Spatial-Behavioral Types, Distributed Services, and Resources

Luís Caires

CITI / Departamento de Informática, Universidade Nova de Lisboa, Portugal

Abstract. We develop a notion of spatial-behavioral typing suitable to discipline interactions in service-based systems modeled in a distributed object calculus. Our type structure reflects a resource aware model of behavior, where a parallel composition type operator expresses resource independence, a sequential composition type operator expresses implicit synchronization, and a modal operator expresses resource ownership. Soundness of our type system is established using a logical relations technique, building on a interpretation of types as properties expressible in a spatial logic.

1 Introduction

The aim of this work is to study typing disciplines for service-based systems, with a particular concern with the key aspects of safety, resource control, and compositionality. For our current purposes, we consider service-based systems to be certain kinds of distributed object systems, but where binding between parties is dynamic rather than static, system assembly is performed on-the-fly depending on discoverable resources, interactions between parties may involve long duration protocols, and the fundamental abstraction mechanism is task composition, rather than just remote method invocation. In this paper, we approach the issue of compositional analysis of distributed services and resources using a new notion of typing inspired by spatial logics. Technically, we proceed by introducing a core calculus for distributed services, where clients and servers are represented by concurrent "objects" (aggregates of operations and state). Services are called by reference, and references (names) to services may be passed around, as in π-calculi. New services may also be dynamically instantiated. We then develop and study a fairly expressive type system aimed at disciplining, in a compositional way, interactions and resource usage in such systems.

Our type structure is motivated by fundamental features and properties of our intended model. We conceive a service-based system as a layered distributed system, where service provider objects execute tasks in behalf of client objects, in a coordinated way. Even if the same object may act as client and server, we do not expect intrinsic cyclic dependencies to occur in such a system. The main coordination abstractions for assembling tasks into services are probably parallel (independent) and sequential composition. Tasks are independent when they never get to compete for resources; independent tasks appear to run simultaneously, this is the default behavior of the "global computer". On the other hand, causality, data flow, and resource competition introduce constraints in the control flow of computations. We will thus consider tasks and resources as the basic building blocks of service based systems.

Models of concurrent programming usually introduce two kinds of entities in their universe of concepts: processes (active) and resources (passive). While processes are

the main subject of analysis, resources are considered atomic, further unspecified entities besides being unshareable by definition (with objects such as as files, channels, etc, memory cells, given as classical examples). We adopt a view where resources and objects are not modeled a priori by different sorts of entities: but where *everything is an object*. Our distinction criteria is observational, and not strict: what distinguishes a resource among other objects is that resources must be used with care so to avoid meaningless or disrupted computations. For example, a massively replicated service such as Google behaves pretty much as if every client owned its own private copy of it. On the other hand, an object handling an e-commerce session with a user, is certainly not supposed to be shared: if other user gets in the middle and interferes with the session things may go wrong! We then consider the latter more "resource-like" than the former. Thus, instead of thinking of resources as external entities, for which usage policies are postulated, we consider a resource to be any object expressible in our model that *must* be used according to a strict discipline to avoid getting into illegal states. Our semantics realizes such illegal states concretely, as standard "message not understood" errors, rather than as violations of extraneously imposed policies, as *e.g.*, in [14, 17, 2].

Adopting a deep model of resources as fragile objects brings generality to our approach. Just as sequentiality in a workflow results from resource competition, resource competition is problematic in the sense that if a system does not respect precise sharing constraints on objects, illegal computation states may arise. This view allows us to conceive more general sharing constraints than the special cases usually considered: *e.g.*, at certain stage of a protocol a resource might be shareable, while at other stage it may be not. Such an uniform approach also naturally supports a computational model where resources may be passed around in transactions, buffered in pools, while ensuring that their capabilities are used consistently, by means of typing. Our type system, we believe, captures the fundamental constraints on resource access present in general concurrent systems. It is based on the following constructors

$$U,V ::= \mathtt{stop} \mid U \mid V \mid U \wedge V \mid U;V \mid U^\circ \mid U \rhd V \mid \mathtt{l}(U)V$$

to which we add a recursion operator (and type variables). The spatial composition type $U \mid V$ states that a service may be used accordingly to U and V by independent clients, one using it as specified by U, the other as specified by V. This implies, in particular, that the tasks U and V may be activated concurrently. For example, an object typed with

$$Travel \triangleq (\mathtt{flight} \mid \mathtt{hotel}); \mathtt{order}$$

will be able to service the `flight` (we abbreviate $\mathtt{l}(\mathtt{stop})\mathtt{stop}$ by $\mathtt{l}$, and so on) and `hotel` tasks simultaneously and after that (and only after that), the `order` task. The spatial reading of $U \mid V$ implies further consequences, namely that the (distributed) resources used by U and V do not interfere; this property is important to ensure closure under composition of certain safety properties of typed systems. Owned types, of the form U°, state not only that the service is usable as specified by U, but also that such usage is completely owned (so that a object possessing a reference of owned type may, for example, store it for later use). Owned types allow one to distinguish between services that *must* be used according to U, and services that *may* be used according to U;

this distinction is crucial to control delegation of resources or services between partners. More familiar behavioral types may also be easily expressed. For example, using sequential composition and conjunction, the usage protocol of a file might be specified

$$File(V) \triangleq (\texttt{open}; (\texttt{read}()V \wedge \texttt{write}(V))^{\star}; \texttt{close})^{\star}$$

where $U^{\star} \triangleq \textbf{rec } \alpha.(\textbf{stop} \wedge (U; \alpha))$ expresses iteration. By combining recursion with spatial types, we then define shared types. A shared type $U!$ states of an object that it may be used according to an unbounded number of independent sessions, each one conforming to type U. By combining such operators, we may specify fine grained shared access protocols, such as standard "multiple readers/unique writer" access pattern:

$$RW(V) \triangleq ((\texttt{read}()V)!; \texttt{write}(V^{\circ}))^{\star}$$

Finally, and crucially, guarantee types, of the form $U \triangleright V$, allows us to compose subsystems into systems while preserving the properties ensured by their typings.

 The paper is structured as follows. In Section 2, we present our core language and its operational semantics, and some examples. In Section 3 we introduce our basic type system, and prove its soundness. Our proof combines syntactical and semantical reasoning, in the spirit of the logical relations technique, where types are interpreted as properties expressed in a spatial logic. In Section 4, we show how to extend our basic system to cover more general forms of sharing. Finally, Section 5 discusses related work and draws some conclusions.

2   A Distributed Service Calculus

In this section we present the syntax and operational semantics of our distributed service calculus. We assume given an infinite set $\mathcal{N}$ of *names*. Names are used to identify objects (n, m, p), threads (b, c, d) and state elements (a). We also assume given an infinite set $\mathcal{X}$ of *variables* (x, y, z), and an infinite set $\mathcal{L}$ of *method labels* $(\texttt{j}, \texttt{k}, \texttt{l})$. We note $\mathcal{X} = \mathcal{N} \cup \mathcal{V}$ and let η range over $\mathcal{X}$ (variables and names). We start by introducing expressions. In the definition of systems, expressions may syntactically occur either in the body of a method definition, or in a thread.

Definition 2.1 (Values, Expressions, Methods). *The sets $\mathcal{V}$ of values, $\mathcal{E}$ of expressions, and $\mathcal{M}$ of methods are defined by the abstract syntax in Fig. 1 (top).*

 We use the notation $\overline{\varsigma}$ to denote a sequence of syntactical elements of class ς. The value **nil** is an atomic value that stands for the null reference. The *call* expression $n.\texttt{l}(v)$ denotes the invocation of the method $\texttt{l}$ of object n, where the value v is passed as argument. The *wait* expression $n.c()$ denotes waiting for a reply to a previously issued method invocation of the form $n.\texttt{l}(v)$, where c is the identifier of the thread which is serving the request (remotely). The wait construct plays a key technical role in our formulation of the dynamic and static semantics of our language, even if it is not expected to appear in source programs. The *composition* construct **let** $\overline{x = e}$ **in** f denotes the parallel evaluation of the expressions e_i, followed by the evaluation of the body f,

$$e, f, h ::= \in \mathcal{E} \quad \text{(Expressions)} \qquad\qquad v, r ::= \in \mathcal{V} \quad \text{(Values)}$$

$e, f, h ::= \in \mathcal{E}$	(Expressions)		$v, r ::= \in \mathcal{V}$	(Values)
v	(Value)		n	(Name)
$\mid \;\; v.\mathtt{l}(v)$	(Call)		$\mid \;\; x$	(Identifier)
$\mid \;\; n.c()$	(Wait)		$\mid \;\; \mathbf{nil}$	(Termination)
$\mid \;\; a?$	(Read)			
$\mid \;\; a!(v)$	(Write)		$M ::= \in \mathcal{M}$	(Methods)
$\mid \;\; \mathbf{new}\;[M]$	(Object Creation)		$\mathbf{0}$	(Empty)
$\mid \;\; \mathbf{let}\;\; \overline{x = e}\;\; \mathbf{in}\;\; e$	(Composition)		$\mid \;\; \mathtt{l}(x) = e$	(Method)
$\mid \;\; \mathbf{rec}\;\; x.e$	(Recursion)		$\mid \;\; M \mid M$	(Methods)

