Local π-Calculus at Work: Mobile Objects as Mobile Processes*

Massimo Merro[1]**, Josva Kleist[2], and Uwe Nestmann[2]

[1] INRIA, Sophia-Antipolis, France
[2] BRICS - Basic Research in Computer Science,
Danish National Research Foundation,
Aalborg University, Denmark

Abstract. Obliq is a distributed, object-based programming language. In Obliq, the *migration* of an object is proposed as creating a clone of the object at the target site, whereafter the original object is turned into an alias for the clone. Obliq has an only informal semantics, so there is no proof that this style of migration is correct, i.e., transparent to object clients. In this paper, we focus on Øjeblik, an abstraction of Obliq. We give a *π-calculus* semantics to Øjeblik, and we use it to formally prove the correctness of object *surrogation*, an abstraction of object migration.

1 Introduction

The work presented in this paper is in line with the research activity to use the π-calculus as a toolbox for reasoning about distributed object-oriented programming languages. Former works on the semantics of objects as processes have shown the value of this approach: while [22,9,19,10] have focused on just providing formal semantics to object-oriented languages and language features, the work of others [18,20] has been driven by a specific programming problem. Our work tackles a problem in Obliq, Cardelli's lexically-scoped distributed programming language [4]. In this setting, Cardelli proposed to implement object migration by creating a *clone* of the object at the target site and then turning the original (local) object into an *alias* for the new (remote) object. The question arises, whether this style of object migration is correct, and how that can be stated formally. However, Obliq is not equipped with a formal semantics, apart from an unpublished note by Talcott [21], which provides a configuration-style semantics for a subset of Obliq excluding migration. The aim of our project [15] is to remedy this lack of formality and to reason formally about migration.

Previous work Since Obliq is *lexically scoped*, we may ignore the aspects of distribution, at least when regarding the results of Obliq computations, unless distribution sites fail. Following this idea, Øjeblik, which we introduce in Section 3, is an object-based language that represents Obliq's concurrent core [16],

* A draft full paper is available at `http://www.cs.auc.dk/research/FS/ojeblik/`

** Supported by Marie Curie fellowhip, EU-TMR, No. ERBFMBICT983504.

J. van Leeuwen et al. (Eds.): IFIP TCS 2000, LNCS 1872, pp. 390–408, 2000.

but it can also be seen as a concurrent extension of the Imperative Object Calculus [1]. Øjeblik supports *surrogation*, a distribution-free abstraction of migration.

In [16] we gave a formal definition of *correctness* for object surrogation in Øjeblik which can be straightforwardly extended to object migration in Obliq. The intuition is that, in order to be correct, the surrogation (resp. migration) of an object must be transparent to the clients of that object, i.e., the object must behave the same before and after surrogation (resp. migration). We have formalized this concept as the simple equation $\underline{a.\mathsf{ping} \doteq a.\mathsf{surrogate}}$ where the left side represents the object a before surrogation ($a.\mathsf{ping}$ returns a if reachable), the right side represents the object a after surrogation, and $\doteq$ is an appropriate contextual equivalence, based on the possibility of convergence.

In [16] we have also given several proposals of configuration-style semantics for Øjeblik. One of them fits the Obliq implementation [3,4], but does not guarantee the correctness of object surrogation as defined above. This has been formally shown by exhibiting Øjeblik contexts that are able to distinguish the terms $a.\mathsf{ping}$ and $a.\mathsf{surrogate}$. Similar counterexamples apply to object surrogation in Obliq, as we have tested using the Obliq interpreter [3]. In order to achieve the correctness of surrogation, we have proposed an improved semantics in [16], but that work did not contain a proof.

Contribution of the paper In this paper, we present a π-calculus semantics for Øjeblik corresponding to the aforementioned variant proposed in [16]. We also give a notion of contextual equivalence for objects defined in terms of may convergence on π-processes corresponding to the equivalence $\doteq$. More precisely, our semantics uses *Local* π [12], in short Lπ, a variant of the asynchronous π-calculus [7], where the recipients of a channel are local to the process that has created the channel. We prove the correctness of surrogation in two parts.

The *algebraic* part (Theorem 1) relates, with respect to arbitrary π-calculus contexts, the core component of the translation of a single object in its ping'ed and surrogate'd version—both *after commitment* to the respective request under the condition that the request did *not* arise *internally* from the object itself. Here, we use powerful adaptations of proof techniques, partially known from standard π-calculus and Lπ [12], also to exhibit that the alias-component of the surrogate'd version behaves like a forwarder for external requests (Lemma 6). Due to the unavoidable complexity of the language and its semantics, the proof of Theorem 1 is non-trivial, but it provides the seeked insight that we gave the proper π-calculus semantics to aliased objects—which actually drove the development of the proper corresponding operational semantics to aliased objects in [16].

The *iterative* part (Theorem 2, as conjectured in [16]) relates the may-convergence behavior of the terms $a.\mathsf{ping}$ and $a.\mathsf{surrogate}$, within Øjeblik-contexts. Here, we constructively simulate arbitrarily long converging sequences "up to" Theorem 1, so the Øjeblik-contexts must guarantee that the requests will be external. The main difficulty of Theorem 2 is that inherently concurrent Øjeblik-contexts may non-deterministically prevent either term from eventually committing to

the externally requested operation; this also rules out both the must-variant of convergence equivalence as well as bisimulation equivalences.

Summing up, we give (to our knowledge) the first formal proof that migration can be correctly implemented in terms of cloning and aliasing. Due to lack of space, proofs are sketched or omitted; complete proofs are found in the full paper.

2 Local π: An "Object-Oriented" π-Calculus

Local π [12], in short Lπ, is a variant of the asynchronous π-calculus [7] where, similar to the Join-calculus [5], the recipients of a channel are local to the process that has created the channel. This is achieved by imposing the syntactic constraint that only the output capability of names may be transmitted, i.e., the recipient of a name may only use it in output actions. This property makes Lπ particularly suitable for giving the semantics to, and reasoning about, concurrent object-oriented languages. In particular, we can easily guarantee the uniqueness of object identities—a fundamental feature of objects: in object-oriented languages, the name of an object may be transmitted; the recipient may use that name to access the methods of the object, but it cannot create a new object with the same name. When representing objects in the π-calculus, this translates directly into the constraint that the process receiving an object name may only use it in output actions—a guarantee in our setting.

2.1 Terms and Types

In Table 1, we consider a typed version of polyadic Lπ extended with: (i) labeled values ℓ_v, called *variants* [19], with case analysis; (ii) tuple values $\langle\, v_1 .. v_n \,\rangle$, with pattern matching, (iii) constants k, called *keys*, with matching.

To deal with these rather complex values, we introduce several syntactic categories. As additional metavariables, we let s, p, q, r, m, t range over channels, y over variables, w range over values, Q over processes, and i, j, d, h, m over tuple, variant, or other indices. We abbreviate $\ell\_\langle\rangle$ and $\ell\_()$ with ℓ, as well as $\overline{q}\langle\rangle$ and $q().P$ with $\overline{q}$ and $q.P$, respectively, while $\widetilde{v}$ denotes a sequence $v_1 .. v_m$.

Restriction, both inputs, and both destructors are *binders* for the names $x, x_1, \ldots, x_m$ in the respective scopes $P, P_1, \ldots, P_m$. We assume the usual definitions of free and bound occurrences of names, based on these binders; the inductively defined functions $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$ denote those of process P. Similarly, $\mathrm{fc}(P)$ and $\mathrm{bc}(P)$ denote the free and bound channels of process P. Moreover, $\mathrm{n}(P) = \mathrm{fn}(P) \cup \mathrm{bn}(P)$ and $\mathrm{c}(P) = \mathrm{fc}(P) \cup \mathrm{bc}(P)$. *Substitutions*, ranged over by σ, are type-preserving functions from names to values (types are introduced below). For an expression e, $e\sigma$ is the result of applying σ to e, with the usual renaming to avoid captures. We write $e\{{}^{v}/_{x}\}$ for a substitution that operates only on name x, leaving all the other names unchanged. *Relabelings*, ranged over by ρ, are functions from labels to labels. We write $P[{}^{\ell'}/_{\ell}]$ to replace all occurrences of label ℓ by label ℓ' in values occurring in term P. Substitution

