t. Intuitively, the agent acts as a "filter" that can be programmed to avoid useless messages to roam the net.

This mechanism also enhances network performances when messages are much greater than notifications and agents. The above message exchange protocol is depicted in Figure 1 where the bold line represents the fact that an agent is spawned from t to s and the dashed line represents the fact that the message is downloaded only if the checking phase is passed.

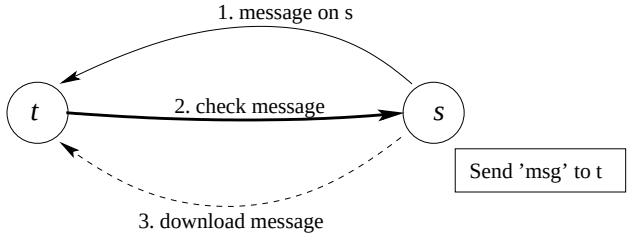


Fig. 1. The filter protocol between s and t

Nodes s and t may not be directly connected but can be connected by means of (a number of) links passing through a number of intermediate nodes that are used for forwarding messages between t and s. We assume that the attributes of a link between r and r' that can affect our messaging application could be

1. the geographical distance between r and r';
2. the capabilities granted to processes executed at r' from r (at r from r');
3. the price of the connection.

Figure 2 depicts a possible way of connecting s and t and the costs associated to the links. Lines which are arrowed at both extremities, see e.g. the line between t and x, represent two links, one from t to x and another from x to t, both links have the same cost (e.g. $\langle 10, \{i, o\}, 2\rangle$). Notice that, in the net of Figure 2, t is connected to s via two paths (the path $t - x - z - s$ and the path $t - y - z - s$), while only one path goes from s to t.

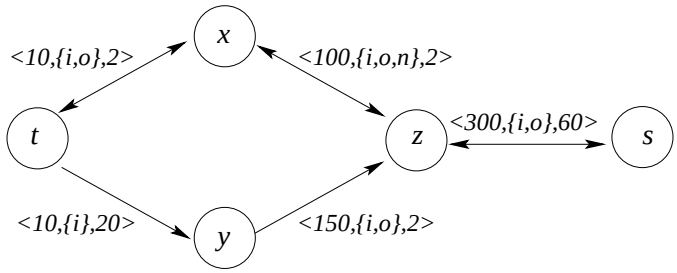


Fig. 2. A net connecting s and t

We now show how KAOS primitives can be exploited both for declaring the network connections and for programing the described messaging application.

Costs First we describe costs as triples $\kappa = \langle d, C, p\rangle$ where

1. d is the geographical distance (in Km);
2. $C \subseteq \{i, o, n\}$ are the capabilities, where i, o and n stand for $i$nput, $o$utput and creation of $n$ew nodes, respectively;
3. p is the price (in euros).

All the components of costs are elements of the following c-semirings, respectively

1. $(N, min, +, +\infty, 0)$,
2. $(\wp(\{i, o, n\}), glb, \cap, \{i, o, n\}, \{i, o, n\})$,
3. $(Q, min, +, +\infty, 0)$.

In [3] it has been proved that the cartesian product of c-semirings is a c-semiring as well. Therefore, we can define operations $\times$ and $+$ of the cost c-semiring as

$$\langle d, C, p\rangle \times \langle d', C', p'\rangle = \langle d + d', C \cap C', p + p'\rangle$$
$$\langle d, C, p\rangle + \langle d', C', p'\rangle = \langle min\{d, d'\}, glb\{C, C'\}, \min\{p, p'\}\rangle.$$

Accordingly, the neutral elements of $\times$ and $+$, respectively are defined as $1 = \langle 0, \{i, o, n\}, 0\rangle$ and $0 = \langle +\infty, \emptyset, +\infty\rangle$.

In the following, we use the convention that κ_{uv} denotes the cost of the link from u to v. In Figure 2, for instance, κ_{xt} is $\langle 10, \{i, o\}, 2\rangle$. We also avoid writing trailing occurrences of $\mathbf{0}$, hence $\gamma.\mathbf{0}$ will be abbreviated as γ.

Connections In order to establish connections among nodes t, x, y, z and s corresponding to Figure 2, we can start with the following KAOS net:

$$t ::^{\emptyset} P_t \parallel x ::^{\emptyset} P_x \parallel y ::^{\emptyset} P_y \parallel z ::^{\emptyset} P_z \parallel s ::^{\emptyset} P_s$$

where

$$P_t \triangleq x \overset{\kappa_{xt}}{\smile} \mid \overset{\kappa_{tx}}{\frown} x \mid \overset{\kappa_{ty}}{\frown} y$$
$$P_x \triangleq t \overset{\kappa_{tx}}{\smile} \mid z \overset{\kappa_{zx}}{\smile} \mid \overset{\kappa_{xt}}{\frown} t \mid \overset{\kappa_{xz}}{\frown} z$$
$$P_y \triangleq t \overset{\kappa_{ty}}{\smile} \mid \overset{\kappa_{yz}}{\frown} z$$
$$P_z \triangleq x \overset{\kappa_{xz}}{\smile} \mid y \overset{\kappa_{yz}}{\smile} \mid s \overset{\kappa_{sz}}{\smile} \mid \overset{\kappa_{zx}}{\frown} x \mid \overset{\kappa_{zs}}{\frown} s$$
$$P_s \triangleq z \overset{\kappa_{zs}}{\smile} \mid \overset{\kappa_{sz}}{\frown} z.$$

According to the semantics of KAOS, after all executions of login and accept actions, the network interfaces of the nodes in the resulting net will correspond

to the graph of Figure 2 and the KAOS net obtained is the following:

$$\begin{aligned}
&t ::^{\{\langle x,\kappa_{xt}\rangle,\langle\kappa_{tx},x\rangle,\langle\kappa_{ty},y\rangle\}}\ \mathbf{0}\ \|\\
&x ::^{\{\langle t,\kappa_{tx}\rangle,\langle z,\kappa_{zx}\rangle,\langle\kappa_{xt},t\rangle,\langle\kappa_{xz},z\rangle\}}\ \mathbf{0}\ \|\\
&y ::^{\{\langle t,\kappa_{ty}\rangle,\langle\kappa_{yz},z\rangle\}}\ \mathbf{0}\ \| \qquad (1)\\
&z ::^{\{\langle x,\kappa_{xz}\rangle,\langle y,\kappa_{yz}\rangle,\langle s,\kappa_{sz}\rangle,\langle\kappa_{zx},x\rangle,\langle\kappa_{zs},s\rangle\}}\ \mathbf{0}\ \|\\
&s ::^{\{\langle z,\kappa_{zs}\rangle,\langle\kappa_{sz},z\rangle\}}\ \mathbf{0}.
\end{aligned}$$

Capabilities In order to consider the intentions of remotely evaluated processes it is necessary to define $T(P)$. Given a process P, we define $T(P)$ as follows

$$T(P) = \begin{cases} \emptyset, & \text{if } P = \mathbf{0} \vee P = \varepsilon(Q)@t \vee P = X \\ \{o\}, & \text{if } P = \langle t\rangle \\ T(P_1) \cup T(P_2), & \text{if } P = P_1 \mid P_2 \\ T(Q), & \text{if } P = rec\ X.Q \\ T(\gamma) \cup T(Q), & \text{if } P = \gamma.Q \end{cases}$$

where

$$T(\gamma) = \begin{cases} \{i\}, & \text{if } \gamma = (x) \\ \{n\}, & \text{if } \gamma = \nu(x \cdot \kappa) \\ \emptyset, & \text{otherwise.} \end{cases}$$

Capabilities of processes that are arguments of spawning actions performed by P are not part of the type of P; instead, they will be looked at when the spawning actions are effectively executed (see rules (route) and (close) in Table 4).

We can now instantiate relation $\models$ to costs and process types defined in this section. Given a process P, we say that a cost $\langle d, C, e\rangle$ *satisfies* $T(P)$ (written $\langle d, C, e\rangle \models T(P)$) if, and only if, $T(P) \subseteq C$. This interpretation states that a remote evaluation of a process P can traverse a link if all the capabilities that P might exercise occur in the cost of the link.

Sender, receiver and filter processes The messaging application can be defined by means of a sender process executed at t, a receiver process executed at s and the agent that filters the file which migrates from t to s and provides the result to t. This three rôles can be formalized in KAOS[1] as follows:

$$\begin{aligned}
S &\triangleq \varepsilon(\langle \text{“}bigfile'', s\rangle)@t \mid \langle \text{“}bigfile'', file\rangle\\
R &\triangleq (f,u).\varepsilon(F_{f,t})@u.(res)...\\
F_{f,t} &\triangleq (f,v).\mathbf{if}\ test(v)\ \mathbf{then}\ \varepsilon(\langle v\rangle)@t\ \mathbf{else}\ \varepsilon(\langle no\rangle)@t
\end{aligned}$$

[1] In this example we use tuples, ground values as string or files, the boolean function *test* and the if-then-else construct. In the formal definition of KAOS we chose not to have many constructs and types in order to give a smoother definition of the calculus. It is, of course, straightforward to extend KAOS to all programming constructs used in our running example.

Process S (allocated at s) notifies to R (allocated at t) that a "big file" is available at s (spawning action of S). When R acquires the notification, the filter F is spawned at u and the result of its check is waited for. Agent F, once in execution, accesses the file, and tests whether it must be sent to t or not. In the first case, the file is effectively sent to t while, in the second case, a conventional signal *no* is sent.

If S and R are respectively allocated at nodes s and t of net (1) we can detail how KAOS semantics can deal with the remote operations and costs of connections. In order to avoid cumbersome replica of link costs, in the following we use L_u to denote the network interface of node u. Hence, we consider the initial configuration as given in net (1)

$$t ::^{L_t} R \quad \| \quad x ::^{L_x} \mathbf{0} \quad \| \quad y ::^{L_y} \mathbf{0} \quad \| \quad z ::^{L_z} \mathbf{0} \quad \| \quad s ::^{L_s} S \tag{2}$$

and show how KAOS semantics can determine a path connecting t and s and, for each path, its total cost. For instance, let us consider the spawning action of S. By definition $T = T(\langle \text{“}bigfile'', s\rangle) = \{o\}$.

$$\underbrace{\dfrac{s ::^{L_s} \varepsilon(\langle \text{“}bigfile'', s\rangle)@t \xrightarrow[s,1]{s(\emptyset,\ \langle \text{“}bigfile'', s\rangle)@t} s ::^{L_s} \mathbf{0}}{s ::^{L_s} S \xrightarrow[s,1]{s(\emptyset,\ \langle \text{“}bigfile'', s\rangle)@t} s ::^{L_s} \langle \text{“}bigfile'', file\rangle}}_{eval-s-z}$$

$$\underbrace{\dfrac{eval - s - z \quad z ::^{L_z} \mathbf{0} \xrightarrow[s,\kappa_{sz}]{X@z} z ::^{L_z} X \quad \kappa_{sz} \models T}{s ::^{L_s} S \parallel z ::^{L_z} \mathbf{0} \xrightarrow[z,1\times\kappa_{sz}]{s(\emptyset,\ \langle \text{“}bigfile'', s\rangle)@t} s ::^{L_s} \langle \text{“}bigfile'', file\rangle \parallel z ::^{L_z} \mathbf{0}}}_{path-s-z}$$

$$\underbrace{\dfrac{path - s - z \qquad x ::^{L_x} \mathbf{0} \xrightarrow[z,\kappa_{zx}]{Y@x} x ::^{L_x} Y \qquad \kappa_{zx} \models T}{N_1 \xrightarrow[x,1\times\kappa_{sz}\times\kappa_{zx}]{s(\emptyset,\ \langle \text{“}bigfile'', s\rangle)@t} N_2}}_{netpath-s-z}$$

$$\dfrac{netpath - s - z \quad t ::^{L_t} R \xrightarrow[x,\kappa_{xt}]{Z@t} t ::^{L_t} R \mid Z \qquad \kappa_{xt} \models T}{N_1 \parallel t ::^{L_t} R \xrightarrow[t,1\times\kappa_{sz}\times\kappa_{zx}\times\kappa_{xt}]{s\ \varepsilon\ t} N_2 \parallel t ::^{L_t} R \mid \langle \text{“}bigfile'', s\rangle}$$

Fig. 3. Proving a net reduction

An example of inference proof that uses the KAOS semantics rules can be found in Figure 3 where, for the sake of readability, we use the following shorthands: N_1 stands for the net $s ::^{L_s} S \parallel z ::^{L_z} \mathbf{0} \parallel x ::^{L_x} \mathbf{0}$ and N_2 for the net

$s ::^{L_s} \langle \text{“}bigfile'', file\rangle \parallel z ::^{L_z} \mathbf{0} \parallel x ::^{L_x} \mathbf{0}$. The first inference is an application of rule (par1) that uses axiom (eval) as premise. The second inference, uses the conclusion of the first one (that we call $eval - s - z$) and axiom (node) for the premises, and then proceeds by first applying (route) and (struct). The resulting net reduction (that we call $path - s - z$ because determines a path from s to z) is then used as the first premise of the third inference proof. This inference also uses axiom (node) as premise and proceeds by applying (route) and (struct). Finally, the resulting net reduction (that we call $netpath - s - z$) and axiom (node) are the premises of the last inference that concludes by applying rule (close).