$s ::= \in \mathcal{S}$ (Stores)	$t ::= \in \mathcal{T}$ (Threads)	$P, Q, R ::= \in \mathcal{P}$	(Network)
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	(Empty)
$\mid \;\; a\langle v \rangle$	$\mid \;\; t \mid t$	$\mid \;\; (\nu n)P$	(Restriction)
$\mid \;\; s \mid s$	$\mid \;\; c\langle e \rangle$	$\mid \;\; P \mid Q$	(Composition)
		$\mid \;\; n[M\,;\,s\,;\,t]$	(Object)

Fig. 1. Values, Expressions, Methods, Stores, Threads, Networks.

$$n.\mathtt{l}(v) \xrightarrow{\overline{n.\mathtt{l}_c(v)}} n.c() \qquad\qquad \mathbf{new}[M] \xrightarrow{n[M]} n$$

$$n.c() \xrightarrow{n.c(v)} v \qquad\qquad \frac{e\{x \leftarrow \mathbf{rec}\;\; x.e\} \xrightarrow{\alpha} e'}{\mathbf{rec}\;\; x.e \xrightarrow{\alpha} e'} \;\text{(Rec)}$$

$$a? \xrightarrow{a?(v)} v \qquad\qquad \frac{e \xrightarrow{\alpha} e'}{\mathbf{let}\;\; \cdots, x = e, \cdots \;\; \mathbf{in}\;\; f \xrightarrow{\alpha} \mathbf{let}\;\; \cdots, x = e', \cdots \;\; \mathbf{in}\;\; f}$$

$$a!(v) \xrightarrow{a!(v)} \mathbf{nil} \qquad\qquad \mathbf{let}\;\; \overline{x = v}\;\; \mathbf{in}\;\; e \xrightarrow{\tau} e\overline{\{x \leftarrow v\}}$$

Fig. 2. Evaluation (Expressions).

$$\frac{e \xrightarrow{\overline{n.\mathtt{l}_c(v)}} e' \quad [c\;fresh]}{n[\mathtt{l}(x) = h\,;\;\,;\,] \mid m[\;;\;\,; b\langle e \rangle] \to (\nu c)(n[\mathtt{l}(x) = h\,;\;\,; c\langle h\{x \leftarrow v\} \rangle] \mid m[\;;\;\,; b\langle e' \rangle])}$$

$$\frac{e \xrightarrow{n.c(r)} e'}{n[\;;\;\,; c\langle r \rangle] \mid m[\;;\;\,; b\langle e \rangle] \to m[\;;\;\,; b\langle e' \rangle]} \qquad \frac{e \xrightarrow{\tau} e'}{n[\;;\;\,; c\langle e \rangle] \to n[\;;\;\,; c\langle e' \rangle]}$$

$$\frac{e \xrightarrow{a?(v)} e'}{n[\;; a\langle v \rangle \,; c\langle e \rangle] \to n[\;;\;\,; c\langle e' \rangle]} \qquad \frac{e \xrightarrow{a!(v)} e'}{n[\;;\;\,; c\langle e \rangle] \to n[\;; a\langle v \rangle \,; c\langle e' \rangle]}$$

$$\frac{e \xrightarrow{m[M]} e' \quad [m\;fresh]}{n[\;;\;\,; c\langle e \rangle] \to (\nu m)(m[M\,;\;\,;\,] \mid n[\;;\;\,; c\langle e' \rangle])} \qquad \frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}$$

$$\frac{P \to Q}{(\nu n)P \to (\nu n)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R}$$

Fig. 3. Reduction (Networks).

where the result of evaluating each e_i is bound to the corresponding x_i. The x_i are distinct bound variables, with scope the body f. The `let` construct allows us to express arbitrary parallel / sequential control flow graphs, in which values may be propagated between parallel and sequential subcomputations. We use the following abbreviations (where x_1 and x_2 do not occur in e_1 and e_2):

$$(e_1 \mid e_2) \triangleq \texttt{let } x_1 = e_1 , x_2 = e_2 \texttt{ in nil} \qquad (e_1;e_2) \triangleq \texttt{let } x_1 = e_1 \texttt{ in } e_2$$

The $a?$ and $a!(v)$ constructs allow objects to manipulate their local store. The *read* expression $a?$ picks and returns a value stored under tag a, while the *write* expression $a!(v)$ stores value v in the store under tag a. The store conforms to a data space model, where reading consumes data, and writing adds new data elements. Evaluation of `new`$[M]$ results in the allocation of a new object, with set of methods M, and whose identity (a fresh name) is returned. In the *method* $\texttt{l}(x) = e$ the parameter x is bound in the scope of the method body e (for the sake of simplicity, we just consider single parameter methods). Finally, the `rec` construct introduces recursion. To keep our language "small", we refrain from introducing other useful ingredients, such as basic data types and related operators, for instance booleans and conditionals. Since it should be straightforward to formally extend our language with such constructs, we will sometimes use them, mostly in examples. Having defined expressions, we introduce:

Definition 2.2 (Stores, Threads, Networks). *The sets $\mathcal{S}$ of stores, $\mathcal{T}$ of threads, and $\mathcal{P}$ of networks are given in Fig. 1 (bottom).*

A network is a (possibly empty) composition of objects, where composition $P \mid Q$ and restriction $(\nu n)P$ are introduced with their usual meaning (cf., the π-calculus). An object $n[M ; s ; t]$ encapsulates, under the object name n, some methods M (passive code), a store s (that holds the object local state), and some threads t (active, running code). A store s is a bag of pairs tag - value. Each value is recorded in a store under an access tag (a name), represented by $a\langle v \rangle$, where a is the tag and v is the value. A store may possibly record several values under the same name a, so that *e.g.*, $a\langle 1 \rangle \mid a\langle 2 \rangle$ is a valid store. A thread $c\langle e \rangle$ is uniquely identified by its identifier c and holds a running piece of code, namely the expression e. Threads are spawned when methods are called, and may run concurrently with other independent threads in the same object or network.

Objects in a system are given unique names, so that, for instance, the network $n[M ; s ; t] \mid n[M' ; s' ; t']$ denotes the same network as $n[M \mid M' ; s \mid s' ; t \mid t']$. Identities between networks such as this one (the **Split** law) are formally captured by structural congruence, defined below. By $fn(P)$ (resp. $fn(t)$, $fn(s)$, etc.) we denote the set of free names in process P (resp. thread t, store s, etc.), defined as expected. We use A, B, C to range over $\mathcal{M} \cup \mathcal{S} \cup \mathcal{T} \cup \mathcal{P}$ (the $\mid$-"composable" entities).

Definition 2.3 (Structural Congruence). *Structural congruence, noted $\equiv$, is the least congruence relation on networks, methods, and threads, such that*

$$
\begin{array}{ll}
A \equiv A \mid \mathbf{0} & n[M ; s ; t] \mid n[N ; r ; u] \equiv n[M \mid N ; s \mid r ; t \mid u] \\
A \mid (B \mid C) \equiv (A \mid B) \mid C & n[M ; s ; t] \equiv n[N ; r ; u] \textbf{ if } M \equiv N, s \equiv r, t \equiv u \\
B \mid A \equiv A \mid B & (\nu m)(\nu n)P \equiv (\nu n)(\nu m)P \\
(\nu n)\mathbf{0} \equiv \mathbf{0} & (\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \textbf{ if } n\#fn(P)
\end{array}
$$

We use *n#S* (resp. *S#S'*) to denote that $n \notin S$ (resp. that S and S' are disjoint). To lighten our notation, we avoid writing **0** in objects slots, leaving the corresponding place holder blank. For example, $n[M ; \mathbf{0} ; \mathbf{0}]$ will be frequently written simply as $n[M ; \; ; \;]$.

The operational semantics of networks is defined by suitable transition relations. The labeled transition system in Fig. 2 specifies evaluation of expressions. A remote method call reduces to a wait expression on a fresh (thread) name c. Such wait expression will reduce to the returned value, upon thread completion. Notice also how **let** introduced expressions are evaluated concurrently, until each one reduces to a value. For object networks, the operational semantics specifies a remote method invocation mechanism by means of the transition system in Fig. 3. Servicing a method call causes a new thread to be spawned at the callee object's location, to execute the method's body. Meanwhile, the thread that originated the call suspends, waiting for a reply. Such a reply will be sent back to the caller, after the handling thread terminates. A $n[M]$ labeled transition, caused by the evaluation of a **new**$[M]$ expression, triggers the creation of a new object. Thus, labels in transitions express the actions that objects may engage into.

Definition 2.4. *Labels $\mathcal{L}$ are given by:* $\alpha ::= \tau \mid \overline{n.1_c(v)} \mid n.c(v) \mid a?(v) \mid a!(v) \mid n[M]$.

Definition 2.5 (Evaluation and Reduction). Evaluation, *noted* $e \xrightarrow{\alpha} e'$, *is the relation defined on expressions by the labeled transition system in Figure 2.* Reduction, *noted* $P \to Q$, *is the relation defined on networks by the transition system in Figure 3.*

We use $\Rightarrow$ for the the reflexive transitive closure of $\to$. Notice the role of $\equiv$ in reduction, in particular the **Split** law, so that each rule may mention just the parts of objects that are relevant for each interaction. An idle object may only become active as effect of an incoming method call, issued by a running thread. An object $n[M ; s ; t]$ such that $t \equiv \mathbf{0}$, is said to be *idle*, since it contains no running threads. Likewise, a network is idle if all of its objects are idle. We set: idle$(P) \triangleq$ *For all Q. if $P \equiv (\nu\overline{m})(n[\; ; \; ; t] \mid Q)$ then $t \equiv \mathbf{0}$*.