Table 1. π-Calculus

Channels: $c \in \mathbf{C}$	*Values*	
Keys: $k \in \mathbf{K}$	$v ::= x$	variable
Names: $\in \mathbf{N}$	$\mid \ell\_v$	variant
$n ::= c \mid k$	$\mid \langle v_1 .. v_n \rangle$	tuple
Auxiliary: $u \in \mathbf{U}$	*Types*	
Variables: $\in \mathbf{X}$	$T ::= \mathbf{C}(T)$	channel type
$x ::= n \mid u$	$\mid \mathbf{K}$	key type
	$\mid [\ell_1{:}T_1 ; \ldots ; \ell_m{:}T_m]$	variant type
Labels $\in \mathbf{L}$	$\mid \langle T_1 .. T_m \rangle$	tuple type
$\ell, \ell_1, \ell_2, \ldots$	$\mid X$	type variable
	$\mid \mu X.\, T$	recursive type

Processes

$P ::= \mathbf{0}$	nil process
$\mid c(x).P$	single *input*
$\mid \overline{c}v$	output
$\mid P_1 \mid P_2$	parallel
$\mid (\boldsymbol{\nu} n{:}T)\, P$	restriction
$\mid \mathbf{!}\, c(x).P$	replicated *input*
$\mid$ if $[k{=}k_1]$ then P_1 elif $[k{=}k_2]$ then P_2 else P_3	key testing
$\mid$ case v of $\ell_1\_(x_1){:}P_1 ; \ldots ; \ell_m\_(x_m){:}P_m$	variant *destructor*
$\mid$ let $(x_1 .. x_m) = v$ in P	tuple *destructor*

The *locality constraint* requires that in (*single* and *replicated*) *inputs* and (*variant* and *tuple*) *destructors* the bound names $x, x_1, \ldots, x_m$ must not be used in free input position within the respective scope $P, P_1, \ldots, P_m$.

and relabeling have the highest operator precedence, parallel composition the lowest.

Types are introduced for essentially three reasons: (i) they allow us to cleanly define some abbreviations, (ii) we use them to give a typed semantics of Øjeblik, and (iii) they allow us to formally prove the main result of the paper using typed behavioral equivalences. Abusing the notation for sets of names and the corresponding types, we use **K** and **C** also as type constructors, where channel types are parameterized over the type of value they carry. For variants and tuples we use standard notations (c.f. [19]). In a recursive type $\mu X.\, T$, occurrences of variable X in type T must be guarded, i.e., underneath a variant or channel constructor. We often omit the type annotation of restriction, when it is clear from the context or not important for the discussion. A *type environment* Γ is a finite mapping from variables to types. A *typing judgment* $\Gamma \vdash P$ asserts that process P is well-typed in Γ, and $\Gamma \vdash v{:}T$ that value v has type T in Γ. There is one typing rule for each process construct; each of them is straightforward. (We

provide the type system in the full version of the paper.) A type environment Γ is *closed* if it contains only names, no auxiliary variables.

2.2 Semantics and Proof Techniques

We equip our π-calculus with a standard *reduction semantics*. Its rules, defining the *reduction relation* $\rightarrow$, are precisely those of [13], but based on a notion of structural equivalence that is extended to deal with if-, case-, and let-constructs. For simplicity, we only consider the semantics of well-typed processes.

Definition 1. Structural equivalence, *written* $\equiv$, *is the smallest relation preserved by parallel composition and restriction, which satisfies the axioms below:*

- $P \equiv Q$, *if* P *is* α*-convertible to* Q;
- $P \mid \mathbf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$;
- $(\boldsymbol{\nu} a)\,\mathbf{0} \equiv \mathbf{0}$, $(\boldsymbol{\nu} a)\,(\boldsymbol{\nu} b)\,P \equiv (\boldsymbol{\nu} b)\,(\boldsymbol{\nu} a)\,P$,
- $(\boldsymbol{\nu} a)\,(P \mid Q) \equiv P \mid (\boldsymbol{\nu} a)\,Q$, *if* $a \notin \mathrm{fn}(P)$;
- $\mathsf{if}\ [k_1{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \equiv P_1$;
- $\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k_2{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \equiv P_2$, *if* $k_1{\neq}k$;
- $\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \equiv P_3$, *if* $k_1{\neq}k{\neq}k_2$;
- $\mathsf{case}\ \ell_j\_v_j\ \mathsf{of}\ \ell_1\_(x_1){:}P_1\,;\ldots;\ell_j\_(x_j){:}P_j\,;\ldots;\ell_m\_(x_m){:}P_m \equiv P_j\{{}^{v_j}/_{x_j}\}$;
- $\mathsf{let}\ (\,x_1\,..\,x_m\,) = \langle\,v_1..\,v_m\,\rangle\ \mathsf{in}\ P \equiv P\{{}^{\widetilde{v}}/_{\widetilde{x}}\}$.

The relation $\Rightarrow$ is the reflexive-transitive closure of $\rightarrow$. For any relation $\mathcal{R}$ on processes, $\rightarrow_{\mathcal{R}}$ denotes $\rightarrow\mathcal{R}$, and $\Rightarrow_{\mathcal{R}}$ the reflexive-transitive closure of $\rightarrow_{\mathcal{R}}$.

As regards behavioral equivalences, we focus on *barbed bisimulation* [14], a uniform mechanism for defining behavioral equivalences in any process calculus possessing (i) a *reduction relation* and (ii) an *observability predicate*. Barbed bisimulation equips a global observer with a minimal ability to observe actions and/or process states but it is not a congruence. By closing barbed bisimulation under contexts we obtain a much finer relation. A context $C[\cdot]$ is a process with exactly one hole, written $[\cdot]$, where a process may be plugged in. A context $C[\cdot]$ is *static* if it is structurally equivalent to $(\boldsymbol{\nu}\widetilde{a})\,(P \mid [\cdot])$, for some P and $\widetilde{a}$. We let $E[\cdot]$ range over static contexts. In an *asynchronous* scenario, only output actions are usually considered observable [2]. A process P has a *barb* at channel c, written $P{\downarrow_c}$, if $P \equiv E[\,\bar{c}v\,]$ for some static context $E[\cdot]$, value v, and channel $c \in \mathrm{fn}(P)$. We write $P{\Downarrow_c}$, if there exists a process P' with $P \Rightarrow P'$ and $P'{\downarrow_c}$. In a *typed* scenario, only well-typed context are usually considered. Therefore, we recall the notion of (Δ/Γ)*-context* [17,19]: when filled with a process P with $\Gamma \vdash P$, a (Δ/Γ)-context $C[\cdot]$ guarantees the typing $\Delta \vdash C[P]$.

We often work with channels that have been extruded to the environment. To keep track of the fact that they cannot be used in input by the environment we generalize the standard typed barbed relations.

Definition 2 (Barbed Relations). *Let* $\mathcal{C} \subseteq \mathbf{C}$. Barbed $\mathcal{C}$-bisimilarity, *written* $\dot{\cong}_{\mathcal{C}}$, *is the largest symmetric relation on processes, such that* $P \,\dot{\cong}_{\mathcal{C}}\, Q$ *implies:*

1. *If* $P \xrightarrow{\tau} P'$, *then there exists* Q' *such that* $Q \Rightarrow Q'$ *and* $P' \,\dot{\cong}_{\mathcal{C}}\, Q'$

2. If $P{\downarrow_c}$, with $c \notin \mathcal{C}$, then $Q{\Downarrow_c}$.

Let Γ be a typing, and P and Q two processes such that $\Gamma \vdash P, Q$. We say that P and Q are barbed $\Gamma;\mathcal{C}$-equivalent*, written $P \simeq_{\Gamma;\mathcal{C}} Q$, if, for each closed type environment Δ and static (Δ/Γ)-context $C[\cdot]$ not containing names in $\mathcal{C}$ in input position, we have $C[P] \mathrel{\dot{\cong}}_{\mathcal{C}} C[Q]$. We say that P and Q are* barbed $\Gamma;\mathcal{C}$-congruent*, written $P \cong_{\Gamma;\mathcal{C}} Q$, if, for each closed type environment Δ and (Δ/Γ)-context $C[\cdot]$ not containing names in $\mathcal{C}$ in input position, we have $C[P] \mathrel{\dot{\cong}}_{\mathcal{C}} C[Q]$.*

If $\mathcal{C}{=}\emptyset$ in Definition 2, we omit $\mathcal{C}$ and get the standard definitions of barbed bisimilarity $\mathrel{\dot{\cong}}$, barbed Γ-equivalence $\simeq_\Gamma$, and barbed Γ-congruence $\cong_\Gamma$, respectively. If $\mathcal{C}{=}\{s\}$, we write $\cong_{\Gamma;s}$ for $\cong_{\Gamma;\mathcal{C}}$ and $\simeq_{\Gamma;s}$ for $\simeq_{\Gamma;\mathcal{C}}$. Due to the restrictions on the contexts, it holds that $\overline{s}v \cong_{\Gamma;s} \mathbf{0}$ and, by asynchrony, $s(u).\mathbf{0} \cong_{\Gamma;s} \mathbf{0}$.