5 A Calculus of Graphs

Graph-based techniques can be usefully adopted for modeling inter-networking systems. Indeed, *(hyper)edges* (namely, edges that connect more than two vertices) can be used to represent components, while vertices model the network environment of components. Edges sharing a vertex means that the corresponding components may interact by exploiting network communication infrastructure. Structured versions of graphs (typed graphs, term graphs, hierarchical graphs) can precisely model complex inter-network configurations [14, 15] and access control policies [18–20].

Graph synchronization adds to network awareness the ability of dealing with the temporal dimension of computations. Graphs synchronization is purely local and it is obtained by the combination of graph rewriting with constraint solving. The intuitive idea is that local rewritings depends on the outcome of a (possibly global) constraint satisfaction algorithm.

Graph rewriting based on edge replacement and synchronization was introduced in [8, 11] and related to distributed constraint satisfaction problems in [25]. The version with mobility, which employs a notation based on logical judgments and inference rules, was introduced recently in [14] and extended in [15] to encode the π-calculus. Abstract semantics based on bisimilarity has been discussed in [21].

Next sections introduce syntax and semantics of graphs.

5.1 Syntax of Graphs

We assume that $\mathcal{V}$ is an ordered set of vertices. An *edge*, is an atomic item with a label from a ranked alphabet $\mathcal{L}$. We write $L(x_1, ..., x_n)$ to indicate an edge labeled L connecting vertices x_1,..., x_n. In this case, we say that L has rank n (written as $L : n$) and that $x_1, ..., x_n$ are the *attachment vertices* (or *attachment points*) of L. Figure 5.1 represents edge $L(a, b, c)$ where $L : 3$. Wires connecting vertices a, b and c to L are called tentacles. *Graphs* are built from ranked edges in $\mathcal{L}$ and *vertices* in $\mathcal{V}$. Moreover, we write $L(\boldsymbol{x})$ with the implicit assumption that L has rank $|\boldsymbol{x}|$, namely the length of vector $\boldsymbol{x}$.

We can now define graphs.

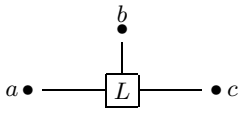


Fig. 4. An edge

Definition 2 (Syntactic judgments). *A* (hyper)graph *is a* syntactic sequent *of the form* $\Gamma \vdash G$*, where*

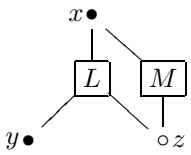
- $\Gamma \subseteq \mathcal{V}$ *is a finite set of vertices called* external vertices*, and*
- G *is one of the terms generated by the following grammar, where* $y \in \mathcal{V}$ *and* $L : |\boldsymbol{x}|$ *is an edge:*

$$G ::= L(\boldsymbol{x}) \mid G \mid G \mid \nu\, y.G \mid nil.$$

We call terms G graph terms*.*

The productions in Definition 2 permits generating single edges ($L(\boldsymbol{x})$), composing terms in parallel ($G \mid G$) and hiding vertices ($\nu\, y.G$). Graph *nil* represents the empty graph. We will use $\mathrm{fv}(G)$ to denote the set of the vertices of G which does not occur in the scope of a ν operator. Hereafter, we omit the curly brackets in judgments and write $x_1, ..., x_n \vdash G$ instead of $\{x_1, ..., x_n\} \vdash G$; moreover, we will often use $\boldsymbol{x} \vdash G$ instead of $x_1, ..., x_n \vdash G$, if $\boldsymbol{x} = (x_1, ..., x_n)$, . We use the notation Γ, x instead of $\Gamma \cup \{x\}$ with the implicit assumption that $x \notin \Gamma$; similarly, we write Γ_1, Γ_2 instead of $\Gamma_1 \cup \Gamma_2$ assuming that $\Gamma_1 \cap \Gamma_2 = \emptyset$.

Example 2. Let us consider the judgment $x, y \vdash \nu\, z.(L(y, z, x) \mid M(x, z))$, where $L : 3$ and $M : 2$; a graphical representation of the judgment is



where filled circles and empty circles are used for representing free and restricted vertices, respectively.

Definition 3 gives the structural congruence rules for graph terms. We take advantage of such congruence to avoid writing cumbersome parenthesis.

Definition 3 (Structural Congruence). *The* structural congruence *is the smallest binary relation* $\equiv$ *over graph terms that obeys axioms in Table 5.*

Axioms (*AG1*), (*AG2*) and (*AG3*) define associativity, commutativity and identity over *nil* for operation $|$, respectively. Axioms (*AG4*) and (*AG5*) state that the vertices of a graph can be restricted in any order and that restriction does not play any rôle on non-free vertices of a graph, respectively. Axiom

$(AG1)$	$(G_1 \mid G_2) \mid G_3 \equiv G_1 \mid (G_2 \mid G_3)$	
$(AG2)$	$G_1 \mid G_2 \equiv G_2 \mid G_1$	
$(AG3)$	$G \mid nil \equiv G$	
$(AG4)$	$\nu\ x.\nu\ y.G \equiv \nu\ y.\nu\ x.G$	
$(AG5)$	$\nu\ x.G \equiv G,$	if $x \notin \mathrm{fv}(G)$
$(AG6)$	$\nu\ x.G \equiv \nu\ y.G\{y/x\},$	if $y \notin \mathrm{fv}(G)$
$(AG7)$	$\nu\ x.(G_1 \mid G_2) \equiv (\nu\ x.G_1) \mid G_2,$	if $x \notin \mathrm{fv}(G_2)$

Table 5. Graphs structural congruence rules

$(AG6)$ deals with alpha conversion of hidden bound vertices, while $(AG7)$ tunes the interplay between hiding and the operator for parallel composition. Occasionally, taking advantage of axiom $(AG4)$, we will write $\nu\ X$, with $X = \bigcup x_i$, to abbreviate $\nu\ x_1.\nu\ x_2 \ldots \nu\ x_n$.

We will work with *well-formed judgments*.

Definition 4 (Well-Formed Judgments). *A judgment is* well-formed *if it is generated by applying the rules in Table 6 up to structural congruence.*

$$\frac{}{x_1, \ldots, x_n \vdash nil}\ (RG1) \qquad \frac{\Gamma \vdash G_1 \qquad \Gamma \vdash G_2}{\Gamma \vdash G_1 \mid G_2}\ (RG3)$$

$$\frac{L : m \qquad y_1, \ldots, y_m \in \{x_1, \ldots, x_n\}}{x_1, \ldots, x_n \vdash L(y_1, \ldots, y_m)}\ (RG2) \qquad \frac{\Gamma, x \vdash G}{\Gamma \vdash \nu\ x.G}\ (RG4)$$

Table 6. Well-formed judgments

Rule $(RG1)$ states that graphs with n isolated vertices and no edges are well-formed. Rule $(RG2)$ states that a graph with n vertices and one edge L (having sort m) whose tentacles are all connected to vertices of the graph is well-formed. Notice that $(RG2)$ does not make any assumption on n and m, hence it could be the case that $m \geq n$. This implies that some tentacles of L can be connected to the same vertex. Rule $(RG3)$ allows one to put together (using $|$) two well-formed judgments that share the same set of external vertices. Finally, rule $(RG4)$ permits hiding a vertex from the environment.

The correspondence theorem expressing that well-formed judgments up to structural axioms are isomorphic to graphs up to isomorphism has been proved in [15].

5.2 Graph Rewritings

We propose a new rewriting mechanism for graphs that permits interconnection modification and relies on edge replacement. The idea is that an edge can be rewritten if the *constraints* it imposes on its external vertices accomplish with constraints of all other edges connected to such vertices. What here is meant for "constraint" is a pair consisting of a label and a tuple of vertices. The label in a constraint is an action of a *synchronization algebra*. A synchronization algebra is a structure $(Act,\ S : Act \times Act \to Act)$ where Act is a set of actions and S is a binary function on Act. If $a = S(a_1, a_2)$ then we say that the a is the action obtained by synchronizing a_1 and a_2.

Observation 1 *The principal and well studied synchronization algebras are* á la *Milner (CCS [23], where* $S(a, \bar{a}) = \tau$*), and* á la *Hoare (CSP [16], where* $S(a, a) = a$*) synchronizations [27]. The former takes two partner to synchronize trough complementary actions, while the latter requires that all participants of a synchronization perform the same action. Hereafter, we consider synchronizations* á la *Milner.*

As will be clarified later, constraints are used to "select" which adjacent edges in a graph must be replaced in a graph rewriting.

Figure 5 aims at giving a graphical intuition of edge replacement. Edge L

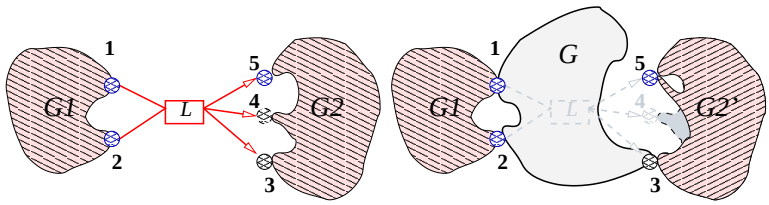


Fig. 5. Hyperedge replacement

in Figure 5(a) is connected to graphs $G1$ and $G2$, indeed external vertices **1** and **2** are attachment points of both L and $G1$, while vertices **3**, **4** and **5** are attachment points of both L and $G2$.

Figure 5(b) represents the graph obtained by replacing edge L with graph G. The dashed gray part of the Figure 5(b) represents the initial situation which disappears after the transition has taken place.

The main things to remark are that (i) $G1$ is not modified after the transition; (ii) all vertices in G different from **1** and **2** are new vertices generated by the transition; (iii) some vertices can be "fused" after the transition as **4** and **5** in Figure 5(b). As will be shown later, this accounts for *mobility* of components, that dynamically can change their connections.

5.3 Productions

A *graph rewriting system*, $\mathcal{G} = \langle \Gamma_0 \vdash G_0, \mathcal{P} \rangle$, consists of an initial graph $\Gamma_0 \vdash G_0$ and a set of *productions*:

Definition 5 (Production). *Let $X \subseteq \mathcal{V}$ be the set $\{x_1, \ldots, x_n\}$ and L be an edge label with rank n. A* production *is a transition of the form*

$$X \vdash L(x_1, \ldots, x_n) \xrightarrow[\pi]{\Lambda} \Gamma \vdash G, \tag{3}$$

where

- *function $\pi : X \to X$ is a* fusion substitution*;*
- *$\Lambda \subseteq X \times Act \times \mathcal{V}^*$ is a* set of constraints*;*
- *$\Gamma = \pi(X) \cup (\mathrm{v}(\Lambda) \setminus X)$;*
- *$\mathrm{fv}(G) \subseteq \Gamma$.*

Production (3) specifies the constraints that the environment must satisfy in order to replace $L(\boldsymbol{x})$ with G. Such constraints are imposed by Λ on the set X of external vertices of L. Once constraints in Λ are satisfied, vertices must be coalesced according to fusion substitution π. Λ and π are detailed below.

Function π is a fusion substitution if

$$\forall x_i, x_j \in X. \pi(x_i) = x_j \Rightarrow \pi(x_j) = x_j,$$

namely π induces an equivalence relation partition $\simeq_\pi$ over X defined as $x \simeq_\pi x' \stackrel{\text{def}}{\iff} \pi(x) = \pi(x')$. Equivalence $\simeq_\pi$ partitions X into equivalence classes where each vertex $x \in X$ is mapped to a *representative element* $\pi(x)$.