Example 2.6. We sketch a toy scenario of service composition, where several sites co-operate to provide a travel booking service. First, there is an object F implementing a service for finding and booking flights. It provides three methods: `flight` to look for and reserve a flight, `book` to commit the booking, and `free` to release a reservation. A similar service is provided by object H, used for booking hotel rooms.

$$F \triangleq f[\, \texttt{flight}() = \cdots \mid \texttt{book}() = \cdots \mid \texttt{free}() = \cdots ; \; ; \;]$$
$$H \triangleq h[\;\; \texttt{hotel}() = \cdots \mid \texttt{book}() = \cdots \mid \texttt{free}() = \cdots ; \; ; \;]$$

$$G \triangleq gw[\, \texttt{pay}(s) = \mathbf{if}\ bk.\texttt{debit}()\ \mathbf{then}\ s.\texttt{book}()\ \mathbf{else}\ s.\texttt{free}() ; \; ; \;]$$

$$B \triangleq br[\, \texttt{flight}() = f.\texttt{flight}() \mid \texttt{hotel}() = h.\texttt{hotel}() \mid$$
$$\texttt{order}() = (gw.\texttt{pay}(f); gw.\texttt{pay}(h)) ; \; ; \;]$$

We elide method implementations in F and G, but assume that the operations must be called in good order to avoid disruption, namely that after calling `flight`, a client is supposed to call either `book` or `free`. The broker B, that implements the front-end of the whole system, is client of F and H, and also of a payment gateway G. The gateway books items if succeeds in processing their payment through a remote bank

service named *bk*. Our travel booking service, available at *br*, is used by first invoking the `flight` and `hotel` operations in any order. In fact, these operations may be called concurrently, since they trigger separate computations. Afterwards, the `order` operation may be invoked to book and pay for both items, delegating access to *f* and *h* to the gateway. The session will then terminate, and the broker becomes ready for another round. We will see below how these usage patterns may be specified by typing, and the type of the whole system compositionally defined from the types of its components.

The operational semantics assumes and preserves some constraints on networks. In an object $n[M ; s ; t]$ no more than a method with the same label may occur in M, and no more than a thread with the same name may occur in t. Objects in a network are uniquely named, also all threads in a system are uniquely named. In general, a network P is *well-defined* if all threads occurring in P have distinct names, all methods in objects have distinct labels, and for each thread name c there is at most one occurrence in P of a wait expression $n.c()$. It is immediate that if P is well-defined and $P \equiv Q \mid R$ then both Q and R are well-defined. On the other direction, the same does not hold in general, e.g., P and Q might clash in method and thread names. We then define by $ft(P)$ the set of free thread names in P, and by $lab(P,n)$ the set of method labels of object n in P:

$$c \in ft(P) \qquad \textit{iff } P \equiv (\nu\overline{m})(n[\ ; \ ; c\langle e\rangle] \mid Q) \textit{ and } c\#\overline{m}$$
$$\mathtt{l} \in lab(P,n) \quad \textit{iff } P \equiv (\nu\overline{m})(n[\mathtt{l}(x) = e ; \ ;] \mid Q) \textit{ and } n\#\overline{m}$$

Definition 2.7. *We assert* $P\|Q \triangleq ft(Q)\#ft(P)$ *and for all n.* $lab(P,n)\#lab(Q,n)$.

$P\|Q$ means that P and Q are composable, in the sense that if $P\|Q$, and both P and Q are well-defined, so is $P \mid Q$. Notice that $P\|Q$ *does not* imply $fn(P)\#fn(Q)$. We also have

Lemma 2.8. *If P is a well-defined network and* $P \rightarrow Q$ *then Q is a well-defined network.*

Henceforth, we assume networks to be always well-defined. It is useful to introduce external action transitions, that extend the reduction semantics with labels $n.\mathtt{l}_c(m)$ and $n.c(r)$, to capture incoming method calls from the environment, and their replies.

Definition 2.9. *External actions of networks are defined by the labeled transitions:*

$$(\nu\overline{m})(n[\mathtt{l}(x) = e ; \ ;] \mid R) \xrightarrow{n.\mathtt{l}_c(u)} (\nu\overline{m})(n[\mathtt{l}(x) = e ; \ ; c\langle e\{x{\leftarrow}u\}\rangle] \mid R) \quad [c \textit{ fresh and } n,u\#\overline{m}]$$

$$(\nu\overline{m})(n[\ ; \ ; c\langle r\rangle] \mid R) \xrightarrow{\overline{n.c(r)}} (\nu\overline{m}\backslash r)(n[\ ; \ ;] \mid R) \qquad\qquad [c,n\#\overline{m}]$$

Even well-defined networks may get *stuck* if a method call is being issued, but the called object does not offer the requested method. We say P is stuck if stuck(P) holds, where

$$\text{stuck}(P) \triangleq \textit{Exists } \overline{m},Q,e,e'. \ P \equiv (\nu\overline{m})(p[\ ; \ ; c\langle e\rangle] \mid Q) \textit{ and } e \xrightarrow{\overline{n.\mathtt{l}_c(v)}} e' \textit{ and } \mathtt{l} \notin lab(Q,n)$$

Service based systems, as we are modeling here, may easily get stuck, if not carefully designed. As in (untyped) object oriented languages, *message-not-understood* errors may arise whenever an object does not implement an invoked method. However, in our present context stuck states may also arise if method calls are not coordinated (do not respect protocols) and timing errors occur (for example, due to races, *e.g.*, competing calls to the same non-shareable method). The presence of state in objects creates history dependencies on resource usage, and introduces a grain of resource sensitiveness in our model, as discussed in the Introduction, and illustrated in the next example.

Example 2.10. Consider the object *S* defined thus

$$S \triangleq server[\ \text{init}() = s!(\texttt{nil})$$
$$\text{open}() = \textbf{let } r = pool.\text{alloc}() \textbf{ in } (s?; s!(r))$$
$$\text{use}() = \textbf{let } r = s? \textbf{ in } (r.\text{use}(); s!(r))$$
$$\text{close}() = \textbf{let } x = s? \textbf{ in } (pool.\text{free}(x); s!(\texttt{nil}))\ ;\ \ ;\ ]$$

Our server *S* is a spooler that offers certain specific service by relying on a remote resource pool to fetch appropriate providers. All resources (*e.g.*, printers) in the pool (*pool*) are supposed to implement the operation of interest (`use`). The server provides the `use` service repeatedly to clients, by forwarding it through a locally cached reference. First, the server is initialized: by calling the `init` method the local reference is set to `nil`. Afterwards, a client must open the service by calling the `open` method before using it (so that the server can acquire an available resource), and close it after use by calling the `close` method (so that the server may release the resource). The server implements these operations by accessing the pool through its `alloc` and `free` methods. The internal state of the server, hidden to clients, will always be either of the form $s(\texttt{nil})$, or $s(r)$ where *r* is a reference to an allocated resource. Notice how the idiom $\textbf{let } r = s? \textbf{ in } (\cdots; s!(r))$ expresses retrieving *r* from the local state, using it (in the $\cdots$ part), and storing it back again. The protocols described above must be strictly followed to avoid runtime errors, due to resource non-availability. This would occur, *e.g.*, if the `use` operation is invoked right after `close`, an attempt to call the `use` method on a `nil` reference will cause the system to crash. Our type system, presented in detail below, prevents erroneous behaviors of this sort to happen, by ensuring that all services in a network conform to well-defined resource usage protocols.

3 Spatial-Behavioral Types

In this section, we present a type system to discipline interactions on networks of objects, as motivated above. A type *T* in our system describes a usage pattern for an object. Typically, an assertion of the form $n : T$ states that the object *n* may be safely used as specified by the type *T*. In general, the type of a composite system is expressed by an assertion $n_1 : T_1 \mid \ldots \mid n_k : T_k$, specifying types of various objects. Such an assertion (or typing environment) states that the system provides *independent* services at the names n_i, each one able to be safely used as specified by the type T_i respectively.

