Location Independent Routing in Process Network Overlays

Mads Dam
School of Computer Science and Communication
KTH Royal Institute of Technology

Abstract

Location transparency, i.e. the decoupling of objects, applications, and VM's from their physical location, is a highly desirable property, for instance to aid application development, to help management and fault isolation, and to support load balancing and efficient resource allocation. Building location transparent systems on top of ip requires objects to be addressed in terms of their hosting node. When objects migrate their hosting node changes, creating a need for a message routing infrastructure using e.g. location servers or message forwarding chains. Such an infrastructure is costly in terms of complexity and overhead. By instead using location independent routing, it is possible to direct messages to the receiving object instead of that objects location, which may be out of date. We show how this idea allows complex object overlays to be implemented in a sound, fully abstract, and efficient (lock-free) manner on top of an abstract network of processing nodes connected by asynchronous point to point communication channels.

1 Introduction

The decoupling of applications from their physical realization, intimately tied to the concept of virtualization, is a recurrent theme in the history of computing. Running applications on virtual machines allows many tasks to be performed independently of the physical computing infrastructure. By migrating virtual machine images between physical processors it is possible for a cloud provider to adapt processing and communication load to changing application demands and to changes in the physical infrastructure. In this way applications can, at least in principle, be freed of the burden of resource allocation. That is, it is left to an underlying processing network to determine on which nodes to place which tasks in order to make efficient use of processing resources, while at the same time meeting requirements on response times and processing capacity. If realized, the result is simpler application logic, better service quality, and, ultimately, lower costs for development, operations and management.

The question is how to realize this potential with a minimum of overhead, and in such a way that applications behave in a predictable manner.

We examine this question in the context of a rudimentary distributed object language, and propose a formal, networked, runtime semantics of this language with some quite novel features. The goal is a “bare-metal” style of semantics where all aspects of computing and communication are accounted for in terms of local operations that could be directly implemented on top of silicon or, say, a hypervisor such as Xen [3].

A key problem is how to handle object and task mobility in an efficient manner. Since the allocation of objects to nodes is dynamic, some form of application level routing is needed to ensure that messages reach their destinations quickly, and with minimal overhead. Various approaches have been considered in the literature (cf. [36] for a survey):

- One option is to maintain a centralized or distributed database of object locations. Such a database can be used for both forwarding, by routing messages through the forwarding server, and for location querying, by using the database to look up destination object locations. In either case, object location and the location database must be kept consistent, which requires synchronization. Many experimental object mobility systems in the literature use some form of replicated or distributed location databases, cf. [13, 36, 4, 19].

- Another option is for nodes to maintain forwarding pointers, as in the Emerald system [24]. Migration then causes forwarding pointer chains to be extended by one further hop, and some mechanism is typically used to piggyback location update information onto messages, to ameliorate forwarding chain growth. This mechanism is used, for instance, in JoCaml [9].

- Many solutions involve some form of broadcast or multicast search. For instance, an object may use multicasting to find an object if a pointer for some reason has become stale, as in Emerald, or for service discovery, as in Jini [1].

- Other solutions have been explored too, such as tree-structured DNS-like location directories [41], Awerbuch and Peleg’s distributed directories [2], and Demmer and Herlihy’s arrow protocol [12].

The main source of the difficulties these approaches are designed to solve is the distinction between destination host identifiers (location) and search identifiers (object identity). But, in a fully mobile setting the location at which an object resides has no intrinsic interest[1]. What is of interest is message destination, that an rpc destined for the object with identity o is routed to the location where o resides, and not somewhere else. The address of the destination is not relevant. In other words, rather than routing messages according to ip address, inter object message routing should really be done according to the identity of the destination object rather than an assumed host identity, which might for all the sender knows be completely out of date.

[1] Location has interest as a source of latency, for instance, but that is another matter.