Λ associates actions in Act and sequences of vertices to (some of the) external vertices of L. Λ is the graph relation of a partial function with (finite) domain X and codomain in $Act \times \mathcal{V}^*$. Given Λ, we indicate the set of constraints of a vertex x with $\Lambda(x)$. If $(x, a, \boldsymbol{y}) \in \Lambda$ then L can synchronize with edges in the environment that have a tentacle connected to x and satisfy condition a (that will depend on the chosen synchronization algebra). Intuitively, in order to perform a transition, all conditions on external vertices must be in accordance with the synchronization policy. Thus, actions in Act constraint the possible synchronizations among connected edges.

Let us again consider constraint $(x, a, \boldsymbol{y})$[2]; vector $\boldsymbol{y}$ contains the vertices of the constraint; we let $\mathrm{v}(\Lambda)$ denote the union of the vertices of the constraints

[2] We assume that any label $a \in Act$ has an *arity*; we let $|\cdot| : Act \to \omega$ be the arity function on Act. Arities of actions are needed to maintain consistent constraints on vertices. More precisely $|a| = |\boldsymbol{y}|$, for each constraint $(x,\ a,\ \boldsymbol{y})$.

in Λ. Vector $\boldsymbol{y}$ either contains vertices that appear in X or new vertices that will be present in $\Gamma \vdash G$. We impose a further condition on productions, indeed we require that $\mathrm{v}(\Lambda) \cap X \subseteq \pi(X)$; namely, the external vertices used in the synchronization must be representative elements according to $\simeq_\pi$.

Let us now consider the structure of the right hand side of judgment (4). Γ consists of the vertices which are image of $x_1, \ldots, x_n$ through π and the new vertices used in the synchronization, namely those vertices that appear in Λ and are not in X. In general, G may be any graph provided that $\mathrm{fv}(G) \subseteq \Gamma$.

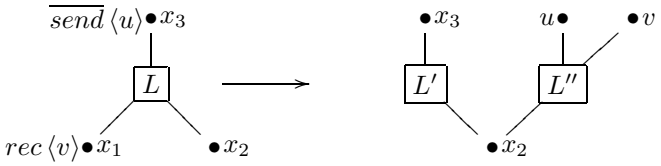
Synchronized edge replacement is obtained by graph rewriting combined with constraint solving. More specifically, we use *context-free* productions labeled with actions useful for coordinating the simultaneous application of two or more productions. Coordinated rewriting allows the propagation of synchronization all over the graph where productions are applied. Determining the productions to synchronize at a given stage corresponds to solving a distributed constraint satisfaction problem [25].

Example 3. Referring to Example 2, let us assume that the following production is given:

$$x_1, x_2, x_3 \vdash L(x_1, x_2, x_3) \xrightarrow[[x_2/x_1]]{\left\{\begin{array}{c}(x_3,\ \overline{send},\ \langle u\rangle),\\ (x_1,\ rec,\ \langle v\rangle)\end{array}\right\}} x_2, x_3, u, v \vdash L'(x_3, x_2) \mid L''(u, x_2, v).$$

The above production states that, once constraints on vertices x_1 and x_3 are satisfied by the environment, edge L is replaced by two edges: L' and L''. L' has tentacles to vertices x_2 and x_3, while L'' is connected to x_3 and to two newly generated vertices u and v. Fusion substitution $[x_2/x_1]$ represents the mapping $\begin{cases} x_1 \mapsto x_1 \\ x_2 \mapsto x_1 \\ x_3 \mapsto x_3 \end{cases}$ and determines the partition $\{\{x_1, x_2\}, \{x_3\}\}$, where x_2 is the representative element of $\{x_1, x_2\}$.

The production can be graphically represented as follows:



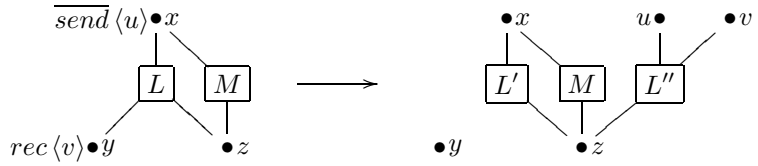
5.4 Edge Replacements

A production rewrites a single edge into an arbitrary graph. Before giving the formal definition of edge replacement we describe an intuitive procedure that can be naively regarded as a procedural way of obtaining edge replacement.

A production $p : L \to R$ can be applied to a graph G yielding H if there exists an occurrence of an edge labeled by L in G. Graph H is obtained from G by

1. removing the occurrence of L,
2. embedding a fresh copy of R in G and
3. coalescing external vertices of R with the corresponding attachment vertices of the occurrence of edge L.

Example 4. If we apply the production defined in Example 3 to the graph of Example 2, i.e. to the judgment $x, y, z \vdash L(y, z, x) \mid M(x, z)$, we obtain the following graph rewriting:



As already stated, it is not mandatory that *all* edges take place in replacements, namely, some components can remain *idle* while others are replaced.

Graphs over edge labels $\mathcal{L}$ and vertices $\mathcal{V}$ obey the usual structural congruence axioms in the same style of Section 2; in particular, given a production

$$\boldsymbol{x} \vdash L(\boldsymbol{x}) \xrightarrow[\pi]{\Lambda} \Gamma \vdash G,$$

renaming can be applied in several ways:

i. external vertices of $\boldsymbol{x}$ can be changed throughout the judgment;
ii. vertices declared in $\mathrm{v}(\Lambda) - \Gamma$ can be α-converted;
iii. the representative vertices chosen by π can be consistently changed.

5.5 Transitions of Graphs

Productions are synchronized via the inference rules in Table 7. Graph semantics is based on productions to specify edge replacement, while inference rules essentially synchronize productions and confer dynamic behaviour to graphs.

A transition is a logical judgment

$$\Gamma_1 \vdash G_1 \xrightarrow[\pi]{\Lambda} \Gamma_2 \vdash G_2 \tag{4}$$

where Λ, π, Γ_2 and G_2 obeys the same conditions imposed on productions. Essentially, transitions can be seen as productions having general graphs on their left hand side. Hence transitions describe the dynamic evolutions of graphs.

Transition (4) states that $\Gamma_1 \vdash G_1$ can take part to rewritings that match constraints Λ and determine fusion substitution π. Once such conditions are satisfied, $\Gamma_1 \vdash G_1$ rewrites as $\Gamma_2 \vdash G_2$.

Definition 6 (Graph transitions). *Let $\langle \Gamma_0 \vdash G_0, \mathcal{P}\rangle$ be a graph rewriting system. The set of* transitions *$T(\mathcal{P})$ is the smallest set that contains $\mathcal{P}$ and that is closed under the four inference rules in Table 7.*

$$(merge1)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma' \vdash G' \qquad \Lambda(y) = \emptyset \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad \rho = [^{\pi(x)}/_{\pi(y)}]}{\Gamma \vdash [^x/_y]G \xrightarrow[(\pi;\rho)_{-y}]{\rho\Lambda}\!\!\gg \mathrm{v}(\rho\Lambda) \cup (\pi;\rho)_{-y}(\Gamma) \vdash \rho G'}$$

$$(merge2)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda \cup \{(x,a,\boldsymbol{v}),(y,\overline{a},\boldsymbol{w})\}}\!\!\gg \Gamma' \vdash G' \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad \rho = mgu\{[^{[^x/_y]\boldsymbol{w}}/_{[^x/_y]\boldsymbol{v}}], [^{\pi(x)}/_{\pi(y)}]\} \qquad \Gamma'' = \mathrm{v}(\rho\Lambda) \cup (\pi;\rho)_{-y}(\Gamma) \qquad U = \rho(\Gamma') \setminus \Gamma''}{\Gamma \vdash [^x/_y]G \xrightarrow[(\pi;\rho)_{-y}]{(\rho\Lambda \cup (x,\tau,\langle\rangle))}\!\!\gg \Gamma'' \vdash \nu\, U.\rho G'}$$

$$(res)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma' \vdash G' \qquad \Lambda(y) = \emptyset \vee \Lambda(y) = \{(y,\tau,\langle\rangle)\} \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad U = \Gamma' \setminus (\mathrm{v}(\Lambda) \cup \pi_{-y}(\Gamma))}{\Gamma \vdash \nu\, y.G \xrightarrow[\pi_{-y}]{\Lambda \setminus (y,\tau,\langle\rangle)}\!\!\gg \mathrm{v}(\Lambda) \cup \pi_{-y}(\Gamma) \vdash \nu\, U.G'}$$

$$(par)\quad \frac{\Gamma_1 \vdash G_1 \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma_2 \vdash G_2 \qquad \Gamma_1' \vdash G_1' \xrightarrow[\pi']{\Lambda'}\!\!\gg \Gamma_2' \vdash G_2' \qquad \Gamma_1 \cap \Gamma_1' = \emptyset}{\Gamma_1 \cup \Gamma_1' \vdash G_1 \mid G_1' \xrightarrow[\pi \cup \pi']{\Lambda \cup \Lambda'}\!\!\gg \Gamma_2 \cup \Gamma_2' \vdash G_2 \mid G_2'}$$

Table 7. Inference rules for graph synchronization

A derivation is obtained by starting from the initial graph and by executing a sequence of transitions, each obtained by synchronizing productions. The synchronization of rewriting rules requires matching of the actions and unification of the third components of the constraints Λ. After productions are applied, the unification function is used to obtain the final graph by merging the corresponding vertices.

In Table 7 we use notation $[^{v_1,\ldots,v_n}/_{u_1,\ldots,u_n}]$ (abbreviated as $[^{\boldsymbol{v}}/_{\boldsymbol{u}}]$) to denote substitutions that are applied both to graphs and sets of constraints. If $\rho = [^{\boldsymbol{v}}/_{\boldsymbol{u}}]$ is a substitution then ρG is the graph obtained by substituting all free occurrences of u_i with v_i in G for each $i = 1, ..., n$, while $\rho\Lambda = \{(x, a, \rho\boldsymbol{y}) : (x, a, \boldsymbol{y}) \in \Lambda\}$ where $\rho\boldsymbol{y}$ is the vector whose components result from applying ρ to the corresponding components of $\boldsymbol{y}$.

Finally, given a function $f : A \to B$ and $y \in A$, $f_{-y} : A \setminus y \to B$ is defined as $f_{-y}(x) = f(x)$, for all $x \in A \setminus y$.

The most important rules in Table 7 are ($merge1$) and ($merge2$). They regulates how vertices can be fused. Rule ($merge1$) fuses two vertices provided that no constraint is required on one of them, whereas rule ($merge2$) handles with vertices upon which complementary actions are required. Rule (res) describes how graph transitions can be performed under vertex restriction. Finally, rule (par) states how transitions on disjoint graphs can be combined together.

Let us comment more on all the rules.

Rule ($merge1$) fuses vertex x and y provided that no constraint is imposed on y (i.e. $\Lambda(y) = \emptyset$) and that x and y are equivalent according to π. Premise $x \simeq_\pi y \Rightarrow \pi(y) \neq y$ imposes that, when y is fused with a different equivalent vertex x, then y must not be the representative element. A transition from $\Gamma, y \vdash G$ may be re-formulated to obtain the transition where y and x are coalesced, provided that fusion of their representative elements, ρ, is reflected on Λ, on π and on continuation $\Gamma' \vdash G'$. Indeed, if y is fused with x, also the other vertices equivalent to them are fused; the fusion substitution in the conclusion of ($merge1$) is $\pi;\rho$ (restricted to Γ), all occurrences of $\pi(y)$ are replaced with $\pi(x)$ in $\mathrm{v}(\Lambda)$ and the final graph is $\rho G'$. It is obtained by merging $\pi(y)$ and $\pi(x)$ in G'.

Rule ($merge2$) synchronizes complementary actions. The rule permits merging x and y in a transition where they offer complementary non-silent actions. As for ($merge1$), x cannot replace the representative element of its equivalence class. Most general unifier ρ takes into account possible equalities due to the transitive closure of substitutions $[^{\boldsymbol{v}}/_{\boldsymbol{u}}]$ after $[^x/_y]$ has been applied. ρ fuses the corresponding vertices of the constraints and propagates previous fusions π. The resulting constraints $\rho\Lambda \cup \{(x, \tau, \langle\rangle)\}$ does not change constraints offered on vertices different from x and y (up to the necessary fusion ρ). Fusion substitution $(\pi;\rho)_{-y}$ acts on Γ by applying ρ. Finally, nodes U are the restricted nodes of $\rho G'$ and are those nodes that neither are in $(\pi;\rho)_{-y}(\Gamma)$ nor are generated by $\rho\Lambda$. This corresponds to the *close* rule of the π-calculus.