Definition 3.1 (Types). *The set $\mathcal{T}$ of types is defined by the following abstract syntax:*

$$
\begin{array}{llll}
T, U, V ::= & \in \mathcal{T} & \text{(Types)} & \\
& \textbf{stop} & \text{(Stop)} & \mid T \mid U & \text{(Spatial Composition)} \\
\mid & T \wedge U & \text{(Conjunction)} & \mid T; U & \text{(Sequential Composition)} \\
\mid & T^\circ & \text{(Owned)} & \mid \texttt{1}(U)V & \text{(Method)} \\
\mid & \alpha & \text{(Variable)} & \mid c(n : U)V & \text{(Thread)} \\
\mid & \textbf{rec } \alpha.T & \text{(Recursion)} & &
\end{array}
$$

We may intuitively explain the meaning of the various kinds of types, by interpreting them as certain properties of objects. An object satisfies **stop** if it is idle. An object

satisfies $n : T \mid U$ if it can independently (in terms of resource separation) satisfy both $n : T$ and $n : U$. We may also understood such a typing as the specification of two independent views for the object n. An object satisfies $n : T \wedge U$ if it may satisfy both $n : T$ and $n : U$, although not concurrently. Such an object may be used either as specified by $n : T$ or as specified by $n : U$, being the choice made by the client. An object satisfies $n : T;U$ if can satisfy both $n : T$ and $n : U$, in sequence. In particular, it will only be obliged to satisfy $n : U$ after being used as specified by $n : T$. The owned type $n : T^{\circ}$ means that the object may be used as specified by T, and furthermore that this T view is *exclusively owned*. For example, a reference of type $n : T^{\circ}$ may be stored in the local state of an object, or returned by a method call, although a reference of type $n : T$ may not, because of possible liveness constraints associated to the type T. This will become clearer in the precise semantic definitions below.

An object satisfies $n : \mathtt{l}(U)V$ if it offers a method $\mathtt{l}$ that whenever passed an argument of type U is ensured to return back a result of type V, and exercise, during the call, an use of the argument conforming to type U. Thus, method types specify both safety and liveness properties. The $c(n : U)V$ types talk about running threads, and are not expected to type source programs, but are useful to define the semantics of method types, as explained below. Recursive types are interpreted as usual. We will not address in detail recursive types in our technical development, for their treatment is fairly independent from the features we want to focus, and should not raise special difficulties.

A typing environment $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \sigma, \delta)$ is a finite partial mapping from $\mathcal{N} \cup \mathcal{V}$ to $\mathcal{T}$. We write $\eta_1 : T_1, \ldots, \eta_n : T_n$ for the typing environment $\mathbf{A}$ with domain $\mathfrak{D}(\mathbf{A}) = \{\eta_1, \ldots, \eta_n\}$ such that $\mathbf{A}(\eta_i) = T_i$, for $i = 1, \ldots, n$. Type operations $\mathtt{stop}$, $(T \mid U)$, $(T \wedge U)$, $(T;U)$ and T° extend to typing environments as follows. $\mathtt{stop}$ denotes any typing environment (including the empty one) that assigns $\mathtt{stop}$ to any element in its domain. Given $\mathbf{A}$ and $\mathbf{B}$ such that $\mathfrak{D}(\mathbf{A}) = \mathfrak{D}(\mathbf{B})$, we define environments $\mathtt{stop}$, $(\mathbf{A} \mid \mathbf{B})$, $(\mathbf{A};\mathbf{B})$, $(\mathbf{A} \wedge \mathbf{B})$, and $\mathbf{A}^{\circ}$, all with domain $\mathfrak{D}(\mathbf{A})$, such that, for all $\eta \in \mathfrak{D}(\mathbf{A})$, we have

$$\mathtt{stop}(\eta) \triangleq \mathtt{stop} \qquad (\mathbf{A} \mid \mathbf{B})(\eta) \triangleq \mathbf{A}(\eta) \mid \mathbf{B}(\eta) \qquad (\mathbf{A};\mathbf{B})(\eta) \triangleq \mathbf{A}(\eta);\mathbf{B}(\eta)$$
$$(\mathbf{A} \wedge \mathbf{B})(\eta) \triangleq \mathbf{A}(\eta) \wedge \mathbf{B}(\eta) \qquad \mathbf{A}^{\circ}(\eta) \triangleq \mathbf{A}(\eta)^{\circ}$$

Given a sequence $\overline{T} = T_1, \ldots, T_n$ of types (or typing environments) we denote by $\Pi(\overline{T})$ the type (or typing environment) $(T_1 \mid \cdots \mid T_n)$. Our type system is based on the following forms of formal judgments:

$\mathbf{A} <: \mathbf{B}$	(Subtyping)	$e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[U]$	(Expressions)
$[M;t] :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[U]$	(Objects)	$P :: \mathbf{A} \triangleright \mathbf{B}$	(Networks)

In an expression typing judgment, e is the expression to be typed, $\mathbf{A}$ and $\mathbf{B}$ are typing environments, and U is a type. The auxiliary type environments σ and δ keep information about effects on the local state of objects, and will be further explained below (notice the $\mid$ symbol separating the global environments $\mathbf{A}$ and $\mathbf{B}$ from the state environments σ and δ in judgments, not to be confused with the $\mid$ type constructor). For networks, the typing judgment assigns to the network P an "assume-guarantee" assertion of the form $\mathbf{A} \triangleright \mathbf{B}$, cf. the adjunct of the composition operator of spatial logics [8]. If a judgment $P :: \mathbf{A} \triangleright \mathbf{B}$ is valid, then if P is composed with any network that satisfies the typing $\mathbf{A}$, one is guaranteed to obtain a network that satisfies the typing $\mathbf{B}$.

$$\mathbf{A} \mid \mathbf{B} <: \mathbf{B} \mid \mathbf{A}$$
$$(\mathbf{A} \mid \mathbf{B}) \mid \mathbf{C} <:> \mathbf{A} \mid (\mathbf{B} \mid \mathbf{C})$$
$$(\mathbf{A};\mathbf{B}) \mid (\mathbf{C};\mathbf{D}) <: (\mathbf{A} \mid \mathbf{C});(\mathbf{B} \mid \mathbf{D})$$
$$\mathbf{A};(\mathbf{B};\mathbf{C}) <:> (\mathbf{A};\mathbf{B});\mathbf{C}$$

$$\mathbf{stop};\mathbf{A} <:> \mathbf{A}$$
$$\mathbf{A};\mathbf{stop} <:> \mathbf{A}$$

$$\mathbf{stop} \mid \mathbf{A} <:> \mathbf{A}$$
$$\mathbf{A} \wedge \mathbf{B} <: \mathbf{A}$$
$$\mathbf{A} \wedge \mathbf{B} <: \mathbf{B}$$

$$\eta : \mathbf{rec}\, \alpha.U <:> \eta : U\{\alpha \leftarrow \mathbf{rec}\, \alpha.U\}$$

$$\mathbf{A} <: \mathbf{A}$$
$$\mathbf{A}^\circ <: \mathbf{stop}$$
$$\mathbf{A}^\circ <: \mathbf{A}$$
$$\mathbf{A}^\circ <: \mathbf{A}^{\circ\circ}$$
$$\mathbf{stop} <: \mathbf{stop}^\circ$$
$$(\mathbf{A} \mid \mathbf{B})^\circ <:> \mathbf{A}^\circ \mid \mathbf{B}^\circ$$
$$\mathbf{A}^\circ;\mathbf{B} <:> \mathbf{A}^\circ \mid \mathbf{B}$$

$$\frac{\mathbf{A} <: \mathbf{B}}{\mathbf{A}^\circ <: \mathbf{B}^\circ}$$

$$\frac{\mathbf{A} <: \mathbf{B} \quad \mathbf{A} <: \mathbf{C}}{\mathbf{A} <: \mathbf{B} \wedge \mathbf{C}}$$

$$\frac{\mathbf{A} <: \mathbf{B} \quad \mathbf{B} <: \mathbf{C}}{\mathbf{A} <: \mathbf{C}}$$

$$\frac{\mathbf{A} <: \mathbf{B}}{\mathbf{A} \mid \mathbf{C} <: \mathbf{B} \mid \mathbf{C}}$$

$$\frac{\mathbf{A} <: \mathbf{B}}{\mathbf{A};\mathbf{C} <: \mathbf{B};\mathbf{C}}$$

$$\frac{\mathbf{A} <: \mathbf{B}}{\mathbf{C};\mathbf{A} <: \mathbf{C};\overline{\mathbf{B}}}$$

$$\frac{\eta : U <: \eta : V}{\eta : \mathbf{rec}\, \alpha.U <: \eta : \mathbf{rec}\, \alpha.V}$$

Fig. 4. Subtyping Rules

$$\mathbf{nil} :: \mathbf{A} \mid \sigma \triangleright \mathbf{A} \mid \sigma[\mathbf{stop}]$$

$$v :: v : T^\circ \mid \sigma \triangleright \mid \sigma[T]$$

$$a? :: \mid \sigma, a : T \triangleright \mid \sigma[T]$$

$$a!(v) :: v : T^\circ \mid \sigma \triangleright \mid \sigma, a : T[\mathbf{stop}]$$

$$v.\mathtt{l}(u) :: v : \mathtt{l}(U)V \mid u : U \mid \sigma \triangleright \mid \sigma[V]$$

$$n.c(v) :: n : c(U)V \mid v : U \mid \sigma \triangleright \mid \sigma[V]$$

$$\frac{[M;\mathbf{0}] :: \mathbf{A}^\circ \mid \; \triangleright \mid [T]}{\mathbf{new}[M] :: \mathbf{A}^\circ \mid \; \triangleright \mid [T^\circ]}$$

$$\frac{\mathbf{A} <: \mathbf{A}' \quad e :: \mathbf{A}' \mid \sigma \triangleright \mathbf{B}' \mid \delta[V'] \quad \mathbf{B}' <: \mathbf{B} \quad V' <: V}{e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[V]}$$

$$\frac{e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[U] \quad e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[V]}{e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[U \wedge V]}$$

$$\frac{e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[V]}{e :: \mathbf{A} \mid \mathbf{C} \mid \sigma, \phi \triangleright \mathbf{B} \mid \mathbf{C} \mid \delta, \phi[V]}$$

$$\frac{e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[V]}{e :: \mathbf{A};\mathbf{C} \mid \sigma \triangleright \mathbf{B};\mathbf{C} \mid \delta[V]}$$

$$\frac{e_i :: \mathbf{B}_i \mid \sigma_i \; \triangleright \mid \delta_i[V_i] \quad e_i \#_{\mathfrak{D}(\Pi(\overline{\sigma}))} e_j \; (i \neq j) \quad f :: \mathbf{C}, \overline{x : V^\circ} \mid \Pi(\overline{\delta}) \; \triangleright \; \mathbf{E}, \overline{x : \mathbf{stop}} \mid \phi[U]}{\mathbf{let}\, \overline{x = e}\, \mathbf{in}\, f :: \Pi(\overline{\mathbf{B}});\mathbf{C} \mid \Pi(\overline{\sigma}) \triangleright \mathbf{E} \mid \phi[U]}$$