The main inconvenience of barbed equivalences and congruences is that they use quantifications over contexts in the definition, and this unnecessarily complicates proofs of process equality. In the long version of this paper, we provide and make use of labeled characterizations. Parts of our proofs are based on the generalization (to our setting) of Lπ proof techniques [12] that are based on special processes called *link*. A link (sometimes called a *forwarder* [8]), is a process $!\,p(u).\overline{q}u$, abbreviated $p \triangleright q$, that behaves as an unbounded unordered buffer receiving values at one end (p) and retransmitting them at the other end (q). The following lemma allows us to always output bound names instead of free names. Its proof, in pure Lπ, can be found in [11], but its generalization to our typed setting is straightforward.

Lemma 1. *Let $\Gamma \vdash \overline{a}v$, for some Γ. Let $b \in \mathrm{fc}(v)$ with $\Gamma \vdash b{:}\mathbf{C}(T)$. Let $d \notin \mathrm{c}(v)$ and $w = v\{{}^{d}/_{b}\}$. Then $\overline{a}v \cong_\Gamma (\boldsymbol{\nu} d{:}\mathbf{C}(T))\,(\overline{a}w \mid d \triangleright b)$.*

3 Øjeblik: A Concurrent Object Calculus

The set $\mathcal{L}$ of untyped Øjeblik expressions is generated, as shown in Table 2, where l ranges over method *labels*. In this section, we present the Øjeblik's call-by-value semantics, first informally, then formalized using the π-calculus of § 2, through which also a standard behavioral semantics is defined.

Objects. An object $[\mathrm{l}_j{=}m_j]_{j\in J}$ consists of a finite collection of updatable named methods $\mathrm{l}_j{=}m_j$, for pairwise distinct labels l_j. In a method $\varsigma(s,\tilde{x})b$, the letter ς denotes a binder for the self variable s and argument variables $\tilde{x}$ within the body b. Moreover, every object in Øjeblik comes equipped with special methods for cloning, aliasing, surrogation, and pinging, which cannot be updated.

Method invocation $a.\mathrm{l}\langle\, a_1 \,..\, a_n \,\rangle$ with field l of the object a containing the method $\varsigma(s,\tilde{x})b$ results in the body b with the self variable s replaced by the enclosing object a, and the formal parameters $\tilde{x}$ replaced by the actual parameters $a_1 \,..\, a_n$ of the invocation. Method update $a.\mathrm{l}{\Leftarrow}m$ overwrites the current content of the named field l in a with method m and evaluates to the modified object. The operation $a.\mathsf{clone}$ creates an object with the same fields as the original object and initializes the fields to the same entries as in the original object.

a, b	$::= \mathbb{O}$	object
	$\mid a.\mathsf{l}\langle\, a_1 \,..\, a_n \,\rangle$	method invocation
	$\mid a.\mathsf{l} \Leftarrow m$	method update
	$\mid a.\mathsf{clone}$	shallow copy
	$\mid a.\mathsf{alias}\langle b \rangle$	object aliasing
	$\mid a.\mathsf{surrogate}$	object surrogation
	$\mid a.\mathsf{ping}$	object ping
	$\mid s, x, y, z$	variables
	$\mid \mathsf{let}\, x = a\, \mathsf{in}\, b$	local definition
	$\mid \mathsf{fork}\langle a \rangle$	thread creation
	$\mid \mathsf{join}\langle a \rangle$	thread destruction
$\mathbb{O}$	$::= [\mathsf{l}_j = m_j]_{j \in J}$	object record
m_j	$::= \varsigma(s_j, \tilde{x}_j) b_j$	method

Table 2. Øjeblik Syntax

The operation $a.\mathsf{alias}\langle b \rangle$ replaces the object a with an alias for b, regardless of whether a is already an alias or still an object record; if b itself is an alias to, e.g., an object c, then we consequently create, by transitivity, an *alias chain*.

The operation $a.\mathsf{surrogate}$ turns object a into a local proxy for a remote copy of itself, as implemented by the uniform method $\mathsf{surrogate} = \varsigma(s)s.\mathsf{alias}\langle s.\mathsf{clone} \rangle$. The operation $a.\mathsf{ping}$ is implemented by another uniform method $\mathsf{ping} = \varsigma(s)s$, such that it returns of the object o that results from the evaluation of a its "current identity", i.e., due to possible forwarding the current endpoint of an alias chain potentially starting at object o.

Scoping. An expression $\mathsf{let}\, x = a\, \mathsf{in}\, b$ (non-recursive) first evaluates a, binding the result to x, and then evaluates b within the scope of x.

Concurrency. Computational activity takes place within so-called *threads*. In addition to the main thread, new threads can be created by the fork command. The result of a forked computation is grabbed by the join command.

Self-Infliction, Serialization, Protection. The *current self* of a thread is the self of the last method invoked in the thread that has not yet completed. An Øjeblik operation is *self-inflicted* (also: *internal*) if it operates on the current self; otherwise, it is *external*. Øjeblik objects are *serialized*: within any object, at any time, at most one thread may be active. Serialization is implemented by associating with each object a *mutex* that is locked when a thread enters the object and released when the thread exits the object. More precisely, the mutex is always acquired for external operations, but never for internal ones; this concept allows a method to recursively call its siblings through self, but it excludes mutual recursion between objects. Øjeblik objects are *protected*: the *critical* operations update, aliasing, and cloning, are only allowed if they are internal. An operation is called *valid* if it is internal or not critical.

$$
\begin{array}{|rcl|}
\hline
[\![\, a.\mathsf{clone} \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,\Big(\; [\![\, a \,]\!]^k_q \,\big|\, q(y,k').\,\overline{y}\langle \mathsf{cln}\_p, k'\rangle \;\Big) \\
[\![\, a.\mathsf{alias}\langle b\rangle \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q_x q_y)\,\Big(\; [\![\, a \,]\!]^k_{q_y} \,\big|\, q_y(y,k_y).([\![\, b \,]\!]^{k_y}_{q_x} \;\mid\; q_x(x,k_x)\,.\,\overline{y}\langle \mathsf{ali}\_\langle x,p\rangle, k_x\rangle)\;\Big) \\
[\![\, a.\mathrm{l}_j \Leftarrow\!\varsigma(s,\widetilde{x})b \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,\Big([\![\, a \,]\!]^k_q \,\big|\, q(y,k').(\boldsymbol{\nu} t)(\;!\,t(s,\widetilde{x},r,k).[\![\, b \,]\!]^k_r \,|\, \overline{y}\langle \mathsf{upd}_j\_\langle t,p\rangle, k'\rangle)\Big) \\
[\![\, a.\mathrm{l}_j\langle\, a_1 \,..\, a_n \,\rangle \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q q_1 \cdots q_n)\,\Big(\; [\![\, a \,]\!]^k_q \,\big|\, q(y,k_0).([\![\, a_1 \,]\!]^{k_0}_{q_1} \mid q_1(x_1,k_1).([\![\, a_2 \,]\!]^{k_1}_{q_2} \mid \cdots \\
 & & \qquad\qquad q_n(x_n,k_n).\overline{y}\langle \mathsf{inv}_j\_\langle x_1 \,..\, x_n, p\rangle, k_n\rangle \cdots))\Big) \\
[\![\, a.\mathsf{surrogate} \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,(\; [\![\, a \,]\!]^k_q \;\mid\; q(y,k')\,.\,\overline{y}\langle \mathsf{sur}\_p, k'\rangle\;) \\
[\![\, a.\mathsf{ping} \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,(\; [\![\, a \,]\!]^k_q \;\mid\; q(y,k')\,.\,\overline{y}\langle \mathsf{png}\_p, k'\rangle\;) \\
\hline
[\![\, \mathsf{let}\, x = a\, \mathsf{in}\, b \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,(\; [\![\, a \,]\!]^k_q \;\mid\; q(x,k')\,.\,[\![\, b \,]\!]^{k'}_p\;) \\
[\![\, x \,]\!]^k_p & \stackrel{\text{def}}{=} & \overline{p}\langle x,k\rangle \\
\hline
[\![\, \mathsf{fork}(a) \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q t)\,(\; [\![\, a \,]\!]^{\nu}_q \;\mid\; \overline{p}\langle t,k\rangle \;\mid\; q(x,k').t(r,k'').\overline{r}\langle x,k''\rangle\;) \\
[\![\, \mathsf{join}(b) \,]\!]^k_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,(\; [\![\, b \,]\!]^k_q \;\mid\; q(t,k')\,.\,\overline{t}\langle p,k'\rangle\;) \\
\hline
\end{array}
$$

Table 3. Øjeblik Semantics — Clients, Scoping, Concurrency

3.1 Translational Semantics

In addition to the core π-calculus of Section 2, we use *parameterized recursive definitions*, which can be faithfully represented in terms of replication [13].