Finally, vertices U are the restricted vertices in $\rho G'$ and are those vertices that are neither in $(\pi;\rho)_{-y}(\Gamma)$ nor are generated by $\rho\Lambda$.

Rule (res) deals with vertex restriction. Representative elements cannot be restricted if other vertices are in their equivalence class. Furthermore, only vertices can be restricted where either a synchronization action takes place or no constraint is imposed. If those conditions hold, the (possible) silent action on y is hidden and vertices not in $\Gamma' \setminus (\mathrm{v}(\Lambda) \cup \pi_{-y}(\Gamma))$ are restricted.

Rule (par) simply combines together disjoint judgments. Function $\pi \cup \pi'$ applied to a vertex x is $\pi(x)$ or $\pi'(x)$ depending on $x \in \Gamma'$. Note that $\pi \cup \pi'$ is well defined because $\Gamma \cap \Gamma' = \emptyset$.

6 KAOS Translation

In this section, by exploiting the graphical calculus, we define an alternative semantics for KAOS which takes care of QoS attributes. We first present a trans-

lation scheme from KAOS nets and processes to the graphical calculus, then we present the productions of edges used in the translation.

Our translation relies on two particular edges: *node edges* and *link edges*. A node edge $\mathfrak{S}^s$ models KAOS node s while link edge G_t^κ represents a link to node t with cost κ. Moreover, we use a distinguished vertex $\diamond$ to represent the communication infrastructure used to interact with other node edges. In this work we simply represent the communication infrastructure with a special vertex, however, in general this layer could be arbitrarily complex; for instance it could be an ethernet or an internet connection. The only assumption on $\diamond$ is that it must be able to connect any two node edges, indeed $\diamond$ will be exploited to establish links among KAOS nodes.

It is worth to remark that $\diamond$ does not play any rôle in managing application QoS features (indeed, in our framework, virtual networks are built over an underlying physical net[3]) and QoS attributes are established by applications and are not directly related to the underlying communication infrastructure.

The mapping function $[\![\_]\!]$ associates a graph to a well-formed KAOS net. Function $[\![\_]\!]$ is defined by induction on the syntactical structure of KAOS nets. The most important case is the translation of a KAOS node $s ::^L P$, where $L = \{\langle s_1, \kappa_1\rangle, \ldots, \langle s_m, \kappa_m\rangle, \langle \kappa_1, t_1\rangle, \ldots, \langle \kappa_n, t_n\rangle\}$. Since $s ::^L P$ is part of a well formed net, $L^s = \{l : l = \langle s, \kappa\rangle \in L\}$ and $L_s = \{l : l = \langle \kappa, s\rangle \in L\}$ are in bijective correspondence. We assume fixed a bijective function $\lambda : L_s \to L^s$. We define a set of vertices Γ containing vertex $\diamond$ and a vertex for each link occurrence in L: Let $\Gamma = \{u_1, \ldots, u_m, v_1, \ldots, v_n, \diamond\}$ (hereafter, we write $\boldsymbol{u}$ in place of $u_1, \ldots, u_m$ and $\boldsymbol{v}$ in place of $v_1, \ldots, v_n$). Then $\Gamma \setminus \diamond$ is in bijective correspondence with L and we say that u_i *corresponds* to $\langle s_i, \kappa_i\rangle$ $(i = 1, \ldots, m)$ and that v_j *corresponds* to $\langle \kappa_j, t_j\rangle$ $(j = 1, \ldots, n)$.

$$[\![s ::^L P]\!] \;=\; \pi(\Gamma \vdash (\nu\, \boldsymbol{x}, p)([\![P]\!]_p \mid \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \mid \prod_{j=1}^{n} G^{\kappa_j}_{t_j}(x_j, v_j))) \qquad (5)$$

where $\boldsymbol{x}$ is a vector of n pairwise distinct vertices, one for each outgoing link in L and $\pi : \Gamma \to \Gamma$ is a fusion substitution such that π is the identity for all vertices which do not correspond to links in L_s, whereas for all $v \in \Gamma$ that corresponds to $\langle \kappa, s\rangle \in L_s$, $\pi(v) = u$ iff u corresponds to l' and $\lambda(\langle \kappa, s\rangle) = l'$. In other words, π opportunely connects the outgoing link edges that connects s with itself. Notice that, depending on the chosen λ, π changes, hence the translation depends on λ. However, the obtained graphs are equivalent in the sense that they have the same behaviour up-to renaming of external vertices.

The graph associated to $s ::^L P$ contains an edge $\mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond)$ representing node s. Vertices in $\boldsymbol{x}$ are used to connect link edges $G^{\kappa_j}_{t_j}$ to the node edge. A graphical representation is given in Figure 6. The graph representing process P allocated at s is connected to $\mathfrak{S}^s$ on vertex p which is used for synchronizing $\mathfrak{S}^s$ with local processes. In some sense, edge $\mathfrak{S}^s$ is the coordinator of node s and interfaces incoming links, processes executed at s and links departing from s. The dotted tentacles in Figure 6 aim at remarking that each edge $G^{\kappa'_i}_s$, corresponding

[3] This is a typical *peer-to-peer* fashion of coordinating distributed computations.

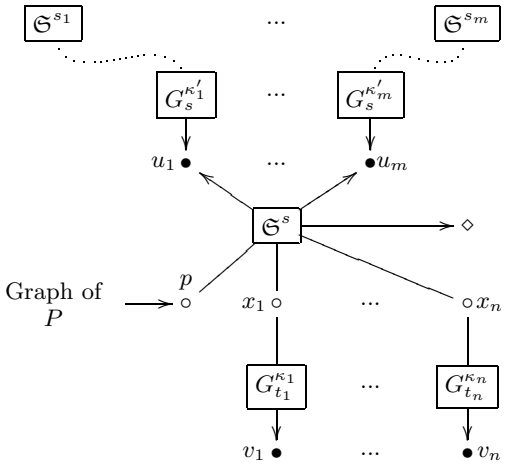


Fig. 6. Graphs for KAOS nodes

to a link from s_i to s, is connected to a restricted node shared with a tentacle of the node edge of s_i.

Net $N_1 \| N_2$ is mapped into a graph obtained by juxtaposing the graphs of the constituent nets, N_1 and N_2 and opportunely connecting their link edges.

$$[\![N_1 \| N_2]\!] = \pi(\Gamma_1, \Gamma_2 \vdash G_1 \mid G_2), \qquad \text{if } [\![N_i]\!] = \Gamma_i \vdash G_i,\ i = 1, 2$$

Function $\pi : \Gamma_1, \Gamma_2 \to \Gamma_1, \Gamma_2$ is fusion substitution that plays that same rôle as in translation of a single done. Indeed, since $N_1 \parallel N_2$ is a well-formed net, there is a bijective correspondence between outgoing links in the network interface of nodes in N_1 and incoming links of nodes in N_2 (and viceversa). Hence, if $v \in \Gamma_1$ ($v \in \Gamma_2$) is a vertex corresponding to a link $\langle \kappa, t \rangle$ in the network interface of node s in N_1 (N_2) and $u \in \Gamma_2$ ($u \in \Gamma_1$) is the vertex corresponding to $\langle s, \kappa \rangle$, then $\pi(v) = u$.

Restriction of nodes is trivially translated according to the following clause:

$$[\![(\nu\, s)N]\!] = \Gamma \setminus \boldsymbol{u} \setminus \boldsymbol{v} \vdash (\nu\, \boldsymbol{u}, \boldsymbol{v}).G, \qquad \text{if } [\![N]\!] = \Gamma \vdash G$$

where, $\boldsymbol{u}$ and $\boldsymbol{v}$ are the vertices where incoming and outgoing link edge of $\mathfrak{S}^s$ are respectively connected, if $\mathfrak{S}^s$ occurs in G and are empty vectors otherwise. The graph of a net with a restricted node name, $(\nu\, s)N$, is computed by first translating N and then restricting all vertices corresponding to node s.

The mapping for processes is described by the equations below:

$$
\begin{aligned}
[\![\ \mathbf{0}\ ]\!]_p &= nil \\
[\![\ \langle t \rangle\ ]\!]_p &= L_{\langle t \rangle}(p) \\
[\![\ \gamma.P\ ]\!]_p &= L_{\gamma.P}(p) \\
[\![\ \varepsilon(P)@s\ ]\!]_p &= (\nu\, u)(\varepsilon_s^{T(P)}(u,p) \mid S_P(u)) \\
[\![\ P_1 \mid P_2\ ]\!]_p &= [\![\ P_1\ ]\!]_p \mid [\![\ P_2\ ]\!]_p \\
[\![\ rec\, X.\, P\ ]\!]_p &= [\![\ P[^{rec\, X.\, P}/_X]\ ]\!]_p.
\end{aligned}
$$

The graph of a process P has an outgoing tentacle toward its execution vertex. The graph relative to the empty process simply is the empty graph; tuple processes and action prefixing are mapped to edges attached to p and labeled with the process. Translation of $\varepsilon(P)@s$ consists of two edges connected through the (hidden) vertex u. On the one hand, edge $\varepsilon_s^T(P)$ is connected to vertex p and handles migration of S_P to its destination node; on the other hand, P cannot be translated as a normal process because it must be executed when the migration has taken place. Hence, edge S_P is used; as will be clear once productions will be specified, S_P remains "idle" until the destination node is reached and at that time, P will be executed on the arrival node. The parallel processes are mapped to the union of the graphs of their parallel components; finally, recursive processes are translated by translating the unfolded process. It is worth to remark that we consider only "guarded" recursion, namely we require that in $rec\, X.\, P$ any process variable is in the scope of a prefix action. This implies that translation always terminates.

The following property holds for the presented translation functions:

Theorem 1. *If N is a well-formed* KAOS *net, then for each link edge $G_s^\kappa(\_, u)$ in $[\![\ N\ ]\!]$ there is a (unique) node edge $\mathfrak{S}_{m,n}^s(\boldsymbol{u}, \_, \_, \diamond)$ in $[\![\ N\ ]\!]$ such that u appears exactly once in $\boldsymbol{u}$.*

7 Productions for KAOS

As anticipated in Section 1, graphs permit path reservation. However, we prefer to introduce first productions which do not consider path reservation, but are more strictly related to KAOS semantics and, later on, we show how a path can be reserved and traversed.

We distinguish between *activity* and *coordination* productions. Indeed, it is necessary to coordinate node, link and process edges in order to detect the best path connecting two vertices. Hence, we separate the presentation of activity and coordination productions in different sections. Section 7.1 describes the activity productions necessary for executing KAOS actions; Section 7.2 and Section 7.3 respectively report coordination productions for node edges and for link edges.

7.1 Activity Productions

Activity productions for KAOS deal with actions for accessing tuple spaces, for managing links, for creating nodes and for spawning processes on remote nodes. Let us consider actions for accessing tuple spaces; the productions for the corresponding edges are:

$$p \vdash L_{(x).P}(p) \xrightarrow{\{(p,\ in\, t,\ \langle\rangle)\}} p \vdash [\![\ P[^{t}/_{x}]\ ]\!]_p$$

$$p \vdash L_{\langle t \rangle}(p) \xrightarrow{\{(p,\ \overline{in\, t},\ \langle\rangle)\}} p \vdash nil$$

The above productions state that edges corresponding to input actions wait on the vertex p for synchronizing with the production of an output. Notice that, in the rhs of the last production the edge corresponding to the output process is removed.

An edge corresponding to $\nu(r \cdot \kappa).P$ synchronizes with its node edge in order to acquire the connection to the net ($\diamond$) and make the node edge of P to add a link to r.

$$p \vdash L_{\nu(r\cdot\kappa).P}(p) \xrightarrow{\{(p,\ \overline{ns},\ \langle y, z\rangle)\}} p, y, z \vdash (\nu\ q, u)([\![\ P\ ]\!]_p \mid \mathfrak{S}^{r}_{1,0}(u, q, z) \mid G^{\kappa}_{r}(y, u)).$$

The above production "reads" into z the attach point to the net; during the synchronization, the node edge also generates a new vertex y, used to connect the outgoing link to vertex u. Production (11) is the complementary of the above production.