Fig. 5. Typing Rules (Expressions).

$$[M;\mathbf{0}] :: \mathbf{A} \mid \sigma \triangleright \mathbf{A} \mid \sigma[\mathbf{stop}]$$

$$\frac{M \equiv (N \mid \mathtt{l}(x) = e) \quad e :: \mathbf{A}, x : U \mid \sigma \triangleright \mathbf{B}, x : \mathbf{stop} \mid \delta[V]}{[M;\mathbf{0}] :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[\mathtt{l}(U)V]}$$

$$\frac{e :: \mathbf{A}, x : U \mid \sigma \triangleright \mathbf{B}, x : \mathbf{stop} \mid \delta[V]}{[M; c\langle e\{x \leftarrow m\}\rangle] :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[c(m : U)V]}$$

$$\frac{[M';t'] :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[U] \quad [M'';t''] :: \mathbf{C} \mid \sigma' \triangleright \mathbf{D} \mid \delta'[V]}{[M' \mid M'';t' \mid t''] :: \mathbf{A} \mid \mathbf{C} \mid \sigma, \sigma' \triangleright \mathbf{B} \mid \mathbf{D} \mid \delta, \delta'[U \mid V]}$$

$$\frac{[M;t] :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[U] \quad [M;\mathbf{0}] :: \mathbf{B} \mid \delta \triangleright \mathbf{C} \mid \phi[V]}{[M;t] :: \mathbf{A} \mid \sigma \triangleright \mathbf{C} \mid \phi[U;V]}$$

$$\frac{[M;\mathbf{0}] :: \mathbf{A}^\circ \mid \; \triangleright \mid [T]}{[M;\mathbf{0}] :: \mathbf{A}^\circ \mid \; \triangleright \mid [T^\circ]}$$

$$\mathbf{0} :: \mathbf{A} \triangleright \mathbf{A}$$

$$\frac{[M;t] :: \mathbf{A} \mid \overline{s_i : V_i} \triangleright \mathbf{B} \mid \delta[T]}{n[M; \overline{s_i\langle n_i\rangle}; t] :: \mathbf{A} \mid \Pi(n_i : V_i^\circ) \triangleright n : T}$$

$$\frac{P :: \mathbf{A} \triangleright \mathbf{B} \quad Q :: \mathbf{C} \triangleright \mathbf{D} \quad P \| Q}{P \mid Q :: \mathbf{A} \mid \mathbf{C} \triangleright \mathbf{B} \mid \mathbf{C}}$$

$$\frac{P :: \mathbf{A} \triangleright \mathbf{B} \quad Q :: \mathbf{B} \triangleright \mathbf{C} \quad P \| Q}{P \mid Q :: \mathbf{A} \triangleright \mathbf{C}}$$

$$\frac{P :: \mathbf{A} \triangleright \mathbf{B} \quad n\#\mathbf{A}, \mathbf{B}}{(\nu n)P :: \mathbf{A} \triangleright \mathbf{B}}$$

$$\frac{\mathbf{A} <: \mathbf{A}' \quad P :: \mathbf{A}' \triangleright \mathbf{B}' \quad \mathbf{B}' <: \mathbf{B}}{P :: \mathbf{A} \triangleright \mathbf{B}}$$

Fig. 6. Typing Rules (Objects and Networks).

What does it mean for a network to satisfy a typing? As discussed above, types are interpreted as properties (sets of networks) expressible in a spatial logic. In Section 3.1 below we will present in detail a logical semantics of types, around which our soundness proofs are organized. First, we present our type system as a formal system, and explain from an intuitive perspective the various rules and main results. Our type system is composed by four sets of rules, to derive judgments of the four forms listed above. In Fig. 4 we present the subtyping rules. Subtyping, which holds between typing environments, is motivated by selected natural properties of types, and reflect valid semantic entailments in our logic (cf. Proposition 3.4). A first set of rules states that $(- \mid -)$ and **stop** define a commutative monoid. The rule $(\mathbf{A};\mathbf{B}) \mid (\mathbf{C};\mathbf{D}) \mathrel{<:} (\mathbf{A} \mid \mathbf{C});(\mathbf{B} \mid \mathbf{D})$ expresses the basic interaction principle between sequential and independent composition, allowing us to derive, *e.g.*, $\mathbf{A} \mid \mathbf{B} \mathrel{<:} \mathbf{A};\mathbf{B}$, expressing interleaving. The rules for $(-)^\circ$ are quite interesting, notice that $(-)^\circ$ and $(- \mid -)$ reveal a familiar algebraic structure. No so familiar is the rule $\mathbf{A}^\circ;\mathbf{B} \mathrel{<:} \mathbf{A}^\circ \mid \mathbf{B}$, asserting a key principle involving sequential composition and ownership: the owned usage $\mathbf{A}^\circ$ is not active (yet), and thus $\mathbf{B}$ cannot causally depend on it. A further set of rules express congruence principles, and unfolding of recursion.

Fig. 5 presents the typing rules for expressions. Intuitively, a expression typing judgment $e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta [U]$ means that e, when given a services conforming to $\mathbf{A}$, in a store conforming to σ will, after termination, yield a value of type U, while leaving a store conforming to δ, and the used services in a state where they may be still used as specified by $\mathbf{B}$. Notice that typing of expressions depends on typing of objects, through the rule for **new**$[M]$. To intuitively grasp the meaning of our rules, we should keep in mind that in a judgment $e :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta [U]$, the return type U, as well as the stored types σ, δ, are implicitly owned (we avoid writing, *e.g.*, U° in the return type $[U]$). So, in the rule for a value (name or variable), the value v may be returned only if its type is owned (T°). The same happens in the rule for a write $a!(v)$, where ownership of a T view of v is handed over from the thread to the store. Notice how read / write effects are recorded in the left (σ) and right (δ) environments. The rule for method call $v.\mathtt{l}(u)$ requires separation between the method server v and the argument u. However, it does not force them to be different objects: a general form of non-interference is here ensured by the spatial typing, stating that the method part and the argument part do not share resources. We also have some congruence rules, a subtyping rule, and a rule for **let**. In the **let** rule, each expression e_i is required not to interfere with a concurrent e_j ($e_i \# e_j$), by reading and writing in the local store. We assert $e \#_N e'$ whenever e and e' do not write $(a!(v))$ or read $(a?)$ using a common tag name a in N (*e.g.*, we have $a!();b? \#_{\{a\}} b!();c?$). This condition will be relaxed in Section 4, after the introduction of shared variables and types. Notice that the values returned from each e_i, whose evaluation depends on separate resources $\mathbf{B}_i$, are separate owned values, each one of type $V_i{}^\circ$.

In Fig. 6, we present the typing rules for objects and networks. Intuitively, if the judgement $[M \, ; \, t] :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta [U]$ is valid, it states that any object $n[M \, ; \, s \, ; \, t]$ where the store s satisfies σ, may be composed with any system satisfying $\mathbf{A}$ and be safely used according to type U. The residuals $\mathbf{B}$ and δ reflect the state of the external and local resources after U has been exercised. Under this intuitive reading, all the rules for objects are already quite transparent, and the same remark also applies to the rules for networks. We discuss a bit the rule for object introduction. The rule requires that all state

elements are distinctly named, and that each of the stored values n_i is actually owned by the object (typed by $V_i{}^\circ$). Although in a perhaps subtle way, subtyping plays a key role in the derivation of expression, objects and network judgments, the factorization of a substantial amount of structural reasoning in the subtyping relation contributed to keep our typing rules reasonably clean (we omit the obvious rule for subtyping object judgments).