This problem is in fact well known in the networking community, and has been the subject of significant attention over the last decade. The idea is to replace the location-based routing of traditional ip networks with location independent schemes that route messages according to names, or content. Names can be flat, unstructured identifiers, as in [6], or they can encode some form of signed content identity, as in content centric networking [20]. The general goal is to devise routing schemes for flat name spaces that are *compact*, such that routing tables can be represented at each node using space sublinear in the size of the network, and such that path lengths, and hence message latencies, do not grow too far from the optimal. The latter requirement of low *stretch*, defined as the ratio of route length to shortest path length, precludes the use of both location registers and hierarchical ip-like naming schemes.

The main purpose of this paper is to show that name-based routing offers a new space for solutions to the object mobility problem with some attractive properties:

- No centralized or decentralized object location database is needed, since the network routing mechanism itself ensures that messages are routed to their proper hosts

- A whole swathe of software becomes superfluous, which manages address lookups, message forwarding, rerouting, address bookkeeping, and the synchronization overhead between location registers and the migrating objects is eliminated. As a result, the "trusted computing base" of the networked execution platform is significantly reduced, in terms of size, and complexity

- Traffic overhead is decreased. First, mobility support on top of IP needs to perform routing both at IP and at application level. Name-based routing in effect eliminates the need for IP level routing. Moreover, in steady state the simple distance vector routing scheme used here has stretch 1, so message delivery overhead is minimal (however, distance vector routing is not compact[2], so our scheme does not scale well. We leave this issue for future work).

- Improved robustness: In faulty situations, if connection to the location register is lost, message delivery is impossible (or needs to resort to more costly mechanisms such as broadcasting, as in Emerald or Jini). Routing can be made self-stabilizing and thus able to adapt to any type of disturbance, as long as connectivity is maintained. This allows computation to progress (including delivery of messages and migration of objects) even when the network is under considerable churn

On the other hand, throwing away network layers 3 and above may seem an excessive price to pay, and the argument above that the cost of ip routing should be taken into account is evidently invalid if ip routing needs to be performed anyway in any realistic implementation. We argue, however,

[2]Routing tables in distance vector routing have size $\Omega(n \log n)$

that this does not need to be the case. First, as we have noted, the general architecture of the future internet is currently very much in flux. Second, although we have so far only explored an early prototype simulator [10] built on top of an ip-based overlay it is perfectly possible today to build large scale non-ip networks with only layer 2 connectivity, sufficient to bootstrap our approach. Third, the simplicity of our approach in comparison to the task of formally verifying, e.g. IP and TCP [5], opens up for the possibility of extending currently ongoing work on formally verified low level software along the lines of seL4 [25] to fully networked operating systems and hypervisors at the device and instruction level. Finally, even if amending the current ip naming schemes turns out to be infeasible, it is still of interest to evaluate the consequences of a much tighter integration between network infrastructure and application level runtimes than what is currently done.

Our study is set in the context of a distributed object language μABS, a rudimentary fragment of the ABS language studied in the EU FP7 project HATS. The ABS language [22] is an intermediate level modeling and programming language in which a number of phenomena related to software evolvability and adaptation can be studied. The μABS fragment includes only a rudimentary set of features, sufficient, however, to allow simple networked programs to be programmed in a natural way, as we illustrate by a couple of small examples. The μABS language is class based, and has features for remote method invocation, object creation, and standard sequential control structures, similar to Sewell et al's Nomadic Pict language [36]. Two semantics of μABS are given, one a standard semantics in rewriting logic style which does not take into account aspects related to location, naming, routing, or communication. The second semantics takes these aspects into account by executing objects on top of an arbitrary, but concrete, processor network. The main result of the paper is to show that the network semantics is sound and fully abstract with respect to the reference semantics. We base the analysis on barbed equivalence [33]. Barbed equivalence requires a witness relation that preserves some set of primitive observations, here remote calls to external objects, in both directions, and is preserved under weak reductions, also in both directions. Barbed equivalence is convenient since the required (unlabelled) reduction relations are easy and fairly uncontroversial to define, both for the reference semantics and for the network semantics. Using a labelled transition semantics particularly at the network level is much more complex, and needs to be subject to a separate justification which is out of scope for the present work.