The semantics as presented in Tables 3 and 4, maps Øjeblik-terms into π-calculus terms parameterized on two names: in a term $[\![\, a \,]\!]^k_p$, the channel p is used to return the term's result, while the key k represents the term's current self, which is required to deal with self-infliction. The essence of the semantics is to set up processes representing objects that serve clients' requests. Different requests for operating on objects are distinguished by corresponding π-calculus labels. We explain the semantics by showing how requests are generated by clients, and then how they are served by objects. We omit explanations for the semantics of scoping and concurrency; they can be found in the full paper.

Clients. In Table 3, the current self k of encoded terms is 'used' as the current self of the evaluation of the first subterm in left-to-right evaluation order. All the translations in Table 3 follow a common scheme. For example, in the translation of a method invocation $[\![\, a.\mathrm{l}_j\langle\, a_1 \,..\, a_n \,\rangle \,]\!]^k_p$, the subterms $a, a_1 \,..\, a_n$ have to be evaluated one after the other: the individual evaluations use private return channels $q, q_1 \,..\, q_n$, which are subsequently asked for the respective results $y, x_1 \,..\, x_n$, but also for the respective new current self $i, i_1 \,..\, i_n$ to be used by the next evaluation—this can be same as for the previous evaluation, but is not necessarily so (c.f. the description of object managers below). After the last subterm a_n has returned its result, the accumulated information is used to send a suitable request with label inv_j to self-channel y of object a, also carrying the overall result channel p and the latest current self i_n. Thus, the responsibility to signal a result on p is passed on to the respective object manager waiting at y.

$$[\![\,\mathbb{O}\,]\!]^k_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} s\widetilde{t})\Big(\,\overline{p}\langle s,k\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s,\widetilde{t}\,\rangle \mid \prod_{j\in J} !\, t_j(s_j,\widetilde{x}_j,r,k').[\![\, b_j\,]\!]^{k'}_r\,\Big)$$

$$\mathsf{newO}_{\mathbb{O}}\langle\, s,\widetilde{t}\,\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu} m_{\mathrm{e}} m_{\mathrm{i}} k_{\mathrm{e}} k_{\mathrm{i}})\Big(\,\overline{m_{\mathrm{e}}} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,m_{\mathrm{e}},m_{\mathrm{i}},k_{\mathrm{e}},k_{\mathrm{i}},\widetilde{t}\,\rangle\,\Big)$$

$$\mathsf{newA}_{\mathbb{O}}\langle\, s,s_{\mathrm{a}}\,\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu} m_{\mathrm{e}} m_{\mathrm{i}} k_{\mathrm{e}} k_{\mathrm{i}})\Big(\,\overline{m_{\mathrm{e}}} \mid \mathsf{AM}_{\mathbb{O}}\langle\, s,m_{\mathrm{e}},m_{\mathrm{i}},k_{\mathrm{e}},k_{\mathrm{i}},s_{\mathrm{a}}\,\rangle\,\Big)$$

$$\begin{array}{l}
\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},\widetilde{t}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\quad \text{if } [k{=}k_{\mathrm{i}}] \text{ then}\\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}_(r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid (\boldsymbol{\nu} s^*)\Big(\,\overline{r}\langle s^*,k^*\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*,\widetilde{t}\,\rangle\,\Big)\ ;\\
\qquad\quad \mathsf{ali}_(s_{\mathrm{a}},r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{r}\langle s_{\mathrm{a}},k^*\rangle\ ;\\
\qquad\quad \mathsf{upd}_j_(t',r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,t_1\,..\,t_{j-1},t',t_{j+1}\,..\,t_n\,\rangle \mid \overline{r}\langle s,k^*\rangle\ ;\\
\qquad\quad \mathsf{inv}_j_(\widetilde{x},r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid \overline{t_j}\langle s,\widetilde{x},r,k^*\rangle\ ;\\
\qquad\quad \mathsf{sur}_(r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid [\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]^{k^*}_r\ ;\\
\qquad\quad \mathsf{png}_(r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid [\![\, s\,]\!]^{k^*}_r\\
\quad \text{elif } [k{=}k_{\mathrm{e}}] \text{ then}\\
\qquad \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \,\big|\, \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}_(r) : m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ;\\
\qquad\qquad\qquad \mathsf{ali}_(s_{\mathrm{a}},r) : m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ;\\
\qquad\qquad\qquad \mathsf{upd}_j_(t',r) : m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ;\\
\qquad\qquad\qquad \mathsf{inv}_j_(\widetilde{x},r) : \mathsf{CM}[\,\overline{t_j}\langle s,\widetilde{x},r^*,k^*\rangle\,]\ ;\\
\qquad\qquad\qquad \mathsf{sur}_(r) : \mathsf{CM}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]^{k^*}_{r^*}\,]\ ;\\
\qquad\qquad\qquad \mathsf{png}_(r) : \mathsf{CM}[\,[\![\, s\,]\!]^{k^*}_{r^*}\,]\\
\quad \text{else } \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},\widetilde{t}\,\rangle \,\big|\, m_{\mathrm{e}}.\big(\,\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k\,\big)\Big)
\end{array}$$

$$\mathsf{CM}[\cdot] \stackrel{\text{def}}{=} (\boldsymbol{\nu} r^*)\big(\,[\cdot] \mid r^*(y,k').m_{\mathrm{i}}(k'').\big(\,\overline{r}\langle y,k''\rangle \mid \overline{m_{\mathrm{e}}}\,\big)\big)$$

$$\begin{array}{l}
\mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},s_{\mathrm{a}}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\quad \text{if } [k{=}k_{\mathrm{i}}] \text{ then}\\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}_(r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid (\boldsymbol{\nu} s^*)\Big(\,\overline{r}\langle s^*,k^*\rangle \mid \mathsf{newA}_{\mathbb{O}}\langle\, s^*,s_{\mathrm{a}}\,\rangle\,\Big)\ ;\\
\qquad\quad \mathsf{ali}_(s'_{\mathrm{a}},r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s'_{\mathrm{a}}\,\rangle \mid \overline{r}\langle s'_{\mathrm{a}},k^*\rangle\ ;\\
\qquad\quad \mathsf{upd}_j_(t',r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l,k\rangle\ ;\\
\qquad\quad \mathsf{inv}_j_(\widetilde{x},r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l,k\rangle\ ;\\
\qquad\quad \mathsf{sur}_(r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l,k\rangle\ ;\\
\qquad\quad \mathsf{png}_(r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l,k\rangle\\
\quad \text{elif } [k{=}k_{\mathrm{e}}] \text{ then } \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid m_{\mathrm{i}}(k).\big(\,\overline{s_{\mathrm{a}}}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}\,\big)\\
\quad \text{else} \qquad\qquad \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},s_{\mathrm{a}}\,\rangle \mid m_{\mathrm{e}}.\big(\,\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k\,\big)\Big)
\end{array}$$

Table 4. Øjeblik Semantics — Objects

Objects. The semantics $[\![\, \mathbb{O} \,]\!]^k_p$ of an object $\mathbb{O} := [\mathrm{l}_j{=}\varsigma(s_j, \tilde{x}_j)b_j]_{j\in J}$, as shown in Table 4 (again similar to [10]), consists of a message that returns the object's reference s together with the current self k on channel p, a composition of replicated processes that give access to the method bodies $[\![\, b_j \,]\!]^{k'}_r$, and a new object process $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t} \,\rangle$ that connects invocations on s from the outside to the method bodies, which are invoked by the trigger names $\widetilde{t}$. Inside $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t} \,\rangle$, several private names are needed: *mutexes* $\widetilde{m} := m_{\mathrm{e}}, m_{\mathrm{i}}$ are used for serialization; the (*internal*) key k_{i} is used to detect self-infliction; the (*external*) key k_{e} is used to implement serialization in a concurrent environment (see later on). The behavior of objects and aliases is represented by the object manager OM and alias manager AM, respectively, which both, for each request arriving along their reference s, first check for self-infliction $[k{=}k_{\mathrm{i}}]$, and then, by simple case analysis, for the kind of the request. We first explain how internal requests are served in objects and aliases. External requests will be served later.