Example 2.6 (*continued*). We now assign types to the system components. For F and H we may expect the typings $F :: \triangleright f : T_f$ and $H :: \triangleright h : T_h$, where we consider $T_f \triangleq$ $\mathbf{rec}\,\alpha.\mathtt{flight}();(\mathtt{book}()\wedge\mathtt{free}());\alpha$ and $T_h \triangleq \mathbf{rec}\,\alpha.\mathtt{hotel}();(\mathtt{book}()\wedge\mathtt{free}());\alpha$. For the gateway G, let $G :: bk : T_{bank} \triangleright gw : T_{gw}$ where $T_{bk} \triangleq \mathbf{rec}\,\alpha.\mathtt{debit}()\mathbf{bool};\alpha$ and $T_{gw} \triangleq \mathbf{rec}\,\alpha.\mathtt{pay}(\mathtt{book}()\wedge\mathtt{free}());\alpha$. Set $T_{br} \triangleq \mathbf{rec}\,\alpha.(\mathtt{flight}()\mid\mathtt{hotel}());\mathtt{order}();\alpha$. Now, the following judgment is derivable: $(F \mid H \mid G \mid B) :: bk : T_{bk} \triangleright br : T_{br}$. It asserts that $(F \mid H \mid G \mid B)$, when composed with any system providing the T_{bk} type at bk, will be safe for use at br as specified by T_{br}. Such typing may be obtained compositionally in many ways. A possible factoring is between broker $B :: gw : T_{gw}, f : T_f, h : T_h \triangleright br : T_{br}$ and back-end $(G \mid H \mid F) :: bk : T_{bk} \triangleright gw : T_{gw}, f : T_f, h : T_h$.

We define the following variant of the Kleene iterator: $T^\otimes \triangleq \mathbf{rec}\,\alpha.(T;\alpha)^\circ$. Notice that we have $T^\otimes <\!:\!> (T;T^\otimes)^\circ <\!: \mathtt{stop} \wedge (T;T^\otimes)$. Hence, $T^\otimes$ can be unfolded infinitely many times into copies of T (as T^* does), but also be stored and returned by method calls, since it is an owned type (while T^* may not).

Example 2.10 (*continued*). For the spooler S, we propose the following typings. First, we abbreviate $T_{res} \triangleq (\mathtt{use}())^\otimes$, $T_{rm} \triangleq \mathbf{rec}\,\alpha.\mathtt{alloc}()T_{res};\mathtt{free}(T_{res}{}^\circ);\alpha$ and $T_{srv} \triangleq$ $\mathbf{rec}\,\alpha.\mathtt{open}();T_{res};\mathtt{close}();\alpha$. Then the following is derivable: $S :: pool : T_{rm} \triangleright server :$ T_{srv}. Notice how owner types $(T_{res}{}^\circ)$ are used to express ownership transfer of resources from the pool to the spooler and back. In general, we would expect a resource pool such as the one expected at *pool* to be shared by multiple users, while here the T_{rm} type just captures a very particular sequential usage. We will return to this in Section 4 below.

The safety properties ensured by our type system may be formally expressed in many ways. The fundamental consequences of typing are stuck-freeness, from which, as discussed in Section 2, other properties follows, such as race absence for unshareable resources, and conformance to usage protocols. We can thus already hint to our main soundness result, in a somewhat specific form.

Claim. Let $P :: \triangleright n : \mathtt{l}(\mathtt{stop})\mathtt{stop}$. Then there is Q such that $P \xrightarrow{n.\mathtt{l}_c(\mathtt{nil})} Q$ and for all R such that $Q \Rightarrow R$ it is not the case that $stuck(R)$.

This states that any network typed by $n : \mathtt{l}(\mathtt{stop})\mathtt{stop}$ offers a method $\mathtt{l}$ at object n that, after invoked, is ensured to induce a well-behaved distributed computation. More general soundness results follow as direct consequence of the semantics of types developed in the next section.

3.1 Logical Semantics of Types

The intended semantics for a typing environment $\mathbf{A}$ is that it denotes a certain property $[\![\mathbf{A}]\!]$, in the sense that if P is assigned type $\mathbf{A}$, then soundness of our type system ensures that $P \in [\![\mathbf{A}]\!]$, or, in terms of logical satisfaction, that $P \models \mathbf{A}$. In fact, we will not interpret types as properties directly, but will rather embed types in a more primitive spatial logic,

$$P \models \mathcal{A} \wedge \mathcal{B} \quad \textit{iff} \quad P \models \mathcal{A} \text{ and } P \models \mathcal{B}$$

$$P \models \mathcal{A} \mid \mathcal{B} \quad \textit{iff} \quad \textit{exists } Q, R.\, P \equiv Q \mid R \text{ and } Q \models \mathcal{A} \text{ and } R \models \mathcal{B}$$

$$P \models \mathcal{A} \triangleright \mathcal{B} \quad \textit{iff} \quad \textit{for all } Q.\, \textit{if } (P \| Q) \text{ and } Q \models \mathcal{A} \text{ then } P \mid Q \models \mathcal{B}$$

$$P \models \forall x.\mathcal{A} \quad \textit{iff} \quad \textit{for all } n.\, P \models \mathcal{A}\{x \leftarrow n\}$$

$$P \models \textbf{stop} \quad \textit{iff} \quad idle(P)$$

$$P \models \mathcal{A}^\circ \quad \textit{iff} \quad P \models \mathcal{A} \text{ and } P \models \textbf{stop}$$

$$P \models \mathcal{A}; \mathcal{B} \quad \textit{iff} \quad \textit{exists } Q, R.\, P \equiv Q \mid R \text{ and } Q \models \mathcal{A} \text{ and}$$
$$\textit{for all } Q'.\, \textit{if } Q \overset{\mathcal{A}}{\longmapsto} Q' \text{ then } Q' \mid R \models \mathcal{B}$$

$$P \models n : \mathtt{l}(m) \quad \textit{iff} \quad \textit{exists } Q.\, P \overset{n.\mathtt{l}_c(m)}{\longrightarrow} Q$$

$$P \models (\nu)\mathcal{A} \quad \textit{iff} \quad \textit{exists } Q.\, P \equiv (\nu \overline{m})Q \text{ and } Q \models \mathcal{A} \text{ and } \overline{m} \# fn(\mathcal{A})$$

$$P \models n : c(\mathcal{A}, r) \quad \textit{iff} \quad \textit{for all } R, Q.\, \textit{if } R \| P \text{ and } R \models \mathcal{A} \text{ and } P \mid R \Rightarrow Q \text{ then}$$
$$\neg stuck(Q) \text{ and}$$
$$\textit{for all } Q', r.\, \textit{if } Q \overset{\overline{n.c(r)}}{\longrightarrow} Q' \text{ then exists } R', P'.\, Q' \equiv R' \mid P' \text{ and } R \overset{\mathcal{A}}{\longrightarrow} R'$$

$$P \overset{\textbf{stop}}{\longmapsto} P \qquad \frac{P \overset{\mathcal{U}}{\longmapsto} Q}{P \overset{\mathcal{U} \wedge \mathcal{V}}{\longmapsto} Q} \qquad \frac{P \overset{\mathcal{V}}{\longmapsto} Q}{P \overset{\mathcal{U} \wedge \mathcal{V}}{\longmapsto} Q} \qquad \frac{P \overset{\mathcal{U}\{x \leftarrow n\}}{\longmapsto} Q}{P \overset{\forall x.\mathcal{U}}{\longmapsto} Q} \qquad \frac{P \overset{n.\mathtt{l}_c(m)}{\longrightarrow} Q}{P \overset{n:\mathtt{l}(m)}{\longmapsto} Q}$$

$$\frac{P \equiv (\nu \overline{m})R \quad R \overset{\mathcal{U}}{\longmapsto} Q}{P \overset{(\nu)\mathcal{U}}{\longmapsto} Q} \qquad \frac{P \equiv P_1 \mid P_2 \quad P_1 \overset{\mathcal{U}}{\longmapsto} Q_1 \quad P_2 \overset{\mathcal{V}}{\longmapsto} Q_2 \quad Q_1 \mid Q_2 \equiv Q}{P \overset{\mathcal{U} \mid \mathcal{V}}{\longmapsto} Q}$$

$$\frac{P \overset{\mathcal{U}}{\longmapsto} R \quad R \overset{\mathcal{V}}{\longmapsto} Q}{P \overset{\mathcal{U}; \mathcal{V}}{\longmapsto} Q} \qquad \frac{P \equiv R \mid Q \quad R \models \mathcal{U}}{P \overset{\mathcal{U}^\circ}{\longmapsto} Q} \qquad \frac{R \models \mathcal{U} \quad P \mid R \Rightarrow \overset{n.c(r)}{\longrightarrow} Q \mid R' \quad R \overset{\mathcal{U}}{\longmapsto} R'}{P \overset{n:c(\mathcal{U},r)}{\longmapsto} Q}$$

Fig. 7. Satisfaction and Typed Usage

so that each typing environment **A** is interpreted by a certain formula $\mathcal{A}$. The satisfaction predicate $\models$ is inductively defined on the structure of formulas, in such a way that $P \models \mathcal{A}$ implies that P enjoys certain general safety properties, in particular, stuck-freeness.