We structure the presentation as follows: First, in sections 3 to 5 the μABS language is introduced, along with the reference semantics and the notion of barbed equivalence. We then turn to the network semantics. In section 6 we first introduce runtime states, or configurations, including routing and network graphs. The reduction relation is presented in section 7. The subsequent analysis relies on a collection of simple well-formedness conditions that are introduced in section 8, along with a proof that reachable configurations are well-formed. After adapting barbed equivalence to the network semantics we then, in section 10, embark on the soundness and full abstraction proof. A key tool is two normal form theorems which allow configurations in the network semantics to be normalized in a behaviour preserving way. The first normal form theorem allows to stabilize routing and at least partially empty message queues. The second normal form theorem serves to as much as possible empty message queues and to migrate all objects to a single central node. Once this is done, soundness and full abstraction is proved, in section 11. The proofs are a bit lengthy but not really complicated, mainly due to the simplicity of the source language. Finally in section 12 we discuss future and related work, including ongoing extensions of this work to richer source languages [11] and to resource allocation [10].

Acknowledgements Many thanks are due to Karl Palmskog for valuable contributions in the early phases of this work, and to Andreas Lundblad and Ali Jafari.

2 Notation

We use a standard boldface vector notation to abbreviate sequences, for compactness. Thus, $\mathbf{x}$ abbreviates a sequence $x_0, \ldots, x_n$, possibly empty, and $\mathbf{f}(\mathbf{x})$ abbreviates a sequence $f_1\ x_1, \ldots, f_n\ x_n$, etc. Let $\mathbf{x} = x_1, \ldots, x_n$. Then $x_0, \mathbf{x}$ abbreviates $x_0, \ldots, x_n$. Let $g : A \to B$ be a finite map. The update operation for g is $g[b/a](x) = g(x)$ if $x \neq a$ and $g[b/a](a) = b$. We use $\perp$ for bottom elements, and $A_\perp$ for the lifted set with partial order $\sqsubseteq$ such that $a \sqsubseteq b$ if and only if either $a = b \in A$ or else $a = \perp$. Also, if x is variable ranging over A we often use $x_\perp$ as a variable ranging over $A_\perp$. For g a function $g : A \to B_\perp$ we write $g(a) \downarrow$ if $g(a) \in B$, and $g(a) \uparrow$ if $g(a) = \perp$. The product of sets (flat cpo's) A and B is $A \times B$ with pairing (a, b) and projections π_1 and π_2.

3 μABS

The syntax of the object language μABS is presented in fig. 1. Programs are

$x, y \in \mathit{Var}$		Variables	$e \in \mathit{Exp}$ Expression
P	$::=$	$\boldsymbol{CL}\{\mathbf{x}, s\}$	Program
CL	$::=$	$\textbf{class}\ C(\mathbf{x})\{\mathbf{y}, \mathbf{M}\}$	Class definition
M	$::=$	$m(\mathbf{x})\{\mathbf{y}, s\}$	Method definition
s	$::=$	$s_1; s_2 \ \mid\ x = \mathit{rhs} \ \mid\ \textbf{skip}$	Statement
		$\mid \textbf{if}\ e\{s_1\}\ \textbf{else}\ \{s_2\}$	
		$\mid \textbf{while}\ e\{s\} \ \mid\ e!m(\mathbf{e})$	
rhs	$::=$	$e \ \mid\ \textbf{new}\ C(\mathbf{e})$	Right hand sides

Figure 1: μABS_1 abstract syntax

sequences of class definitions, along with a set of global variables $\mathbf{x}$, and a "main" statement s. The class hierarchy is flat and fixed. Objects have

```
class Server1(){,
  serve(from,x){,from!response(foo(x))}
} ,
class Client(arg){,
  use(server){,server!serve(self,arg)},
  response(y){,env!output(y) }
}
{
server, client,
server = new Server1() ;
client = new Client(42) ;
client!use(server)
}
```

Figure 2: Simple server

parameters $\mathbf{x}$, local variable declarations $\mathbf{y}$, and methods $\mathbf{M}$. Methods have parameters $\mathbf{x}$, local variable declarations $\mathbf{y}$ and a statement body. For simplicity we assume that variables have unique declarations. The definition of expressions e is left open, but we require that expressions are side-effect free. Types are omitted from this presentation. It is possible to add types and a notion of well-typedness. However, this will not affect the presentation in any significant way, and for this reason we choose to work in an untyped setting.