Creation of new links requires to synchronize process and node edges.

$$p \vdash L_{\overset{\kappa}{\frown} x.P}(p) \xrightarrow{\{(p,\ \overline{x\ log\ \kappa},\ \langle\rangle)\}} p \vdash [\![\ P\ ]\!]_p \tag{6}$$

Production (6) sends a signal for creating a new link with cost κ to (the node edge of) x.

The accept action can be similarly handled:

$$p \vdash L_{x \overset{\kappa}{\smile} .P}(p) \xrightarrow{\{(p,\ \overline{x\ acc\ \kappa},\ \langle\rangle)\}} p \vdash [\![\ P\ ]\!]_p.$$

According to the above production, the process simply "says" to its node edge that is willing to *acc*ept a request of connection from vertex x with cost κ.

Let x and κ respectively be the node and the cost of a link l; the following productions manage disconnection of l:

$$p \vdash L_{\delta\, l.P}(p) \xrightarrow{\{(p,\ \overline{\kappa\ det\ x},\ \langle\rangle)\}} p \vdash [\![\ P\ ]\!]_p \tag{7}$$

$$p \vdash L_{\delta\, l.P}(p) \xrightarrow{\{(p,\ \overline{x\ det\ \kappa},\ \langle\rangle)\}} p \vdash [\![\ P\ ]\!]_p \tag{8}$$

Production (7) removes outgoing links, while incoming links are removed by production (8).

Remote process evaluation is managed by the productions for edge $\varepsilon_s^T(u,p)$. Vertex u is used to connect the "quiescent" process that must be spawned

$$u, p \vdash \varepsilon_s^T(u,p) \xrightarrow{\{(r,\ \overline{ev\ T\ s},\ \langle\rangle)\}} u, p \vdash \varepsilon'_s(u,p).$$

The previous production asks for a path to s that can be exploited to move a process with capability T. When the path is found, an $s\,\kappa$ signal is received (where $\kappa \neq 0$):

$$u, p \vdash \varepsilon'_s(u,p) \xrightarrow{\{(p,\ s\,\kappa,\ \langle y\rangle),\ (u,\ \overline{run},\ \langle y\rangle)\}} u, p, y \vdash nil; \tag{9}$$

vertex y represents the p-vertex of the remote node. Simultaneously, the quiescent process connected to u is waken by the action $\overline{run}$.

$$u \vdash S_Q(u) \xrightarrow{\{(u,\ run,\ \langle y\rangle)\}} u, y \vdash [\![\ Q\ ]\!]_y. \tag{10}$$

Finally, the quiescent process S_Q starts its execution when it synchronizes with its ε edge on vertex y. This corresponds to move the process Q to the target node.

7.2 Productions for Nodes

Productions for node edges must coordinate the activity of processes. We start with the simplest case. If a process wants to create a new node, then the node edge can immediately synchronize by sending vertex $\diamond$, namely the net connection where the new vertex must be attached. The following production formalizes what informally stated:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{(p,\ ns,\ \langle y, \diamond\rangle)\}} \boldsymbol{x}, y, p, s, \diamond \vdash \mathfrak{S}^s_{m,n+1}(\boldsymbol{u}, y, \boldsymbol{x}, p, \diamond). \tag{11}$$

Notice that production (11) adds a tentacle to the node edge and connects it to the newly generated vertex y which is also returned to the process waiting on p (see productions 9 and 10).

A slightly more complex production is required for handling new link creation:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\Lambda} \boldsymbol{u}, \boldsymbol{x}, p, \diamond, y \vdash \nu\, z.(\mathfrak{S}^s_{m,n+1}(\boldsymbol{u}, z, \boldsymbol{x}, p, \diamond) \mid G^\kappa_y(z,y)),$$

where $\Lambda = \{(p,\ \ t\ log\,\kappa,\ \ \langle\rangle), (\diamond,\ \ t\ acc\,\kappa,\ \ \langle y\rangle)\}$. The intuition is that, when a process asks to its node edge for a new link to node t with attributes κ (action on p), then the node edge synchronizes (over $\diamond$) with the node edge at t that must *acc*ept the connection. The new link edge is connected to vertex z and reaches vertex y. A simpler production is for accepting new links:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\left\{\begin{smallmatrix}(p,\ t\ acc\,\kappa,\ \langle\rangle),\\ (\diamond,\ \overline{t\ acc\,\kappa},\ \langle z\rangle)\end{smallmatrix}\right\}} \boldsymbol{u}, \boldsymbol{x}, z, p, \diamond \vdash \mathfrak{S}^s_{m+1,n}(z, \boldsymbol{u}, \boldsymbol{x}, p, \diamond).$$

Indeed, the node edge must simply forward the *acc* signal to the net.

Finally, incoming link disconnection simply requires to forward the *det* signal to the incoming links of node s

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{\begin{smallmatrix}(p,\ t\ det\ \kappa,\ \langle\rangle),\\ (\boldsymbol{u},\ \overline{t\ det\ \kappa},\ \langle\rangle)\end{smallmatrix}\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond).$$

The above production synchronizes with an incoming link edge that will disappear.

Removing an outgoing link is more complex because the node edge must first find which is the tentacle to remove. Hence, the node edge forwards the disconnection signal (received on p) to its link edges:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{\begin{smallmatrix}(p,\ \kappa\ det\ t,\ \langle\rangle),\\ (\boldsymbol{x},\ \overline{\kappa\ det\ t},\ \langle\rangle)\end{smallmatrix}\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}'^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond).$$

Edge $\mathfrak{S}'$ waits for the links to determine whether they must disconnect or not:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}'^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\begin{smallmatrix}(x_1,\ nodet,\ \langle\rangle),\\ \{\ldots(x_i,\ det,\ \langle\rangle),\ldots\}\\ (x_n,\ nodet,\ \langle\rangle)\end{smallmatrix}} \boldsymbol{u}, \boldsymbol{z}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n-1}(\boldsymbol{u}, \boldsymbol{z}, p, \diamond)$$

where $\boldsymbol{z} = x_1, ..., x_{i-1}, x_{i+1}, ..., x_n$. The link that replies with the signal (*det*) disappears (see production (15)) while all other edges remain connected to the router.

The last task of node edges is the search of paths for remote process spawning. First, when a process asks for a path to node t such that a process with capabilities T can roam the path (action $(p,\ ev\ T\ t,\ \langle\rangle)$), the signal ev is forwarded to the outgoing links:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{\begin{smallmatrix}(p,\ ev\ T\ t,\ \langle\rangle),\\ (\boldsymbol{x},\ \overline{ev\ T\ t},\ \langle\rangle)\end{smallmatrix}\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \overline{F}^{s\ \varepsilon\ t}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, p, \diamond)$$

where the signal sent over $\boldsymbol{x}$ contains the type of the migrating process T and the target node t. As formally stated in the next section, link edges forward signals $ev\ T\ t$ received from their node edge to the remote node edge they are connected to. Hence, node edges must synchronize with link edges and make the request traverse the net:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{\begin{smallmatrix}(\boldsymbol{u},\ ev\ T\ t,\ \langle\rangle),\\ (\boldsymbol{x},\ \overline{ev\ T\ t},\ \langle\rangle)\end{smallmatrix}\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash F^{s\ \varepsilon\ t}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \boldsymbol{u}, \diamond).$$

Edge F and $\overline{F}$ have similar productions, the only difference being that $\overline{F}$ forwards search results on vertex p, while F sends them to the incoming links connected to $\boldsymbol{u}$. Therefore, in the following we consider only productions for F. When costs are communicated to F, it starts to forward them to links.

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash F^{s\ \varepsilon\ t}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \boldsymbol{u}, \diamond) \xrightarrow{\begin{smallmatrix}(x_1,\ t\ \kappa_1,\ \langle y_1\rangle), ...,\\ \{\ (x_n,\ \overline{t\ \kappa_n},\ \langle y_n\rangle),\ \}\\ (\boldsymbol{u},\ \overline{t\ \kappa_h},\ \langle y_h\rangle)\end{smallmatrix}} \boldsymbol{u}, \boldsymbol{x}, p, \boldsymbol{y}, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond)$$

where $\kappa_h = \kappa_1 + ... + \kappa_n$.

Finally, node edges communicate their p-vertex when incoming links require them with an *eval* action (see page 31):

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{(u_i,\ eval,\ \langle p \rangle)\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond).$$

7.3 Productions for Links

Whenever a link $G^\kappa_s(x, v)$ receives a message for searching a path to a vertex t ($t \neq s$) suitable for a process with capabilities T, then it forwards the signal, provided that $\kappa \models T$:

$$x, v \vdash G^\kappa_s(x, v) \xrightarrow{\{(x,\ ev\ T\ t,\ \langle\rangle),\ (v,\ \overline{ev\ T\ t},\ \langle\rangle)\}} x, v \vdash \widehat{G}^{t,\kappa}_s(x, v).$$

$\widehat{G}^{t,\kappa}_s(x, v)$ waits on v for the cost κ' of the path from s to t and sends back to the router edge the new value of the optimal path:

$$x, v \vdash \widehat{G}^{t,\kappa}_s(x, v) \xrightarrow{(v,\ t\ \kappa',\ \langle u \rangle),\ (x,\ \overline{t\ \kappa' \times \kappa},\ \langle u \rangle)} x, v, u \vdash G^\kappa_s(x, v). \tag{12}$$

Otherwise, if $\kappa \not\models T$, the "infinite" cost 0 is backward propagated:

$$x, v \vdash G^\kappa_s(x, v) \xrightarrow{\{(x,\ ev\ T\ t,\ \langle\rangle)\}} x, v \vdash \underline{G}^{t,\kappa}_s(x, v)$$

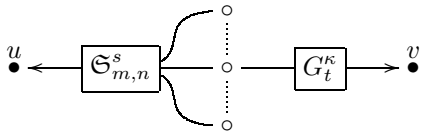
$$x, v \vdash \underline{G}^{t,\kappa}_s(x, v) \xrightarrow{\{(x,\ \overline{t\ 0},\ \langle\rangle)\}} x, v \vdash G^\kappa_s(x, v).$$

Finally, when a link enters the target vertex, then it asks for the p-vertex of the node edge and back-forwards it:

$$x, v \vdash G^\kappa_s(x, v) \xrightarrow{\{(x,\ ev\ T\ s,\ \langle\rangle),\ (v,\ \overline{eval},\ \langle y \rangle)\}} x, v \vdash {G'}^\kappa_s(x, v, y)$$

$$x, v \vdash {G'}^\kappa_s(x, v, y) \xrightarrow{\{(x,\ s\ \kappa,\ \langle y \rangle)\}} x, v \vdash G^\kappa_s(x, v). \tag{13}$$

Given a graph $\Gamma \vdash G$, we say that vertices u and v of G are *link-adjacent* if the graph below is a subgraph of $\Gamma \vdash G$.



A *link path in* G is a sequence of link-adjacent vertices; we say that (free) vertices of a link path are *link-connected*. The cost of a link path is the sum of the costs associated to each link edge appearing in the path. We can now state an important result on selecting the minimal cost path between two link-connected vertices.

Theorem 2. *Let $\Gamma \vdash G$ be a graph and $u, v \in \Gamma$ and v be the vertex where the edge node is connected. If*

$$\Gamma \vdash G \xrightarrow{\Lambda \cup \{(u,\ t\,\kappa,\ \langle u \rangle)\}} \Gamma' \vdash G' \tag{14}$$

then the following properties hold:

1. *if transition (14) can be derived then u and v are link-connected by a path of cost κ;*
2. *if*

is a link-path between u and v in G, then there is a transition like (14) such that $\kappa \leq \Sigma_{i=1}^{h} \kappa_i$.

Theorem 2 means that the path search triggered by remote actions detects a link-path if it exists in the graph (first part of the theorem), moreover the search always selects the minimal cost path connecting two link-connected vertices (second part of the theorem).