Serving Internal Requests $[k{=}k_{\mathrm{i}}]$ No serialization or protection is required.

Object Managers (OM). For each field, the manager may activate appropriate instances of method bodies (case inv_j: the method body bound to l_j along trigger name t_j) and administer updates (case upd_j: install a new trigger name t'). Cloning (case cln) restarts the current object manager in parallel with a new object, which uses the same method bodies $\widetilde{t}$, but is accessible through a fresh reference s^*. Aliasing (case ali) starts an appropriate alias manager AM instead of re-starting the previous object manager OM. Surrogation and ping (cases sur and png) are modeled according to their uniform method definitions.

Alias Managers (AM). Local requests for cloning and aliasing are allowed and behave analogous to the respective clauses in object managers, but restarting AM instead of OM. Update, invocation, surrogation, and ping requests are immediately forwarded *as is* to the aliasing target s_{a}.

Nonces (k^*). To guarantee the receptiveness of objects, managers OM and AM always have to be restarted with some possibly changed state. However, serialization requires that at any moment, only one request shall be *active* in an object. According to our semantics, program contexts will never give rise to several competing self-inflicted requests, but, when reasoning within arbitrary π-calculus contexts, as we do in § 4.2, their existence must be taken into account. Therefore, we add another layer of protection to increase the robustness of serialization: each time a request enters a manager, a fresh key k^* is created to be used in the restarted manager; this key must subsequently be used as the current self for all activities enabled by the current request. Thus, the consumption of one of several competing pending requests renders its competitors external. Note that nonces would not be necessary if we were interested in correct behaviors within only translated contexts.

Serving External Requests $[k{\neq}k_{\mathrm{i}}]$ Serialization and protection are required.

In order to clarify the behavior of an object manager serving an external request, Figure 1 show the five relevant "states", starting from a *free* manager

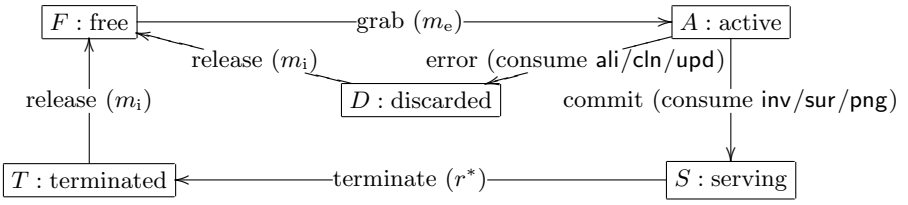

Fig. 1. Object Manager Serving External Requests

that becomes *active* by some pending request grabbing its serialization-lock. Then, if the request is protection-critical it is *discarded*, otherwise the manager *commits* to it and *serves* it until explicit *termination*. In both cases, the manager becomes free again by releasing the lock. Note that internal requests can only be served in "state" S of a manager that currently already serves some request. In Subsection 3.3, we explain Figure 1 in more detail; for now, it just offers an informal device to guide the explanations of the semantics of object managers.

Serialization ($m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}$). As mentioned earlier, mutual exclusion within an object is implemented by mutexes, so, upon creation of a new object newO, the fresh mutex channel m_e is initialized. According to serialization, the intended continuation behavior of an incoming external requests is blocked on m_e, once it enters a manager. The manager itself is immediately restarted and remains receptive; it also remains in its "state" according to Figure 1. Arbitrarily many requests can be blocked this way and compete for the mutex m_e once it becomes available. A successfully unblocked request is resent to the same manager, but now carrying another key k_e, which allows the manager to detect that the request has grabbed the mutex, so the manager can evolve into "state" A. We call *pre-processing* the procedure of intermediate blocking of requests and their subsequent reemission with key k_e instead. Alongside with the pre-processed request, its former current self k is stored on the (internal) mutex m_i for recovery after termination. This recovery is actually necessary since the original current self k is possibly required for use later on by the sender of the request.

Nonces (k^*). Pre-processing must not reinitialize the key k_i of the restarted manager: a currently self-inflicted operation interleaved by pre-processing might be hindered to proceed, because it could unintendedly become external.

Object Managers (OM). Cloning, aliasing, or update, are critical operations. Once a respective pre-processed request is consumed, the manager evolves from "state" A into "state" D: the request and its former current self k, stored on channel m_i, will be simply ignored when releasing $\overline{m_\mathrm{e}}$ by consuming $\overline{m_\mathrm{i}}k$.

Invocation, surrogation, and ping, are non-critical operations. Once a respective pre-processed request is consumed, the manager evolves from "state" A into "state" S implying that no other external request shall be served (apart from pre-processing) until the current one has terminated. In order to be notified of that event, we employ a *call manager* protocol, represented by the context $\mathsf{CM}[\cdot]$: instead of delegating to some other process the responsibility of returning a result

$$
\begin{array}{rll}
A, B & ::= [\mathrm{l}_j{:}\widetilde{B}_j{\to}\widehat{B}_j]_{j\in J} & \text{object record type} \\
 & \mid \mathrm{Thr}(A) & \text{thread type}
\end{array}
$$

$$
\begin{array}{rl}
\mathbf{R}(X) & \stackrel{\text{def}}{=} \mathbf{C}(X, \mathbf{K}) \\
\mathbf{M}(B_1 \,..\, B_n{\to}\widehat{B}) & \stackrel{\text{def}}{=} [\![\, B_j \,]\!] \,..\, [\![\, B_n \,]\!], \mathbf{R}([\![\, \widehat{B} \,]\!]) \\
[\![\, [\mathrm{l}_j{:}\widetilde{B}_j{\to}\widehat{B}_j]_{j\in J} \,]\!] & \stackrel{\text{def}}{=} \\
\multicolumn{2}{r}{\mu X.\, \mathbf{C}(\langle \left[\begin{array}{ll}
\mathsf{cln} : & \mathbf{R}(X) \\
\mathsf{ali} : & \langle\, X, \mathbf{R}(X) \,\rangle \\
\mathsf{upd}_j : & \langle\, \mathbf{C}(\langle\, X, \mathbf{M}(\widetilde{B}_j{\to}\widehat{B}_j), \mathbf{K} \,\rangle, \mathbf{R}(X)) \,\rangle \\
\mathsf{inv}_j : & \langle\, \mathbf{M}(\widetilde{B}_j{\to}\widehat{B}_j) \,\rangle \\
\mathsf{sur} : & \mathbf{R}(X) \\
\mathsf{png} : & \mathbf{R}(X)
\end{array}\right]_{j\in J}, \mathbf{K} \,\rangle)} \\
[\![\, \mathrm{Thr}(A) \,]\!] & \stackrel{\text{def}}{=} \mathbf{C}(\langle\, \mathbf{R}(A), \mathbf{K} \,\rangle) \\
[\![\, \Gamma, x{:}A \,]\!] & \stackrel{\text{def}}{=} [\![\, \Gamma \,]\!], x : [\![\, A \,]\!]
\end{array}
$$

Table 5. Translation of Øjeblik-types

on r, a fresh return channel r^* is created to be used within $[\cdot]$ in place of r, such that the result will first appear on r^*. Until this event, other external requests remain blocked, while internal request may well be served. After this event, the manager evolves from "state" S into "state" T, where the former current self can be grabbed from m_i, the result y be forwarded to the intended result channel r (along with the former current self), and the mutex m_e be released. Note that externally triggered method bodies $[\![\, b_j \,]\!]$, and also surrogation and ping bodies $[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]$ and $[\![\, s \,]\!]$, are all run in the context of the nonce k^*, which is now the internal key of the OM, so their further calls to s will be self-inflicted. This is essential for surrogation, since cloning and aliasing are only allowed internally.

Alias Managers (AM). According to our discussion in [16], external requests that arrive at an active alias manager should be blocked until the current activity finishes and the lock m_e is released. Once this happens, all external requests are—after three intermediate steps via channels m_e, s, and m_i—forwarded to the aliasing target s_a. The pre-processing of requests, presumably superfluous in alias managers, is necessary also there, because there may be pending pre-processed requests that have come to existence when s was connected to an OM.

Type Translation Øjeblik is equipped with a standard static type system [16]. The translation of terms into our π-calculus is accompanied with a straightforward translation of the corresponding types in Table 5. Its importance is that (i) the two translations together preserve typings (see the full paper for details), and (ii) that we can exploit the type information in proofs of properties of Øjeblik-terms, not least by applying typed barbed relations.