Besides standard sequential control structures (the choice of which is largely irrelevant), statements involve a minimal set of constructs for asynchronous method invocation and object creation. Sequential composition is associative with unit **skip**. That is, the statements $s;\mathbf{skip}$, $\mathbf{skip};s$ and s are identified. Method bodies lack a return statement. We consider return statements with futures in a companion paper [11]. For now, method bodies are simply evaluated to the end at which point the evaluating task is terminated. In the absence of return statements, objects communicate using callbacks in a manner which is not dissimilar to communication in Erlang, as illustrated in the following examples.

Example 3.1. A very simple server applying `foo` to its argument is shown in fig. 2. We assume a reserved OID `env` with reserved method `output` to be used as a standard output channel.

Example 3.2. Just to show that the language is not trivial, the program in fig. 3 constructs an object ring with (here) 42 elements. A value is circulated along the ring, computing `bar(...(bar(x,42),41) ...,1)`. Each ring element decrements an iterator `iter` initialized to the original value 42 first received. The final ring element returns the final value to the server, which then finally returns it to the client through the `output` method of object `env`.

```
class Server(){,
  serve(from,x){c,
    c = new Cell() ;
    c!process(x,self,x)}
 return(result){from!response(result)}
} ,
class Cell(){,
  process(x,root,iter){c,
    if iter = 0{root!return(y)}
    else {
      c = new Cell() ;
      c!process(bar(x,iter),root,iter -1)}
 }
} ,
class Client(arg){,
  use(server){,server!serve(self(),arg)},
  response(y){,env!output(y) }
}
{
server, client,
server = new Server() ;
client = new Client(42) ;
client!use(server)
}
```

Figure 3: Dynamic ring

4 Type 1 Reduction Semantics

We first present a reduction semantics for μABS in rewriting logic style. This semantics is important as the point of reference for later refinements. The reduction semantics uses a reduction relation $cn \to cn'$ where cn, cn' are *configurations*, as determined by the runtime syntax in fig. 4. Later on, we introduce different configurations and transition relations, and so use index 1, or talk of e.g. configurations of "type 1", for this first semantics when we need to disambiguate. Terms of the runtime syntax are ranged over by M,

$x \in \mathit{Var}$			Variables
$o \in \mathit{OID}$			Object id
$p \in \mathit{PVal}$			Primitive values
$v \in \mathit{Val}$	$=$	$\mathit{PVal} \cup \mathit{OID}$	Values
$l \in \mathit{MEnv}$	$=$	$\mathit{Var} \to \mathit{Val}$	Method environment
$a \in \mathit{OEnv}$	$=$	$\mathit{Var} \to \mathit{Val}$	Object environment
$\mathit{tsk} \in \mathit{Tsk}$	$::=$	$\mathsf{t}(o, l, s)$	Task
$\mathit{obj} \in \mathit{Obj}$	$::=$	$\mathsf{o}(o, a)$	Object
$\mathit{cl} \in \mathit{Call}$	$::=$	$\mathsf{c}(o, o', m, \mathbf{v})$	External call
$\mathit{ct} \in \mathit{Ct}$	$::=$	$\mathit{tsk} \mid \mathit{obj}$	Container
$\mathit{cn} \in \mathit{Cn}$	$::=$	$0 \mid \mathit{ct} \mid \mathit{cn}\ \mathit{cn}' \mid \mathsf{bind}\ o.\mathit{cn}$	Configuration

Figure 4: $\mu\mathrm{ABS}_1$ type 1 runtime syntax

and $\preceq$ is the subterm relation. The runtime syntax uses disjoint, denumerable sets of object identifiers $o \in \mathit{OID}$ and primitive values $p \in \mathit{PVal}$. Lifted values are ranged over by $v_\perp \in \mathit{Val}_\perp$. We often refer to OID's as *names*, and bind OID's using the π-like binder bind. The free names of configuration cn is the set $\mathit{fn}(cn)$, as usual, and $\mathit{OID}(cn) = \{o \mid \exists a.\mathsf{o}(o, a) \preceq cn\}$ is the set of OID's of objects occurring in cn. Standard alpha-congruence applies to name binding (but is dropped once we move to the network semantics).