Definition 3.2 (Spatial Logic). *The set $\mathcal{F}$ of formulas is defined by:*

$$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{U}, \mathcal{V} ::= \mathcal{A} \wedge \mathcal{B} \mid \forall x.\mathcal{A} \mid \mathcal{A} \mid \mathcal{B} \mid \mathcal{A} \triangleright \mathcal{B} \mid \textbf{stop} \mid \mathcal{A}; \mathcal{B} \mid \mathcal{A}^\circ \mid n : c(\mathcal{A}, r) \mid n : \mathtt{l}(m) \mid (\nu)\mathcal{A}$$

As in [6, 7, 5], our logic includes (positive) first-order logic, the basic spatial operators of composition and its adjunct with their standard meanings, and certain specific operators, in particular some behavioral modalities. Instead of including action prefixing modalities, we introduce a general sequential composition formula of the form $\mathcal{A}; \mathcal{B}$, where $\mathcal{A}$ is interpreted both a property, and a *usage pattern*. Usage patterns are modeled by *typed usage*, a transition relation between networks and labeled by formulas, noted $P \overset{\mathcal{A}}{\longmapsto} Q$. The intuitive meaning of $P \overset{\mathcal{A}}{\longmapsto} Q$ is that if P is used as specified by $\mathcal{A}$, it may evolve to Q. Since satisfaction and typed usage are defined by mutual recursion, we present them in a single definition, for the sake of clarity.

Definition 3.3. Satisfaction, $P \models \mathcal{A}$, and typed usage, $P \overset{\mathcal{A}}{\longmapsto} Q$ are defined in Fig. 7.

To avoid clashes between fresh names introduced in the subsidiary transitions, the rule for $P \overset{\mathcal{U} \mid \mathcal{V}}{\longmapsto} Q$ is subject to the proviso $(fn(Q_1) \setminus fn(P_1)) \# (fn(Q_2) \setminus fn(P_2))$.

The semantics of $n : \mathtt{1}(m)$ and $n : c(\mathcal{A}, r)$ are defined from external actions of networks (Def 2.9). Intuitively, a network P satisfies formula $n : c(\mathcal{A}, r)$ if it contains a thread c that whenever passed a resource R satisfying $\mathcal{A}$, is guaranteed to always evolve in a stuck free way until a value r is returned, while exercising on R an usage as specified by $\mathcal{A}$. Thus, $n : c(\mathcal{A}, r)$ enforces both safety and liveness properties. Using these ingredients, we now define our interpretation of types. Given a type environment $\mathbf{A}$, we define a formula $\lceil \mathbf{A} \rceil$ by considering the embedding:

$$
\begin{array}{llll}
\lceil n : \mathtt{stop} \rceil & \triangleq \mathtt{stop} & \lceil n : U \mid V \rceil & \triangleq (\nu)(\lceil n : U \rceil \mid \lceil n : V \rceil) \\
\lceil n : U^{\circ} \rceil & \triangleq (\nu)\lceil n : U \rceil^{\circ} & \lceil n : U; V \rceil & \triangleq (\nu)(\lceil n : U \rceil; \lceil n : V \rceil) \\
\lceil n : \mathtt{1}(U)V \rceil & \triangleq \mathtt{stop} \wedge \forall u . n : \mathtt{1}(u); \lceil n : c(u : U)V \rceil & \lceil n : U \wedge V \rceil & \triangleq \lceil n : U \rceil \wedge \lceil n : V \rceil \\
\lceil n : c(u : U)V \rceil & \triangleq \forall r . c(\lceil u : U \rceil, r); \lceil r : V^{\circ} \rceil & \lceil \mathbf{A}, \mathbf{B} \rceil & \triangleq \lceil \mathbf{A} \rceil \mid \lceil \mathbf{B} \rceil
\end{array}
$$

Notice that all types are interpreted quite directly, except method and thread types, which are interpreted in terms of finer grain primitives. Building on this interpretation, we define validity of subtyping and typing judgments as follows:

$$
valid(\mathbf{A} <: \mathbf{B}) \triangleq \llbracket \mathbf{A} \rrbracket \subseteq \llbracket \mathbf{B} \rrbracket \qquad valid(P :: \mathbf{A} \triangleright \mathbf{B}) \triangleq P \in \llbracket \mathbf{A} \triangleright \mathbf{B} \rrbracket
$$

From now on, we will sometimes write typing environments where formulas are expected, having in mind the interpretation just presented. Our interpretation enjoys several nice properties. For example, the property stated in the *Claim* above (right before Section 3.1) is a direct consequence of the definition of the logical predicate $\models$. We can now state our main results:

Proposition 3.4 (Soundness of Subtyping). *For all* $\mathbf{A}$*,* $\mathbf{B}$*, if* $\mathbf{A} <: \mathbf{B}$ *then* $\llbracket \mathbf{A} \rrbracket \subseteq \llbracket \mathbf{B} \rrbracket$.

Proof. We may show the result for all properties, not just encodings of types, with the exception of congruence on the left for sequential composition. For that, we consider a stronger statement and prove, by induction on the derivation of $\mathbf{A} <: \mathbf{B}$, that if $\mathbf{A} <: \mathbf{B}$ and $\llbracket C \rrbracket \subseteq \llbracket \mathcal{D} \rrbracket$ then $\llbracket \mathbf{A}; C \rrbracket \subseteq \llbracket \mathbf{B}; \mathcal{D} \rrbracket$. ∎

Theorem 3.5 (Soundness of Typing). *If* $P :: \mathbf{A} \triangleright \mathbf{B}$ *is derivable then* $P \models \mathbf{A} \triangleright \mathbf{B}$.

Proof. The proof requires establishing a few facts about the satisfaction and typed usage relations, and some Lemmas stating soundness of typing for expressions and objects with respect to the intended notions of validity, which are given thus:

$$
\begin{aligned}
&valid([M; \overline{s_i \langle n_i \rangle}; t] :: \mathbf{A} \mid \sigma \triangleright \mathbf{B} \mid \delta[T]) \triangleq \\
&\quad \textit{ForAll } n . n[M ; \overline{s_i \langle n_i \rangle} ; t] \models (\mathbf{A} \mid \Pi(n_i : \sigma(n_i)^{\circ}) \triangleright (n : T); (\mathbf{B} \mid \Pi(n_i : \delta(n_i)^{\circ})))
\end{aligned}
$$

$$
\begin{aligned}
&valid(e :: \mathbf{A}, \overline{x : T} \mid \sigma \triangleright \mathbf{B}, \overline{x : S} \mid \delta[V]) \triangleq \\
&\quad \textit{Exists } \mathbf{C}, \mathbf{U} . \mathbf{A} <: \mathbf{C}; \mathbf{B} . \overline{x : T} <: \mathbf{U}; \overline{x : S} . \textit{ ForAll } n, \overline{s}, v, \overline{p} . \\
&\quad\quad n[; \overline{s_i \langle n_i \rangle} ; c \langle e\{x_1 \leftarrow p_1\} \cdots \{x_k \leftarrow p_k\} \rangle] \models \\
&\quad\quad\quad (\mathbf{A} \mid \overline{p : T} \mid \Pi(n_i : \sigma(n_i)^{\circ}) \triangleright c(\mathbf{C} \mid \mathbf{U}, v); (\mathbf{B} \mid \overline{p : S} \mid \Pi(n_i : \delta(n_i)^{\circ}) \mid v : V^{\circ})
\end{aligned}
$$

The definition of validity for expression judgments is a bit more involved, as it requires closure under substitution. Notice how our logic provides a suitable metalanguage in

which the properties of interest, explained above in intuitive terms, may be formally expressed rather succinctly. As in typical semantical soundness proofs of logical systems, the proof proceeds by checking that each rule preserves validity. ∎

The proof technique we have developed here may be seen as an instance of the general method of logical relations, well understood in the setting of functional programming, but still quite unexplored in concurrency. In a similar way, we build on a semantic interpretation of typed terms, which is defined by induction on types (as formulas), and then prove soundness by induction on typing derivations. Our result establishing validity under substitution for derivable expression typing judgments then plays the role of the so-called Basic Lemma in the logical relations method. Because types are directly interpreted as properties of networks, our soundness results allows us to conclude:

Proposition 3.6. *Let $P \models \mathbf{A}$ and $\mathbf{A} \mathrel{<:} \mathbf{B}; \mathbf{C}$. For all Q. if $P \xmapsto{\;\mathbf{B}\;} Q$ then $Q \models \mathbf{C}$.*

Proposition 3.6 is a semantic counterpart of the more familiar syntactic subject reduction property. In our case, it is an immediate consequence of the soundness of subtyping and the semantics of $\mathbf{B}; \mathbf{C}$. By interpreting the type $n : \mathtt{l}(U)V$, we also have:

Proposition 3.7 (Stuck Freeness). *Let $P :: \;\triangleright\; n : \mathtt{l}(U)V$ be derivable. Let R be such that $R \models m : U$ and $R \| P$. Then, for all Q such that $P \mid R \xrightarrow{n.\mathtt{l}_c(m)} Q$ it is not the case that* $\mathrm{stuck}(Q)$. *Moreover, if $Q \xrightarrow{\overline{n.c(r)}} Q'$ for some Q', then $Q' \models r : V^{\circ}$.*

4 Resource Sharing and Shared Types

Although our framework already seems fairly powerful, it still prevents useful forms of sharing to be typable. While race absence may be a desirable correctness property of concurrent programming in general, in many situations, races are not problematic if the involved resources may be safely shared (*e.g.*, read only variables). Moreover, many system resources are deliberately assumed to be raceful (*e.g.*, semaphores, buffers). Sharing is also particularly useful to allow local communication between different threads. In this section, we sketch how sharing is accommodated in our framework. No major extensions to the calculus are needed, we just add replicated methods to the basic syntax, and enrich structural congruence accordingly:

$$M ::= \cdots \mid *\mathtt{l}(x) = e \qquad\qquad \begin{aligned} *\mathtt{l}(x) = e &\equiv (*\mathtt{l}(x) = e \mid *\mathtt{l}(x) = e) \\ *\mathtt{l}(x) = e &\equiv (*\mathtt{l}(x) = e \mid \;\;\mathtt{l}(x) = e) \end{aligned}$$

The operational semantics in kept unmodified. Not surprisingly, more fundamental extensions relate to typing, and to the need to discipline shared access to the local store of objects. To that end, we assume that the local state of every object is classified in a unshared part (as in our basic model), and a shared part. The intent is that while the types of the values stored under a given tag in the unshared part may dynamically change (cf. the spooler example 2.10), values stored under a given tag in the shared part must satisfy a fixed invariant. Since the shared part may suffer interference from parallel running threads, we rely on this invariant to ensure soundness. To type such shared usages it is then useful to introduce shared types, defined $U! \triangleq \mathbf{rec}\,\alpha.(\mathbf{stop} \wedge U \wedge (\alpha \mid \alpha))$.