3.2 Behavioral Semantics

A standard way to define *program equivalence* is to compare the behavior of programs within arbitrary program contexts, as, for example, shown in previous work on the Imperative Object Calculus (IOC) [1,6]. In our setting, an *Øjeblik context* $C[\cdot]$ has a single hole $[\cdot]$ that may be filled with an Øjeblik term. In the remainder of the paper, we assume that Øjeblik-contexts always yield well-typed terms when plugging some Øjeblik-term into the hole. As a simple notion of program behavior to be tested based on our Øjeblik-semantics, we choose the existence of barbs as in § 2. This closely follows the intuition that a term $[\![\, a \,]\!]_p^k$ should tell its result on name p as soon as it knows it. So, an Øjeblik term *converges*, if its semantics *may* report its result on the name p.

Definition 3 (Convergence). *If $a \in \mathcal{L}$ is an Øjeblik term, then $a{\Downarrow}$ if $[\![\, a \,]\!]_p^k{\Downarrow_p}$.*

Definition 4 (Behavioral equivalence). *Two Øjeblik terms $a, b \in \mathcal{L}$ are* behaviorally equivalent*, written $a \doteq b$, if $C[a] \Downarrow$ iff $C[b] \Downarrow$ for all contexts $C[\cdot]$.*

Note that this equivalence is based on a *may*-variant of convergence. With respect to our goal of reasoning about surrogation, *must*-variants of convergence would be too strong, because, in a concurrent language with fork, threads may nondeterministically affect the outcome and convergence of evaluation.

3.3 Properties of the Translational Semantics

An important advantage of using a typed Lπ semantics is the easy provability that object managers are the unique receptors for their (bound) self-channels.

Lemma 2 (Uniqueness of Objects). *Let a be an Øjeblik term. If $[\![\, a \,]\!]_p^k \Rightarrow Z$ with $Z \equiv (\boldsymbol{\nu}\widetilde{z})\,(\, M \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \ldots \,\rangle\,)$ or $Z \equiv (\boldsymbol{\nu}\widetilde{z})\,(\, M \mid \mathsf{AM}_{\mathbb{O}}\langle\, s, \ldots \,\rangle\,)$, then $s \in \widetilde{z}$ and s does not appear free in input position within M.*

We now analyze, referring to Figure 1, how the shape of a particular object manager and its surrounding context evolves during computation. We will need a particular case thereof (Lemma 3) later on in the proof of Theorem 2.

Observation 1: Pre-processing does not change the "state" of object managers. At any time, an object/alias manager is ready to receive a request $\overline{s}\langle l, k\rangle$ with $k_{\mathrm{e}} \neq k \neq k_{\mathrm{i}}$. The manager is identically restarted afterwards, but will have spawned a process $m_{\mathrm{e}}.(\, \overline{s}\langle l, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k \,)$ that replaces the consumed request. Let us assume requests $\overline{s}v_j$ with $v_j := \langle\, l_j, k_j \,\rangle$ for $1 \leq j \leq h$ (and $\widetilde{v} := v_1 \ldots v_h$) are pre-processed by the object manager $\mathsf{OM}_{\mathbb{O}}\langle\, s, m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle$, so $k_{\mathrm{e}} \neq k_j \neq k_{\mathrm{i}}$ for all $1 \leq j \leq h$. Then:

$$\mathsf{PP}_{\mathbb{O}}\langle\, s, m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}, \widetilde{v}\,\rangle \stackrel{\text{def}}{=} \prod_{1 \leq j \leq h} m_{\mathrm{e}}.(\, \overline{s}\langle l_j, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_j \,)$$

Observation 2: In object managers, k_i may be extruded, $m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}$ may not. Assume that an inv_j-request along s appears at $\mathsf{OM}_\mathbb{O}\langle\, s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k_\mathrm{i}, \widetilde{t}\,\rangle$, is pre-processed, gets the mutex m_e and re-enters along s with k_e. During this, according to the semantics, a fresh internal key k^* is created and extruded to the corresponding method body. The names $\widetilde{n} := m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}$ are never extruded; they constitute the proper boundary of a manager during computation. This observation also provides the formal basis for Figure 1: a term Z containing an object manager $\mathsf{OM}_\mathbb{O}\langle\, s, \widetilde{n}, k_\mathrm{i}, \widetilde{t}\,\rangle$ corresponds to "state" F, A, D, S, or T, respectively (for this manager), if after moving—as far as possible—all top-level parallel components of Z outside the scope of $(\boldsymbol{\nu}\widetilde{n})$ the remaining component of Z inside the scope has a characteristic shape. In the full paper, we show the complete analysis; here, we outline two special cases: *free* object managers in "state" F, and *committing* object managers ready to evolve from "state" A into "state" S.

Observation 3: An object manager is free*, if its external mutex is available.* In our semantics, a manager is willing to grant access, if the external mutex m_e occurs unguarded in the term that describes the current "state", so the general shape of a *free object* (and analogously *alias*) *manager* is:

$$\begin{aligned}\mathsf{freeO}_\mathbb{O}\langle\, s, k_\mathrm{i}, \widetilde{t}, \widetilde{v}\,\rangle &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{n})\big(\,\overline{m_\mathrm{e}} \;\big|\; \mathsf{OM}_\mathbb{O}\langle\, s, \widetilde{n}, k_\mathrm{i}, \widetilde{t}\,\rangle \;\big|\; \mathsf{PP}_\mathbb{O}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\,\big)\\ \mathsf{freeA}_\mathbb{O}\langle\, s, k_\mathrm{i}, s_\mathrm{a}, \widetilde{v}\,\rangle &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{n})\big(\,\overline{m_\mathrm{e}} \;\big|\; \mathsf{AM}_\mathbb{O}\langle\, s, \widetilde{n}, k_\mathrm{i}, s_\mathrm{a}\,\rangle \;\big|\; \mathsf{PP}_\mathbb{O}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\,\big)\end{aligned}$$

where the keys mentioned in $\widetilde{v}$ of $\mathsf{PP}_\mathbb{O}\langle\ldots\rangle$ neither match k_e nor k_i. Note that $\mathsf{newO}_\mathbb{O}\langle\, s, \widetilde{t}\,\rangle \equiv (\boldsymbol{\nu}k_\mathrm{i})\,\mathsf{freeO}_\mathbb{O}\langle\, s, k_\mathrm{i}, \widetilde{t}, \emptyset\,\rangle$, and analogously for $\mathsf{newA}_\mathbb{O}\langle\ldots\rangle$.

Observation 4: An object manager is ready to commit*, if it may consume a valid pre-processed request.* The following lemma derives from the ability to commit to a valid external request—visible as the availability of a valid pre-processed request, i.e., a request carrying k_e—the shape of the object manager before and after commitment, including all of its current pre-processed requests.

Lemma 3 (Committing Object Manager). *Let $a \in \mathcal{L}$ and $[\![\, a \,]\!]_p^k \;\Rightarrow Z$. If $Z \equiv E[\,\overline{s}\langle l, k_\mathrm{e}\rangle \mid \mathsf{OM}_\mathbb{O}\langle\, s, \widetilde{n}, k_\mathrm{i}, \widetilde{t}\,\rangle\,]$ with $l \in \{\mathsf{inv}_j\_\langle\widetilde{x}, r\rangle, \mathsf{png}\_r, \mathsf{sur}\_r\}$, and $\widetilde{n} = m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}$, then $Z \;\rightarrow Z'$ for*

$$\begin{aligned}Z &\equiv \widehat{E}[\,(\boldsymbol{\nu}\widetilde{n})\big(\,\overline{m_\mathrm{i}}k' \mid \mathsf{PP}_\mathbb{O}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \mid \mathsf{OM}_\mathbb{O}\langle\, s, \widetilde{n}, k_\mathrm{i}, \widetilde{t}\,\rangle \mid \overline{s}\langle l, k_\mathrm{e}\rangle\,\big)\,]\\ Z' &\equiv \widehat{E}[\,(\boldsymbol{\nu}\widetilde{n})\big(\,\overline{m_\mathrm{i}}k' \mid \mathsf{PP}_\mathbb{O}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \mid (\boldsymbol{\nu}k^*)\big(\,\mathsf{OM}_\mathbb{O}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \mid \mathsf{CM}[\,X_l\langle\, s\,\rangle_{r^*}^{k^*}\,]\,\big)\big)\,]\end{aligned}$$

for some key k', some set $\widetilde{v}$ of pre-processed requests, and $X_l\langle\, s\,\rangle$ denoting the respective continuation behavior of Table 4.