In order for computations to have observable effects we assume a fixed set Ext of external OID's with dedicated methods, such as the OID `env` and the method `output` in examples 2 and 3. External OID's are not allowed to be bound.

Method and object environments l and a, respectively, map local variables to assignable values. Upon invocation, the method environment is initialized using the operation $\mathit{locals}(o, m, \mathbf{v})$ by mapping the formal parameters of m in o to the corresponding actual parameters in $\mathbf{v}$, by initializing the method local variables to a suitable null value, and by mapping self to o. Object environments are initialized using the operation $\mathit{init}(C, \mathbf{v})$, which maps the parameters of C to $\mathbf{v}$, and initializes the object local variables as above.

Configurations are multisets of containers of which there are three types, tasks, objects, and external calls. Object identifiers are scoped within configurations using the π-like binder bind. Configuration juxtaposition is assumed to be commutative and associative with unit 0. In addition we assume the standard structural identities bind $o.0 = 0$ and bind $o.(cn_1\ cn_2) =$ (bind $o.cn_1)\ cn_2$ when $o \notin \mathit{fn}(cn_2)$. We often use a vectorized notation

$\mathsf{bind}\ \mathbf{o}.cn$ as abbreviation, letting $\mathsf{bind}\ \varepsilon.cn = cn$ where ε is the empty sequence. The structural identities then allows us to rewrite each configuration into a *standard form* $\mathsf{bind}\ \mathbf{o}.cn$ such that each member of $\mathbf{o}$ occurs free in cn, and cn has no occurrences of the binding operator bind. We use standard forms frequently below.

In addition to $locals$ and $init$, the reduction rules presented below use the following helper functions:

- $body(o, m)$ retrieves the statement of the shape s in the definition body for m in the class of o
- $\hat{e}(a, l) \in Val$ evaluates e using method environment l and object environment o

Call containers play a special, somewhat subtle role in defining the external observations of a configuration cn. An observation, or *barb*, is a call expression of the form $o!m(\mathbf{v})$, ranged over by obs. In order to define the observations of a given configuration, we assume a fixed set Ext of external OID's to which outgoing method calls can be directed. Names in Ext are not allowed to be bound. A barb, then, is an external method call, i.e. a method call to an OID in Ext. Calls that are not external are meant to be completed in usual reduction semantics style, by internal reaction with the called object, to spawn a new task. External calls could be represented directly, without introducing a special container type (which is not present in the core ABS semantics of [22]), by saying that a configuration cn has barb $obs = o!m(\mathbf{v})$ if and only if cn has the shape

$$\mathsf{bind}\ \mathbf{o_1}.(cn'\ \mathsf{o}(o_2, a)\ \mathsf{t}(o_2, l, e_1!m(\mathbf{e_2}); s))\ , \qquad (1)$$

where $\hat{e_1}(a, l) = o \in Ext$ and $\hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l}) = \mathbf{v}$. However, in a semantics with unordered communication, which is what is assumed of core ABS [22], and which we also implement in this paper, consecutive calls should commute, i.e. there should be no observational distinction between the method bodies $e_1!m_1(\mathbf{e'_1}); e_2!m_2(\mathbf{e'_2})$ and $e_2!m_2(\mathbf{e'_2}); e_1!m_1(\mathbf{e'_1})$. This, however, is difficult to reconcile with the representation (1). To this end call containers are introduced, to allow configurations like (1) to produce a corresponding call, and then proceed to elaborate s.