Shared types satisfy interesting subtyping principles, namely $U! \mathtt{<:>} U! \mid U!$, $U! \mathtt{<:} U$ and $U! \mathtt{<:} \mathbf{stop}$. The first principle allows a service of type $U!$ to be used simultaneously by an unbounded number of clients. We may also derive $U! \mathtt{<:} U; (U!)$.

For a first (trivial) example, consider the object $NL \triangleq nl[*\mathtt{null}() = \mathbf{nil} ; ;]$. It offers a method $\mathtt{null}$ that whenever called returns $\mathbf{nil}$. Clearly, the service provided by *NL* may be shared by an arbitrary number of clients, without incurring in any execution error (stuck state). So we expect the typing $NL :: \; \triangleright (\mathtt{null}()\mathbf{stop})!$ to be acceptable. For another example, consider the code: $BF \triangleq buf[*\mathtt{put}(x) = a!(x) \mid *\mathtt{get}() = a?; ;]$. Object *BF* implements a resource pool, that keeps in its local state a bunch of references for resources of a given type, say R°. Provided that the invariant $buf.a : R$ is maintained, we expect to assign a typing $BF :: \; \triangleright buf : (\mathtt{put}(R^\circ) \wedge \mathtt{get}()R)!$. This type allows any number of clients to share the pool, while using both methods, possibly concurrently. Another possible typing for *BF* is $BF :: \; \triangleright buf : \mathtt{put}(R^\circ)! \mid (\mathtt{get}()R)!$. This latter typing allows *BF* to be used as an (unordered) queue, in a context where a bunch of writers use the $buf : \mathtt{put}(R^\circ)!$ view, while a bunch of readers use the $buf : (\mathtt{get}()R)!$ view of *BF*. Notice that although the methods $\mathtt{put}$ and $\mathtt{get}$ interfere through the store, according to our intended semantics their are still separable by $(- \mid -)$ (up to changes in the store conforming to the sharing invariant $buf.a : R$).

Example 2.10 (*continued*). Given an implementation of a resource pool *RP* similar to *BF* above, typed by $RP :: \triangleright pool : T_p$ where $T_p \triangleq (\mathtt{free}(T_{res}^\circ) \wedge \mathtt{alloc}()R)!$, we expect then to type *S* with $S :: pool : T_p \triangleright server : T_{\mathtt{srv}}$.

We now illustrate the technical development needed to introduce sharing in our type system. Basically, we extend typing judgments with a further extra component (ε), that specifies (by typing) the invariants on admissible interferences through the stores. We illustrate our general approach with a few key rules.

$$a? :: \mid \sigma \mid \varepsilon, a : T \; \triangleright \; \mid \sigma[T] \qquad a!(v) :: v : T^\circ \mid \sigma \mid \varepsilon, a : T \; \triangleright \; \mid \sigma[T]$$

$$\frac{[M;t] :: \mathbf{A} \mid \overline{s_i : V_i} \mid \overline{p_i : U_i} \triangleright \mathbf{B}; \delta[T]}{n[M; \overline{s_i \langle n_i \rangle} \mid \overline{p_{i_j} \langle r_{i_j} \rangle};t] :: \mathbf{A} \mid \Pi(n_i : V_i^\circ) \mid \Pi(r_{i_j} : U_i^\circ) \mid \overline{n.p_i : U_i} \triangleright n : T}$$

The $\overline{n.p_i : U_i}$ (or $\overline{p_i : U_i}$) slot in the judgments specifies admissible interference from the environment, meaning that the store of object *n* may well be modified on cells $p_i \langle - \rangle$ provided that they will always contain values of type U_i°. To interpret the extended judgments, the logical predicate $\models$ is modified so that the interpretation of $c(\mathcal{A}, v)$ also takes into account interferences through the local store. Soundness proofs then follow the same lines of those above; a full treatment of these issues is left for an extended version of this paper.

5 Related Work and Discussion

We have presented a distributed object calculus able to model some essential aspects of service-based systems. However, the main focus of this work is on notions of spatial-behavioral typing, and their use to discipline interactions in distributed systems. Although the design of our calculus was influenced by several object calculi and related models [1, 3, 13], the distributed remote method invocation semantics adopted has not

been much explored, even if it seems a natural choice when modeling distributed services [12, 18]. In our case, such a model seemed to be fundamental for our spatial interpretation of types. Our type system enforces several safety properties, in particular availability (method calls are always served), and race absence with respect to unshareable resource access. Such properties result from the fact that our types are able to specify constraints on sequentiality of behavior, separation of resources, and dynamic propagation of ownership, in a compositional way. Compositionality is certainly a desirable property of any verification method, but it seems absolutely critical when one considers distributed service based systems, which are by nature open-ended, and dynamically assembled by relying on local interface specifications.

Formally, our type system can be seen as a fragment of a spatial logic for concurrency [6, 7, 5], where the composition operator plays a key role in ensuring resource control and non-interference. In our model, separation, up to structural congruence, cuts across the structure of objects, in order to separate both global and local resources. Our work draws inspiration on some specification techniques for the separation logics [20, 19], in particular our use of $\mid$ to talk about a form of resource separation, even if in the case of dynamic spatial logics the "resources" are active processes, quite unlike with the separation logics, that talk about the passive state (the heap). We have also introduced a sequential type composition operator and a owned type operator in our type structure. The owned type constructor, as we have studied here, seems to be new. Different notions of ownership and associated type systems have been proposed [4, 11], where ownership is considered a structural rather than a dynamic capability. The spatial interpretation of composition, together with owned types, also distinguishes our approach from other type systems for concurrent calculi that also include a composition operation [16, 10]. In those approaches, parallel composition is interpreted behaviorally, rather than as spatial separation, and subtyping corresponds to behavioral simulation, rather than to logical entailment; the same observation applies to [17]. Protocols definable in our type system are also reminiscent of session types [15], it would be interesting to see how sessions might be represented in this setting.

Unlike most works on type systems for concurrent and distributed calculi, we have adopted a semantic view of typing, and build on a logical relations technique to prove soundness. The (original) understanding of types as properties has not always been a common guiding principle in the design of types for concurrent calculi, where a syntactical view seems to be dominant (for an exception, we must refer to [9]). It would be also challenging to investigate how the compositional approach we have followed might also be applicable to (at least) certain kinds of security properties [2].

References

1. M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. *Inf. Comput.*, 125(2), 1996.
2. M. Bartoletti, P. Degano, and G. Ferrari. Enforcing secure service composition. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005)*, pages 211–223. IEEE Computer Society, 2005.

3. P. Di Blasio and K. Fisher. A calculus for concurrent objects. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 655–670. Springer-Verlag, 1996.
4. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.
5. L. Caires. Behavioral and Spatial Properties in a Logic for the Pi-Calculus. In Igor Walukiewicz, editor, *Proc. of Foundations of Software Science and Computation Structures'2004*, number 2987 in Lecture Notes in Computer Science. Springer Verlag, 2004.
6. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.
7. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). *Theoretical Computer Science*, 3(322):517–565, 2004.
8. L. Cardelli and A. D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In *27th ACM Symp. on Principles of Programming Languages*, pages 365–377. ACM, 2000.
9. G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 92–101. IEEE Computer Society, 2005.
10. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 45–57, 2002.
11. D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, pages 292–310, 2002.
12. M. Boreale et al. SCC: a Service Centered Calculus. In *Proceedings of WS-FM 2006, 3rd International Workshop on Web Services and Formal Methods*, Lecture Notes in Computer Science. Springer-Verlag, 2006.
13. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. *Electr. Notes Theor. Comput. Sci.*, 16(3), 1998.
14. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Inf. Comput.*, 173(1):82–120, 2002.
15. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.
16. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *in POPL 2001: 28th Annual Symposium on Principles of Programming Languages*, 2001.
17. A. Igarashi and N. Kobayashi. Resource usage analysis. In *POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 331–342, 2002.
18. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 2006.
19. P. W. O'Hearn. Resources, concurrency and local reasoning. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2004.
20. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Third Annual Symposium on Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Computer Society.