Finally, we must consider the productions for disconnecting links. When link edges receive the logout signal from their router edge, they simply disappears. We model this by transforming the cost of the link in an infinite cost:

$$x, v \vdash G^{\kappa}_{s}(x, v) \xrightarrow{\{(x,\ \kappa\ det\ s,\ \langle\rangle)\}} x, v \vdash G^{0}_{s}(x, v) \tag{15}$$

$$x, v \vdash G^{\kappa}_{s}(x, v) \xrightarrow{\{(v,\ s\ det\ \kappa,\ \langle\rangle),\ (x,\ \overline{det},\ \langle\rangle)\}} x, v \vdash G^{0}_{s}(x, v).$$

If the link is not the link selected by the logout signal, the link edge remains connected:

$$x, v \vdash G^{\kappa}_{s}(x, v) \xrightarrow{\{(x,\ t\ det\ \kappa',\ \langle\rangle)\}} x, v \vdash G^{\kappa}_{s}(x, v).$$

8 Path Reservation

This section aims at modifying the productions presented so far in order to permit path reservation and "routing" along reserved link edges. We show how path reservation is essentially obtained by enriching the behaviour of node and link edges with new productions and with slight variations of productions for KAOS actions introduced in Section 7.1.

Let us again consider production:

$$u, p \vdash \varepsilon'_{s}(u, p) \xrightarrow{\{(p,\ s\,\kappa,\ \langle y \rangle),\ (u,\ \overline{run},\ \langle y \rangle)\}} u, p, y \vdash nil. \tag{16}$$

In order to reserve paths, we change the behavior of edge ε'_s. Indeed, vertex y should be considered as the "next-hop" vertex instead of being the final vertex. Therefore, we replace production (16) with

$$u, p \vdash \varepsilon'_s(u,p) \xrightarrow{\{(p,\ s\,\kappa,\ \langle y\rangle)\}} u, p, y \vdash \varepsilon''_s(u,y)$$

Edge ε''_s communicates its destination and waits the vertex where to jump to:

$$u, y \vdash \varepsilon''_s(u,y) \xrightarrow{\{(y,\ \overline{dest\ s},\ \langle\rangle)\}} u, y \vdash \widehat{\varepsilon}_s(u,y)$$

$$u, y \vdash \widehat{\varepsilon}_s(u,y) \xrightarrow{\{(y,\ jump,\ \langle z\rangle)\}} u, y, z \vdash \varepsilon''_s(u,z)$$

until a *stop* signal is received. In this case, ε'' triggers the roaming process sending the *run* message:

$$u, y \vdash \varepsilon''_s(u,y) \xrightarrow{\{(y,\ stop,\ \langle p\rangle),\ (u,\ \overline{run},\ \langle p\rangle)\}} u, y, p \vdash nil.$$

As will be clear later, the last link of the route will synchronize with the above production and stop the migration.

It is also necessary to communicate to the link edges whether they are reserved or not. Therefore, the production of F edges must be changed as

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash F^{s\,\varepsilon\,t}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \boldsymbol{u}, \diamond) \xrightarrow{\Lambda} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \mid \widehat{\Delta}^{h}_{n}(\boldsymbol{x})$$

where $\Lambda = (x_1,\ t\,\kappa_1,\ \langle y_1\rangle), ..., (x_n,\ t\,\kappa_n,\ \langle y_n\rangle), (\boldsymbol{u},\ \overline{t\,\kappa_h},\ \langle y_h\rangle)$ and $\kappa_h = \kappa_1 + ... + \kappa_n$. Edge $\widehat{\Delta}^h_n$ informs link edges whether they are reserved or not:

$$\boldsymbol{x} \vdash \widehat{\Delta}^{h}_{n}(\boldsymbol{x}) \xrightarrow{\Lambda} \boldsymbol{x} \vdash nil$$

where $\Lambda = \{(x_i,\ \overline{nores},\ \langle\rangle) : i = 1, ..., n\ \wedge\ i \neq h\}$. This production makes $\widehat{\Delta}^h_n$ to communicate to the h-th link that it is reserved and to the remaining edges that they have not been selected. Of course links must interact with Δ edges in order to accomplish the previous productions. In particular, productions (12) and (13) must be respectively changed with

$$x, v \vdash \widehat{G}^{t,\kappa}_{s}(x,v) \xrightarrow{\{(v,\ t\,\kappa',\ \langle y\rangle), (x,\ \overline{t\,\kappa' \times \kappa},\ \langle x\rangle)\}} x, v, y \vdash Pr^{\kappa}_{s}(x,v,y).$$

and

$$x, v \vdash {G'}^{\kappa}_{s}(x,v,y) \xrightarrow{\{(x,\ \overline{s\,\kappa},\ \langle x\rangle)\}} x, v, y \vdash Pr^{\kappa}_{s}(x,v,y).$$

The difference lies on the fact that, once the link has backward propagated the cost, it moves to a state Pr^{κ}_{s} where either the *nores* signal is waited or a migrating packet arrives. Edge $Pr^{\kappa}_{s}(x,v,y)$ has an incoming tentacle from x, an outgoing tentacle to v and one to y (where y represents the next-hop vertex).

If a signal *nores* is received, then Pr^{κ}_{s} becomes the link to v as stated in the following production:

$$x, v, y \vdash Pr^{\kappa}_{s}(x,v,y) \xrightarrow{\{(x,\ nores,\ \langle\rangle)\}} x, v, y \vdash G^{\kappa}_{s}(x,v).$$

Otherwise, a packet will be attached to s and Pr_s^κ will take care of its destination. If the destination is s, the packet will terminate its travel:

$$x, v, y \vdash Pr_s^\kappa(x, v, y) \xrightarrow{\{(x,\ dest\, s,\ \langle\rangle)\}} x, v, y \vdash \widehat{Pr}_s^\kappa(x, v, y).$$

$$x, v, y \vdash \widehat{Pr}_s^\kappa(x, v, y) \xrightarrow{\{(x,\ \overline{stop},\ \langle y\rangle)\}} x, v, y \vdash G_s^\kappa(x, v).$$

Once Pr_s^κ receives a signal from an edge ε_s'' that wants to reach s, it replies with a *stop* message where the last hop vertex is communicated. The intention is that y is the p-vertex of the node edge of s.

A *jump* signal is emitted,to let the packet reach vertex t different from s:

$$x, v, y \vdash Pr_s^\kappa(x, v, y) \xrightarrow{\{(x,\ dest\, t,\ \langle\rangle)\}} x, v, y \vdash {Pr'}_s^\kappa(x, v, y)$$

$$x, v, y \vdash {Pr'}_s^\kappa(x, v, y) \xrightarrow{\{(x,\ \overline{jump},\ \langle y\rangle)\}} x, v, y \vdash G_s^\kappa(x, v).$$

The productions presented in this section and Theorem 2 in the previous section ensure that whenever a remote operation is performed the graphical calculus always selects the optimal path with respect to the QoS attributes specified by the KAOS networking constructs. This result depends on the outcome of a distributed constraint satisfaction problem, the *rule matching problem* [25]. For the result to hold, QoS attributes must form an ordered c-semiring [3], whose additive and multiplicative operations allow us to compare and compose QoS parameters.

9 Messaging & Graphs

This section shows how the graphical calculus is applied to the messaging application of Section 4. We aim at illustrating how a minimal path between two nodes can be reserved and traversed by a remotely evaluated process. We also describe how the Floyd-Warshall algorithm [12] computes the same paths when applied to the graph.

We consider the configuration reached by net (2) (see page 15) after the notification message has reached t and has been acquired by R. More precisely, we discuss how filter process F is remotely executed at s. In particular, we focus on determining the minimal path from t to s and how F traverses it in the net topology determined in Section 4.

Figure 7 reports the graphical representation of the described configuration; it does not faithfully represent the KAOS net in terms of graphs because we avoid depicting vertex $\diamond$ and all tentacles connecting node edges with it. Moreover, vertices where process edges are attached are graphically represented as $\star$. Both these choices have respectively been adopted because

- no synchronization takes place on vertex $\diamond$ during the routing phase of F, and

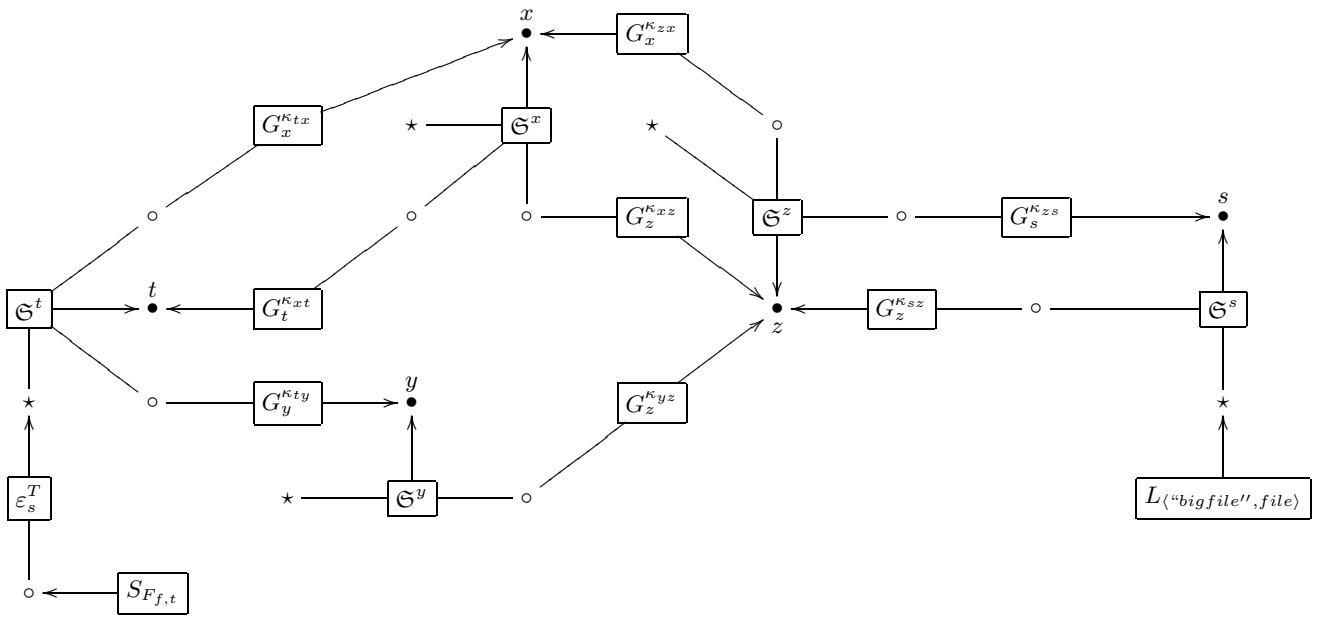


Fig. 7. Graph for messaging net

– Figure 7 becomes more readable.

The graph of Figure 7 is the counterpart of Figure 2 in terms of graphs. Node edges are connected to their outgoing link edges on vertices ∘, while edges for incoming links are attached on vertices •. Initially, edge ε_s^T "wraps" the edge corresponding to the filter process $S_{F_{f,t}}$ and is connected to the ⋆ vertex corresponding to node edge $\mathfrak{S}^t$. This amounts to say that $\varepsilon(F_{f,t})@s$ is allocated at t. Similarly, edge $L_{\langle \text{“}bigfile'', file\rangle}$ is allocated at s.

Instead of detailing the semantical framework for deducing graph transitions, we describe edge behaviour and synchronization in terms of graphical figures (similar to Figure 7), where tentacles are annotated by synchronizing actions. Since tentacles are connected to the vertices where synchronizations take place, we avoid writing those information in the labels.

Figure 8 summarizes the productions used for searching a (minimal) path connecting s and t and represents the graph with the annotated edges. Notice how each node edge $\mathfrak{S}^u$ enriches *ev* labels with name u. For instance, edge $\mathfrak{S}^x$ "receives" the signal *t ev Ts* along tentacle to x and forwards the signal *tx ev Ts* to its links. We remark that node edge $\mathfrak{S}^z$ can non-deterministically synchronize with two different *ev* actions, i.e. the one triggered by the link from x or from y nodes. However, the result does not depend on the chosen synchronization.
Differently from the other link and node edges, $G_s^{\kappa_{zs}}$ and $\mathfrak{S}^s$ synchronize through the *eval* signal because $G_s^{\kappa_{zs}}$ is a link to the target vertex s. This synchronization allows $G_s^{\kappa_{zs}}$ to determine the ⋆ vertex of $\mathfrak{S}^s$ as will be explained in the following.