Note that the k^* in Z' is fresh, so it can be extruded over $\mathsf{PP}_\mathbb{O}\langle\ldots\rangle$ and $\overline{m_\mathrm{i}}k'$. As special cases, for $l \in \{\mathsf{png}\_r, \mathsf{sur}\_r\}$, of *committed* object managers, we define

$$\begin{aligned}F[\cdot] &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{n}k^*)\big(\,\overline{m_\mathrm{i}}k \big|\, \mathsf{PP}_\mathbb{O}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \;\big|\; \mathsf{OM}_\mathbb{O}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \;\big|\; [\cdot]\,\big)\\ \mathsf{pingO}_\mathbb{O}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle &\stackrel{\text{def}}{=} F[\mathsf{CM}[\,[\![\, s \,]\!]_{r^*}^{k^*}\,]]\\ \mathsf{surO}_\mathbb{O}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle &\stackrel{\text{def}}{=} F[\mathsf{CM}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_{r^*}^{k^*}\,]]\end{aligned}$$

and discuss their properties in Section 4.2.

4 On the Safety of Surrogation

In [16], we motivated the equation $a.\mathsf{ping} \doteq a.\mathsf{surrogate}$ for contextual equivalence $\doteq$ based on convergence as a valuable goal for proving safety of surrogation. Its interpretation is that an object a, if it is responsive to a ping-request, behaves the same as the object a after surrogation. One of the main observations in [16] was that the desired equation can not hold in full generality for Øjeblik-contexts $C[\cdot]$, in which the operation $x.\mathsf{surrogate}$ could occur internally. The reason is that, after *internal* surrogation, an object may misuse by intention the old and new references to *itself*. Actually, the advice to avoid internal surrogation is analogous to the fact that programmers, knowing that $x{=}0$, should never use division by x. In constrast, external surrogation corresponds to the case where a program receives x from some other module, so it should be guaranteed that x will never be 0. Analogously, we conjectured in [16] that in our semantics *external* surrogation is guaranteed to be safe.

In this section, we prove that $C[x.\mathsf{ping}]{\Downarrow}$ iff $C[x.\mathsf{surrogate}]{\Downarrow}$ for precisely those cases where $C[\cdot]$ will never lead to self-inflicted occurrences of $x.\mathsf{surrogate}$. Although this is an undecidable criterion [4], we may still formalize it in terms of our π-calculus semantics, as we do in Subsection 4.1, for its use in formal proofs. In Subsection 4.2, we study the behavior of objects before and after surrogation within tightly restricted contexts and prove them to be barbed equivalent. In Subsection 4.3, we then give an outline of the formal proof for the safety of external surrogations. The full paper also offers a static type system that guarantees that surrogations will never be internal. The full paper also offers a static type system that guarantees that surrogations will never be internal.

4.1 On the Absence of Internal Surrogation

Here, we study how to formalize that $C[\cdot]$ will never lead to self-inflicted occurrences of the term $x.\mathsf{surrogate}$, when plugged into the hole.

Recall from the Øjeblik semantics in § 3 that in a particular state $[\![\, a \,]\!]^k_p \;\Rightarrow Z$ in the computation of an arbitrary Øjeblik term a, a particular sur-request is self-inflicted, if $Z \equiv E[\; \overline{s}\langle \mathsf{sur}\_r, k\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle \;]$ with $k{=}k_{\mathrm{i}}$, because it is ready to enter the OM with $k{=}k_{\mathrm{i}}$ (c.f. Table 4). Since we must ensure that a sur-request *never* leads to internal surrogation, we must quantify over all derivatives of $[\![\, a \,]\!]^k_p$ and check for self-infliction in each of them.

Note that, starting from the term $[\![\, C[x.\mathsf{surrogate}] \,]\!]^k_p$, we should not be concerned with arbitrary sur-requests that appear at top-level during computation, but only with those that "arise from the request in the hole". However, this property is hard to determine for two different reasons: (1) *All* of the names mentioned in a sur-request may be changed dynamically by instantiation: s (due to forwarding), r (due to a call manager protocol), and k (due to pre-processing). (2) We have to consider arbitrarily many duplications of the request in the case that the hole appears, at the level of Øjeblik terms, within in a method body, which leads to replication in the π-calculus semantics. For both reasons, we need a tool to uniquely identify the various incarnations of the request.

Let $\mathsf{operate} \in \{\mathsf{ping}, \mathsf{surrogate}\}$, and let $\mathsf{op} \in \{\mathsf{png}, \mathsf{sur}\}$ denote the corresponding π-calculus label (c.f. Table 3). We introduce the *additional* Øjeblik labels $\mathsf{operate}^\star \in \{\mathsf{ping}^\star, \mathsf{surrogate}^\star\}$, writing $\mathcal{L}_T$ for the resulting language. The intuition is that tagged labels are semantically treated exactly like their untagged counterparts, but can syntactically be distinguished from them. Consequently, we give a tagged semantics, written $[\![\ ]\!]$, by adding the respective clauses for tagged labels, which are just copies of the clauses for the untagged labels; we use the tagged π-calculus labels $\mathsf{op}^\star \in \{\mathsf{png}^\star, \mathsf{sur}^\star\}$ at the semantics level. As a result, both tagged and untagged requests can be sent to object and alias managers; object managers ignore the tagging information of requests and treat $\mathsf{op}^\star$-and op-requests identically, but alias managers preserve the tagging information since they simply forward requests. We also add a tag to all parameterized definitions and abbreviations when considering the tagged semantics.

The semantics is not affected by including tagging information.

Lemma 4. *Let x be an Øjeblik variable and $C[\cdot]$ an untagged Øjeblik context. Then: $C[x.\mathsf{operate}]{\Downarrow}$ iff $[\![C[x.\mathsf{operate}^\star]]\!]^k_p{\Downarrow_p}$.*

However, tagging helps us to detect all "requests arising from the hole".

Definition 5 (Safe Contexts). *Let x be a variable and $C[\cdot]$ an untagged Øjeblik context. Then, $C[\cdot]$ is called* safe *for $x.\mathsf{surrogate}$, if $[\![C[x.\mathsf{surrogate}^\star]]\!]^k_p \Rightarrow_{\equiv} E[\ \overline{s}\langle \mathsf{sur}^\star\_r, k\rangle \mid \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle\ ]$ implies that $k \neq k_{\mathrm{i}}$.*

We replay the definition using ping instead of $\mathsf{surrogate}$. By definition of the semantics, an Øjeblik context $C[\cdot]$ is then safe for $x.\mathsf{surrogate}$ if and only if it is safe for $x.\mathsf{ping}$. For convenience, by abuse, we simply call $C[\cdot]$ to be *safe for x*.

4.2 Committed External Surrogation is Transparent

Our main result focuses on external surrogations. In the following we show that the two versions of an object manager at s that are committed to an external png- and sur-request, respectively (c.f.§ 3.3), are related by typed barbed equivalence.

Theorem 1. *Let $\Gamma \vdash \mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$ and $\Gamma \vdash \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$. Then:*

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \simeq_{\Gamma;s} \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle.$$

The proof of Theorem 1 requires several strong lemmas. Lemma 5 proves that surrogation results in an alias pointing to a clone of the old object. Its proof heavily relies on the nonces (c.f. page 399) used in the implementation of object and alias managers, which control the interference with the environment. Lemma 6 proves that the aliased object manager appearing in Lemma 5 behaves as a forwarder. Lemma 7 uses Lemma 1 to prove correctness of inlining. Lemma 8 proves that pre-processing external requests does not preclude other requests. Lemma 9 involves two confluent reductions from right to left along r^* and m_{i}, respectively. Finally, Theorem 1 can be established by applying the previous lemmas.