According to the edge replacement mechanisms, the graph in Figure 8 rewrites as shown in Figure 9 (where also the productions for the next graph transition are listed). Figure 9 depicts the graph after the synchronizations enabled in the graph in Figure 8 have taken place. We let $\kappa = \kappa_{zs} \times \kappa_{xz} \times \kappa_{tx}$ and $\kappa' = \kappa_{zs} \times \kappa_{yz} \times \kappa_{ty}$. Labels appearing on tentacles do not mention the "next-hop" vertex as (formally) required by the corresponding productions because it is graphically represented by the ∘ vertex where link edges are attached at. As stated above, $G_z^{\kappa_{zs}}$ has interacted with $\mathfrak{S}^s$ and has acquired the vertex where (the graph corresponding to) filter F must be connected and executed. This is represented by the dotted tentacle in Figure 9.
In this phase, costs are backwardly propagated by link and node edges. Notice that $F^{t\,\varepsilon\,s}$ edge connected to t forwads the (minimal) cost on its ⋆ edge, whereas the other $F^{t\,\varepsilon\,s}$ edges send the costs on their • vertices. This is due to the fact that the second and third productions reported in Figure 8 distinguish whether the *ev* signal has been received on the ⋆ or the • vertex and consequently determine the "result" vertex r.

The graph resulting from synchronizing productions in Figure 9 is reported in Figure 10. As before, dotted tentacles represents next-hop "address" of the reserved links. Indeed, edge ε' moves on the "tail" vertex of the link connecting t to x. On this vertex, it will synchronize with $Pr_x^{\kappa_{tx}}$ through the *jump* productions and first reach the (reserved) link from x to z, then the link from z to s and, finally, it will receive the signal *stop* together with the p-vertex of $\mathfrak{S}^t$.

$$u, p \vdash \varepsilon_s^T(u,p) \xrightarrow{\{(r,\ \overline{ev\ T\ s},\ \langle\rangle)\}} u, p \vdash \varepsilon'_s(u,p)$$

$$\boldsymbol{x}, p, s, \diamond \vdash \mathfrak{S}_n^s(\boldsymbol{x}, p, s, \diamond) \xrightarrow{\{(p,\ ev\ T\ t,\ \langle\rangle),\ (\boldsymbol{x},\ \overline{s\ ev\ T\ t},\ \langle\rangle)\}} \boldsymbol{x}, p, s, \diamond \vdash F_{n,1}^{s\,\varepsilon\,t}(\boldsymbol{x}, p, s, p, \diamond)$$

$$\boldsymbol{x}, p, s, \diamond \vdash \mathfrak{S}_n^s(\boldsymbol{x}, p, s, \diamond) \xrightarrow{\{(s,\ \sigma\ ev\ T\ t,\ \langle\rangle),\ (\boldsymbol{x},\ \overline{\sigma s\ ev\ T\ t},\ \langle\rangle)\}} \boldsymbol{x}, p, s, \diamond \vdash F_{n,1}^{s\,\varepsilon\,t}(\boldsymbol{x}, p, s, s, \diamond)$$

$$x, s \vdash G_s^\kappa(x,s) \xrightarrow{\{(x,\ \sigma\ ev\ T\ t,\ \langle\rangle),\ (s,\ \overline{\sigma\ ev\ T\ t},\ \langle\rangle)\}} x, s \vdash \widehat{G}_s^{t,\kappa}(x,s)$$

$$x, s \vdash G_s^\kappa(x,s) \xrightarrow{(x,\ \sigma\ ev\ T\ t,\ \langle\rangle)} x, s \vdash \underline{G}_s^{t,\kappa}(x,s), \text{ if } s \text{ appears in } \sigma$$

$$x, s \vdash G_s^\kappa(x,s) \xrightarrow{\{(x,\ \sigma\ ev\ T\ s,\ \langle\rangle),\ (s,\ \overline{eval},\ \langle y\rangle)\}} x, s \vdash G'^\kappa{}_s(x,s,y)$$

$$\boldsymbol{x}, p, s, \diamond \vdash \mathfrak{S}_n^s(\boldsymbol{x}, p, s, \diamond) \xrightarrow{\{(s,\ eval,\ \langle p\rangle)\}} \boldsymbol{x}, p, s, \diamond \vdash \mathfrak{S}_n^s(\boldsymbol{x}, p, s, \diamond)$$

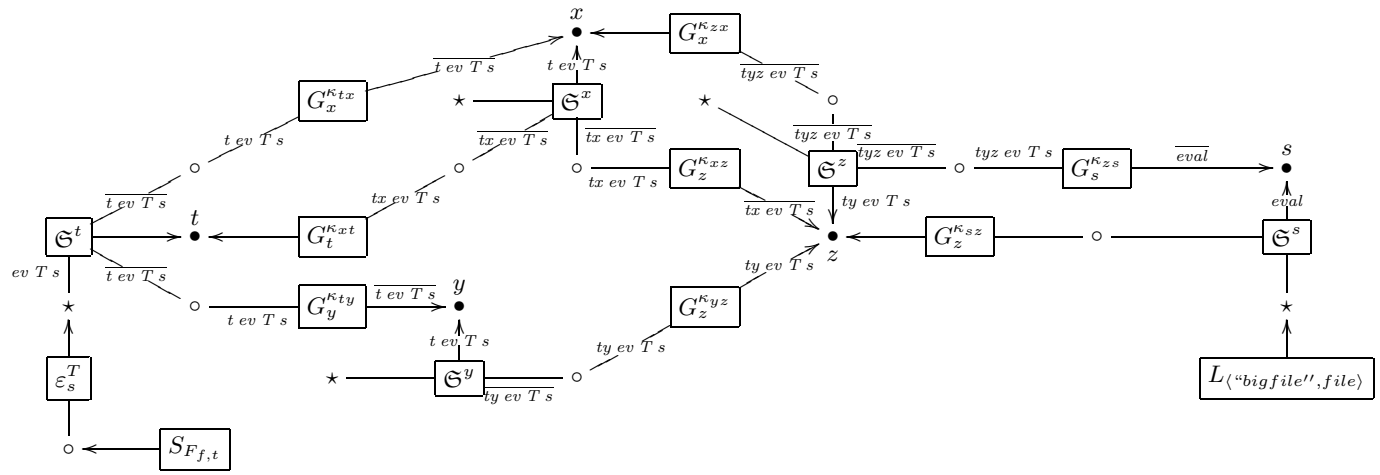


Fig. 8. Asking for paths

$$\boldsymbol{x}, p, s, r, \diamond \vdash F^{s\ \varepsilon\ t}_{n,i}(\boldsymbol{x}, p, s, r, \diamond) \xrightarrow{\substack{(x_1,\ t\,\kappa_1,\ \langle y_1\rangle), ..., \\ (x_n,\ t\,\kappa_n,\ \langle y_n\rangle), \\ (r,\ \overline{t\,\kappa_h},\ \langle y_h\rangle)}} \boldsymbol{x}, p, s, r, \diamond \vdash \mathfrak{S}^s_n(\boldsymbol{x}, p, s, \diamond) \mid \widehat{F}^{t\,\kappa}_{i-1}(r) \mid \widehat{\Delta}^h_n(\boldsymbol{x}),$$
where $i \geq 1$ and $\kappa_h = \kappa_1 + ... + \kappa_n$

$$\boldsymbol{x} \vdash \widehat{\Delta}^h_n(\boldsymbol{x}) \xrightarrow{\Lambda} \boldsymbol{x} \vdash nil, \text{ where } \Lambda = \{(x_i,\ \overline{nores},\ \langle\rangle) : i = 1, ..., n\ \wedge\ i \neq h\} \cup (x_h,\ \overline{res},\ \langle\rangle)$$

$$x, s \vdash \widehat{G}^{t,\kappa}_s(x, s) \xrightarrow{\{(s,\ t\,\kappa',\ \langle y\rangle), (x,\ \overline{t\,\kappa' \times \kappa},\ \langle x\rangle)\}} x, s, y \vdash Pr^{\kappa}_s(x, s, y)$$

$$x, s \vdash {G'}^{\kappa}_s(x, s, y) \xrightarrow{\{(x,\ \overline{s\,\kappa},\ \langle y\rangle)\}} x, s, y \vdash Pr^{\kappa}_s(x, s, y)$$

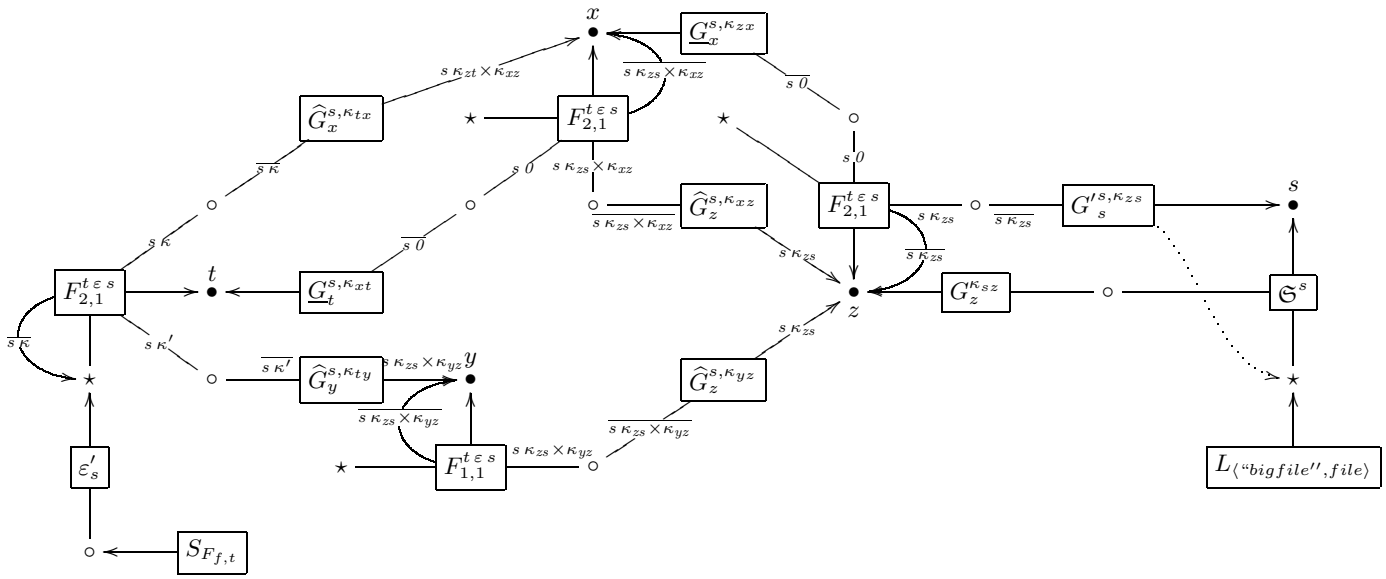


Fig. 9. Computing paths

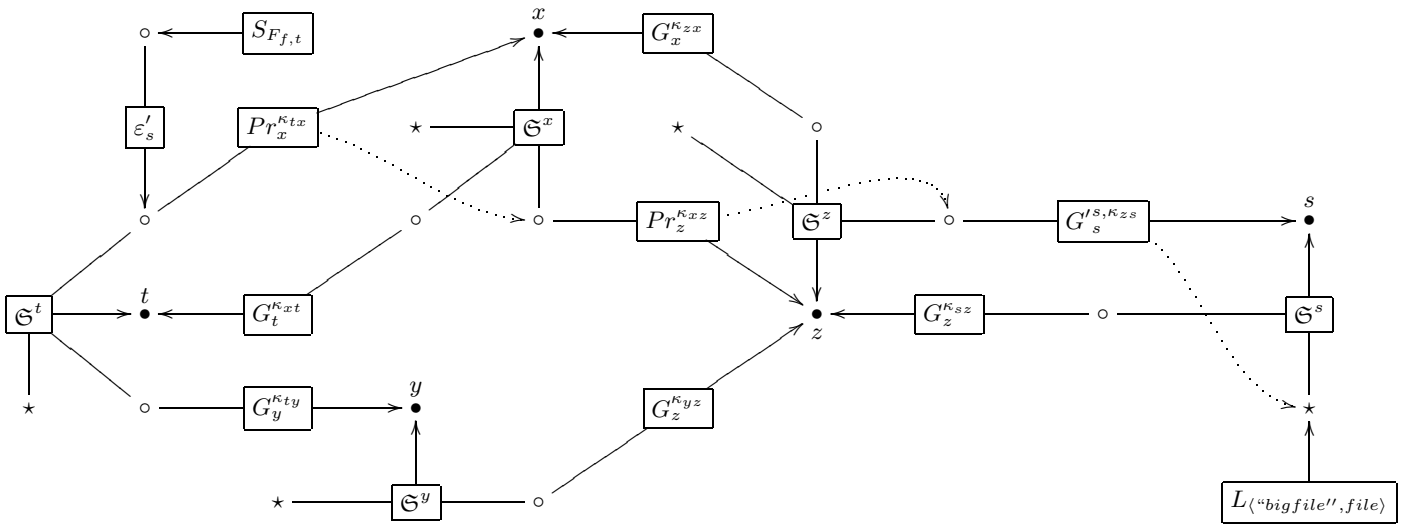


Fig. 10. Minimal path

In this example, in order to give a smooth presentation, we do not consider the productions for path reservation that are determined by synchronizing the first and the second productions in Figure 9. However, such synchronizations are straightforward and simply make link edges become Pr edges or make them return in their initial "state", depending whether they are reserved or not.