Lemma 5. *If Γ is a suitable type environment for the processes below, then:*

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \simeq_{\Gamma;s} (\boldsymbol{\nu} s^*)\big((\boldsymbol{\nu} k_{\mathrm{i}})\,\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*, k\rangle\big).$$

Lemma 6. *Let $\widetilde{v} := v_1..v_n$, and $v_j{:=}\langle\, l_j, k_j\,\rangle$ for $1{\leq}j{\leq}n$. If Γ is a suitable type environment for the processes below, then:*

$$(\boldsymbol{\nu} k_{\mathrm{i}})\,\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v}\,\rangle \simeq_{\Gamma;s} s \triangleright s^* \,\Big|\, \prod_{1\leq j\leq n} \overline{s^*} v_j.$$

Lemma 7. *Let P be a process and s a channel such that $s \notin \mathrm{fc}(P)$. If Γ is a suitable type environment for the processes below, then:*

$$(\boldsymbol{\nu} s^*)\,\big(\, s \triangleright s^* \mid P\,\big) \simeq_{\Gamma;s} P\{{}^{s}/_{s^*}\}.$$

Lemma 8. *Let $\widetilde{v} := v_1..v_n$ with $v_j{:=}\langle\, l_j, k_j\,\rangle$ and $k_j{\neq}k_{\mathrm{i}}$ for $1{\leq}j{\leq}n$. If Γ is a suitable type environment for the processes below, then:*

$$\prod_{1\leq j\leq n} \overline{s} v_j \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle \simeq_{\Gamma;s} (\boldsymbol{\nu} k_{\mathrm{i}})\,\mathsf{freeO}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, \widetilde{t}, \widetilde{v}\,\rangle.$$

Lemma 9. *Let $\widetilde{v} := v_1..v_n$ with $v_j{:=}\langle\, l_j, k_j\,\rangle$ and $k_j{\neq}k_{\mathrm{i}}$ for $1{\leq}j{\leq}n$. If Γ is a suitable type environment for the processes below, Then:*

$$\overline{r}\langle s, k\rangle \,\big|\, (\boldsymbol{\nu} k_{\mathrm{i}})\,\mathsf{freeO}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, \widetilde{t}, \widetilde{v}\,\rangle \simeq_{\Gamma} \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle.$$

4.3 External Surrogation is Safe

We prove our main theorem that $x.\mathsf{ping} \doteq x.\mathsf{surrogate}$ for safe contexts $C[\cdot]$.

Theorem 2 (Safety). *Let x be an object variable and $C[\cdot]$ an untagged context in Øjeblik. If $C[\cdot]$ is safe for x, then $C[x.\mathsf{ping}]{\Downarrow}$ iff $C[x.\mathsf{surrogate}]{\Downarrow}$.*

Proof. (Sketch) By Lemma 4, our proof obligation is equivalent to:

$$[\![\, C[x.\mathsf{ping}^\star] \,]\!]^k_p{\Downarrow_p} \;\text{ iff }\; [\![\, C[x.\mathsf{surrogate}^\star] \,]\!]^k_p{\Downarrow_p}.$$

This allows us to make use of the assumption on the safety of context $C[\cdot]$.

Since the semantics $[\![\ ]\!]$ is compositional, there is a π-calculus context $D[\cdot]$ and names y, j, q, such that $[\![\, C[x.\mathsf{operate}^\star] \,]\!]^k_p = D[\overline{y}\langle \mathsf{op}^\star\_q, j\rangle]$, where $D[\cdot]$ itself does not contain any message carrying a tagged request. We prove that

$$D[\overline{y}\langle \mathsf{png}^\star\_q, j\rangle]{\Downarrow_p} \;\text{ iff }\; D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle]{\Downarrow_p}.$$

and concentrate on the implication from right to left. The converse is analogous.

Assume that $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \Downarrow_p$. If $D[N]{\Downarrow_p}$ for every process N, then this is also the case for $N = \overline{y}\langle \mathsf{png}^\star\_q, j\rangle$; otherwise, the $\mathsf{sur}^\star$-request must contribute to the barb. Therefore, we assume $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \;\Rightarrow P \downarrow_p$ and show that there is

a corresponding $D[\overline{y}\langle \mathsf{png}^\star\_q, j\rangle] \Rightarrow_{\simeq_\Gamma} Q \downarrow_p$ where $Q = P[{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}]$. Since typed barbed equivalence $\simeq_\Gamma$ and relabeling preserve convergence, this suffices.

We distinguish between insignificant and significant transitions, where the former can be mimicked easily, while the latter require some work. Recall that a reduction step due to an external request is *committing* (c.f. § 3.3), if it represents the consumption of a pre-processed request by an object manager. Now, we combine this characterization with the fact that we have to concentrate on surrogation requests arising from the hole within the reduction sequence $D[\overline{y}\langle \mathsf{op}^\star, q, j\rangle] \Rightarrow P \downarrow_p$ and call *significant* ($\rightarrow_{\mathrm{s}}$) precisely those steps that exhibit the commitment to an external $\mathsf{op}^\star$-request. The other steps—except for the cases of internal surrogation, which are precisely excluded by assumption—are *insignificant*: they can even be mimicked up to structural equivalence.

In order to make the proof work, we iterate the simulation steps along the given sequence $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \Rightarrow P \downarrow_p$. Let us assume that this sequence has d significant steps and that we have iterated already $h-1$ of them, for $0 < h \leq d$:

$$D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle]\ (\Rightarrow \rightarrow_{\mathrm{s}})^{h-1}\ \boxed{P_{h-1}}\ \Rightarrow \rightarrow_{\mathrm{s}}\ \boxed{P_h} \equiv \widehat{E}[\ \mathsf{surO}^\star_{\mathbb{O}}\langle\, s_h, \ldots\,\rangle\ ]$$

By (the tagged counterparts of) Lemma 3, we can precisely localize the state of the committed object manager inside P_h after the significant step (c.f. § 4.2). With respect to the relabeling $\rho := [{}^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}]$, by assumption and iteration, we also have the sequence:

$$D[\overline{y}\langle \mathsf{png}^\star\_q, j\rangle]\ (\Rightarrow \rightarrow_{\mathrm{s}})^{h-1} \simeq_\Gamma\ \boxed{P_{h-1}\rho}\ \Rightarrow \rightarrow_{\mathrm{s}}\ \boxed{Q_h} \equiv \widehat{E}[\ \mathsf{pingO}^\star_{\mathbb{O}}\langle\, s_h, \ldots\,\rangle\ ]\rho$$

by consuming a $\mathsf{png}^\star$-request. Now, we apply (the tagged counterparts of) Theorem 1 and Lemma 2, and the fact that barbed equivalence $\simeq_\Gamma$ implies structural equivalence $\equiv$ and is preserved by relabeling and get $Q_h \simeq_\Gamma P_h\rho$. This means that we can mimic the significant steps, thus the whole sequence, up to $\simeq_\Gamma$.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs CS. Springer, 1996.

[2] R. M. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[3] L. Cardelli. `obliq-std.exe` — Binaries for Windows NT. `http://www.luca.-demon.co.uk/Obliq/Obliq.html`, 1994.

[4] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995. An exteded abstract as appeared in *Proceedings of POPL '95*

[5] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the join-calculus. In *Proceedings of POPL '96*, ACM, 1996.

[6] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and Equivalence of Imperative Objects. In *Proceedings of FSTTCS '97*, *LNCS* 1346. Springer, 1997.

[7] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Proceedings of ECOOP '91*, volume 512 of *LNCS*, Springer, 1991.

[8] K. Honda and N. Yoshida. On Reduction-Based Process Semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[9] H. Hüttel and J. Kleist. Objects as Mobile Processes. Research Series RS-96-38, BRICS, 1996. Presented at *MFPS '96*.
[10] J. Kleist and D. Sangiorgi. Imperative Objects and Mobile Processes. In *Proceedings of PROCOMET '98*. Chapman & Hall, 1998.
[11] M. Merro. *Local π: A Model for Concurrent and Distributed Programming Languages*. PhD thesis, Ecole des Mines, France, 2000.
[12] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proceedings of ICALP '98*, volume 1443 of *LNCS*. Springer, 1998.
[13] R. Milner. The Polyadic π-calculus: A Tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.
[14] R. Milner and D. Sangiorgi. Barbed Bisimulation. In *Proceedings of ICALP '92*, volume 623 of *LNCS*. Springer, 1992.
[15] U. Nestmann. Mobile Objects (A Project Overview). In *Proceedings of FBT '99*. Herbert Utz Verlag, 1999.
[16] U. Nestmann, H. Hüttel, J. Kleist, and M. Merro. Aliasing Models for Mobile Objects. Accepted for *Journal of Information and Computation*. An extended abstract has appeared as Distinguished Paper in the *Proceedings of EUROPAR '99*, pages 1353–1368, LNCS 1685, September 1999, 1999.
[17] B. C. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
[18] A. Philippou and D. Walker. On Transformations of Concurrent Object Programs. *Theoretical Computer Science*, 195(2):259–289, 1998.
[19] D. Sangiorgi. An Interpretation of Typed Objects into Typed π-calculus. *Information and Computation*, 143(1):34–73, 1998.
[20] D. Sangiorgi. Typed π-calculus at Work: A proof of Jones's parallelisation theorem on Concurrent Objects. *Theory and Practice of Object-Oriented Systems*, 5, 1999.
[21] C. L. Talcott. Obliq Semantics Notes. Unpublished note. Available from `clt@cs.stanford.edu`, Jan. 1996.
[22] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.