Given a graph representing a KAOS net and a process $\varepsilon(Q)@t$ allocated on some node s (as net in Figure 7), we can built a matrix of costs such that the Floyd-Warshall algorithm [12] can be used to compute (one of) the minimal path(s) connecting s and t that can be traversed by Q.

Intuitively, if $\kappa_{uv} \models T(Q)$ then position (u, v) of the matrix contains κ_{uv}, the cost of the the link edge from u to v;if $\kappa_{uv} \not\models T(Q)$ or no link edges connects u and v, then position (u, v) contains 0. Table 8 reports the matrix corresponding to the net in Figure 7.

	t	x	y	z	s
t	0	κ_{tx}	0	0	0
x	κ_{xt}	0	0	κ_{xz}	0
y	0	0	0	κ_{yz}	0
z	0	κ_{zx}	0	0	κ_{zs}
s	0	0	0	κ_{sz}	0

Table 8. The initial matrix

Notice that position (t, y) contains 0 because $\kappa_{ty} \not\models T(F_{f,t})$.

Given a vertex $z \neq t$, let $z{-}\mathbf{1}$ represent the vertex that precedes z in the list $[t, x, y, z, s]$. The Floyd-Warshall algorithm is an iterative algorithm that transforms the matrix of costs according the following relation:

$$\kappa_{uv}^{z} = \kappa_{uv}^{z-\mathbf{1}} + (\kappa_{uz}^{z-\mathbf{1}} \times \kappa_{zv}^{z-\mathbf{1}})$$

where $+$ and $\times$ are the c-semiring operations.

Table 9 reports the matrices computed by the iterations of the Floyd-Warshall algorithm starting from the cost matrix of Table 8. Position (t, s) of the last matrix in Table 9 contains the cost of the minimal path from t to s.

We remark that the Floyd-Warshall algorithm can be applied when costs of edges are totally ordered. In our example this was the case, but in general it is not. For instance, consider two vertices connected by two link edges having costs $\langle 1, T, 2\rangle$ and $\langle 2, T, 1\rangle$, respectively. According to the definition of $+$, we have that $\langle 1, T, 2\rangle + \langle 2, T, 1\rangle = \langle 1, T, 1\rangle$, which does not correspond to any path between the vertices. In order to overcome this problem, we can use the *Hoare powerdomain* of the previous c-semiring of costs. In [4] it has been noticed that if $(A, +, \times, 0, 1)$ is a c-semiring then the Hoare powerdomain $(\wp^H(A), \cup, \times^*, \emptyset, A)$ is also a c-semiring; here

	$\boxed{t}$	x	y	z	s
t	0	κ_{tx}	0	0	0
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	0
y	0	0	0	κ_{yz}	0
z	0	κ_{zx}	0	0	κ_{zs}
s	0	0	0	κ_{sz}	0

	t	$\boxed{x}$	y	z	s
t	$\kappa_{tx} \times \kappa_{xt}$	κ_{tx}	0	$\kappa_{tx} \times \kappa_{xz}$	0
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	0
y	0	0	0	κ_{yz}	0
z	$\kappa_{zx} \times \kappa_{xt}$	κ_{zx}	0	$\kappa_{zx} \times \kappa_{xz}$	κ_{zs}
s	0	0	0	κ_{sz}	0

	t	x	$\boxed{y}$	z	s
t	$\kappa_{tx} \times \kappa_{xt}$	κ_{tx}	0	$\kappa_{tx} \times \kappa_{xz}$	0
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	0
y	0	0	0	κ_{yz}	0
z	$\kappa_{zx} \times \kappa_{xt}$	κ_{zx}	0	$\kappa_{zx} \times \kappa_{xz}$	κ_{zs}
s	0	0	0	κ_{sz}	0

	t	x	y	$\boxed{z}$	s
t	$\kappa_{tx} \times \kappa_{xt}$	κ_{tx}	0	$\kappa_{tx} \times \kappa_{xz}$	$\kappa_{tx} \times \kappa_{xz} \times \kappa_{zs}$
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	$\kappa_{xz} \times \kappa_{zs}$
y	$\kappa_{yz} \times \kappa_{zx} \times \kappa_{xt}$	$\kappa_{yz} \times \kappa_{zx}$	0	κ_{yz}	$\kappa_{yz} \times \kappa_{zs}$
z	$\kappa_{zx} \times \kappa_{xt}$	κ_{zx}	0	$\kappa_{zx} \times \kappa_{xz}$	κ_{zs}
s	$\kappa_{sz} \times \kappa_{zx} \times \kappa_{xt}$	$\kappa_{sz} \times \kappa_{zx}$	0	κ_{sz}	$\kappa_{sz} \times \kappa_{zs}$

	t	x	y	z	$\boxed{s}$
t	$\kappa_{tx} \times \kappa_{xt}$	κ_{tx}	0	$\kappa_{tx} \times \kappa_{xz}$	$\kappa_{tx} \times \kappa_{xz} \times \kappa_{zs}$
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	$\kappa_{xz} \times \kappa_{zs}$
y	$\kappa_{yz} \times \kappa_{zx} \times \kappa_{xt}$	$\kappa_{yz} \times \kappa_{zx}$	0	κ_{yz}	$\kappa_{yz} \times \kappa_{zs}$
z	$\kappa_{zx} \times \kappa_{xt}$	κ_{zx}	0	$\kappa_{zx} \times \kappa_{xz}$	κ_{zs}
s	$\kappa_{sz} \times \kappa_{zx} \times \kappa_{xt}$	$\kappa_{sz} \times \kappa_{zx}$	0	κ_{sz}	$\kappa_{sz} \times \kappa_{zs}$

Table 9. Iterations of the Floyd-Warshall Algorithm

- $\wp^H(A)$ is the set of all the subsets X of A which are downward closed under the ordering $\leq$ induced on A by the $+$ operation[4], i.e. $\wp^H(A) = \{X \subseteq A : \forall x \in X. \forall y \in A. y \leq x \Rightarrow y \in X\}$;
- $\cup$ is set union and multiplication $\times^*$ is just $\times$ extended to sets, namely $X \times^* Y = \{x \times y : x \in X \wedge y \in Y\}$.

Moreover, if $+$ induces a total order on A then $\wp^H(A)$ is isomorphic to A with an additional bottom element $\emptyset$, hence it does not give any additional information, whereas, if A is not totally ordered by $+$, then application of the Floyd-Warshall algorithm to the Hoare powerdomain computes the costs of all paths. To reduce

[4] Remember that $x \leq y \Leftrightarrow \exists z. x + z = y$.

the computational cost of the algorithm, it is possible to represent each $X \in \wp^H(A)$ with the set of its local maxima which may be small with respect to X. Then the Floyd-Warshall algorithm computes the costs of all the *non-dominated* paths between two nodes, namely all the paths which are maximal and are not comparable according to the order induced by $+$. Once the costs of all the non-dominated paths out of a node s have been computed, to trace an actual path of cost κ to node t it is sufficient to find an edge out of s with cost κ_1 connected to a node r having a path of cost κ_2 to t such that $\kappa = \kappa_1 \times \kappa_2$, and then to proceed similarly from r.

10 Concluding Remarks

We have introduced a formal model that provides mechanisms to specify and reason about application-oriented QoS. We demonstrate the applicability of the approach by providing the formal modeling of KAOS QoS mechanisms.

The novelty of our proposal is given by the combination of the following ingredients:

- adopt a declarative approach to the specification of QoS attributes;
- adopt a graphical calculus to describe system evolution;
- reduce declarative QoS specification to semantic constraints of the graphical calculus.

One may wonder if this approach is too abstract and general and it does not capture the intrinsic limitations of inter-networking computations. We feel that on the one side the generality of the approach can be tamed and adapted to the needs of the various layers of applications, more powerful primitives being made available to upper layers, like *business to business* (B2B) or *computer supported collaborative work* (CSCW). On the other side, some important network technologies actually require the solution of global constraints, like modifying local router tables according to the routing update information sent by the adjacent routers.

As a future work, we plan to investigate the expressive power of the graphical model and to develop proof techniques to analyze QoS properties.

References

1. O. Angin, A. Campbell, M. Kounavis, and R. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Personal Communications Magazine*, August 1998.
2. L. Bettini, M. Loreti, and R. Pugliese. An infrastructure language for open nets. In Proc. of the 2002 ACM Symposium on Applied Computing (SAC'02), Special Track on Coordination Models, Languages and Applications. ACM Press, 2002.
3. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
4. S. Bistarelli, U. Montanari, and F. Rossi. Soft constraint logic programming and generalized shortest path problems. *Journal of Heuristics*, 8:25–41, 2002.

5. S. Blake, D. Black, M. Carlson, E. Davies, Z. Wand, and W. Weiss. An architecture for differentiated services. Technical Report RFC 2475, The Internet Engineering Task Force (IETF), 1998.
6. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp) - version 1 functional specification.
7. L. Cardelli and R. Davies. Service combinators for web computing. *Software Engineering*, 25(3):309–316, 1999.
8. I. Castellani and U. Montanari. Graph Grammars for Distributed Systems. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Proc. 2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 20–38. Springer-Verlag, 1983.
9. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
10. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, June 2000.
11. P. Degano and U. Montanari. A model of distributed systems based of graph rewriting. *Journal of the ACM*, 34:411–449, 1987.
12. R. Floyd. Algorithm97 (shortestpath). *Communication of the ACM*, 5(6):345, 1962.
13. I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, 1999.
14. D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of software architecture styles with name mobility. In A. Porto and G.-C. Roman, editors, *Coordination 2000*, volume 1906 of *LNCS*, pages 148–163. Springer Verlag, 2000.
15. D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility: A graphical calculus for name mobility. In *12th International Conference in Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 121–136, Aalborg, Denmark, 2001. Springer Verlag.
16. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985. & 0-13-153289-8.
17. IBM Software Group. Web services conceptual architecture. In *IBM White Papers*, 2000.
18. M. Koch, L. Mancini, and F. Parisi-Presicce. A formal model for role-based access control using graph transformation. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, editors, *ESORICS*, volume 1895 of *LNCS*, pages 122–139, 6th European Symposium on Research in Computer Security, 2000. Springer Verlag.
19. M. Koch, L. Mancini, and F. Parisi-Presicce. Foundations for a graph-based approach to the specification of access control policies. In F. Honsell and M. Lenisa, editors, *FoSSaCS*, LNCS, Foundations of Software Science and Computation Structures, 2001. Springer Verlag.
20. M. Koch and F. Parise-Presicce. Describing policies with graph constraints and rules. In A. Corradini, H. Ehrig, H. Kreowski, and G. Rozenberg, editors, *Graph Transformation*, volume 2505 of *LNCS*, pages 223–238, First International Conference on Graph Transformation, Barcelona, Spain, October 2002. Springer Verlag.
21. B. Koenig and U. Montanari. Observational equivalence for synchronized graph rewriting. In *Proc. TACS'01*, LNCS. Springer Verlag, 2001. To appear.
22. B. Li. *Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations*. PhD thesis, University of Illinois, 2000.

23. R. Milner. *Communication and Concurrency*. Printice Hall, 1989.
24. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
25. U. Montanari and F. Rossi. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference COORDINATION '96, Cesena, Italy*, volume 1061 of *LNCS*. Springer Verlag, April 1996.
26. J. Sobrinho. Algebra and algorithms for qos path computation and hop-by-hop routing in the internet. *IEEE Transactions on Networking*, 10(4):541–550, August 2002.
27. G. Winskel. Synchronization trees. *Theoretical Computer Science*, May 1985.
28. X. Xiao and L. M. Ni. Internet qos: A big picture. *IEEE Network*, 13(2):8–18, Mar 1999.
29. M. Yokoo and K. Hirayama. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.