Figures 5 and 6 present the reduction rules, using the notation $cn \vdash cn' \to cn''$ as shorthand for $cn\ cn' \to cn\ cn''$. We use $\to_1$ when we want to make the reference to the type 1 reduction semantics explicit. Fig. 5 gives the mostly routine rules for assignment, control structures, and contextual reasoning, and fig. 6 gives the slightly more interesting rules for inter-method communication and object creation. We note some basic properties of the reduction semantics.

Proposition 4.1. *Let $cn \to_1 cn'$.*

1. $fn(cn) \subseteq fn(cn')$
2. *If $\mathsf{o}(o, a) \preceq cn$ then $\mathsf{o}(o, a') \preceq cn'$ for some object environment a'*

ctxt-1: If $cn_1 \to cn_2$ then $cn \vdash cn_1 \to cn_2$

ctxt-2: If $cn_1 \to cn_2$ then $\mathsf{bind}\ o.cn_1 \to \mathsf{bind}\ o.cn_2$

wlocal: If $x \in dom(l)$ then $\mathsf{t}(o, l, x = e; s) \to \mathsf{t}(o, l[\hat{e}(a, l)/x], s)$

wfield: If $x \in dom(a)$ then $\mathsf{o}(o, a)\ \mathsf{t}(o, l, x = e; s) \to \mathsf{o}(o, a[\hat{e}(a, l)/x])\ \mathsf{t}(o, l, s)$

skip: $\mathsf{t}(o, l, \mathbf{skip}) \to 0$

if-true: If $\hat{e}(a, l) \neq 0$ then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{if}\ e\{s_1\}\ \mathbf{else}\ \{s_2\}; s) \to \mathsf{t}(o, l, s_1; s)$

if-false: If $\hat{e}(a, l) = 0$ then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{if}\ e\{s_1\}\ \mathbf{else}\ \{s_2\}; s) \to \mathsf{t}(o, l, s_2; s)$

while-true: If $\hat{e}(a, l) \neq 0$ then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\{s_1\}; s) \to \mathsf{t}(o, l, s_1; \mathbf{while}\ e\{s_1\}; s)$

while-false: If $\hat{e}(a, l) = 0$ then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\{s_1\}; s) \to \mathsf{t}(o, l, s)$

Figure 5: μABS$_1$ reduction rules part 1

Proof. We note that no structural identity nor any reduction rule allows an OID to escape its binder. The result follows. □

Consider a program $\boldsymbol{CL}\{\mathbf{x}, s\}$. Assume a reserved OID *main*. A *type 1 initial configuration* is any configuration of the shape

$$cn_{init} = \mathsf{o}(main, \bot)\ \mathsf{t}(main, l_{init}, s)$$

where $\bot$ is the everywhere undefined object environment and $l_{init,1}$ is the initial type 1 method environment assigning default values to the variables in $\mathbf{x}$.

We say that a configuration cn_n of type 1 is *reachable* if there is a derivation $cn_{init} = cn_0 \to_1 \cdots \to_1 cn_n$ where cn_{init} is an initial configuration. Reachable configurations satisfy some well-formedness conditions which we note.

Definition 4.2 (Type 1 Well-formedness)**.** A configuration cn is *type 1 well-formed* (WF1) if cn satisfies:

1. OID Uniqueness: Suppose $\mathsf{o}(o_1, a_1), \mathsf{o}(o_2, a_2) \preceq cn$ are distinct object occurrences. Then $o_1 \neq o_2$
2. Task-Object Existence: If $\mathsf{t}(o, l, s) \preceq cn$ then $\mathsf{o}(o, a) \preceq cn$ for some object environment a

call: Let $o' = \hat{e_1}(a, l)$ in $\mathsf{o}(o, a)\ \mathsf{o}(o', a') \vdash \mathsf{t}(o, l, e_1!m(\mathbf{e_2}); s) \to$
$\mathsf{t}(o, l, s)\ \mathsf{t}(o', locals(o', m, \hat{\mathbf{e_2}}(a, l)), body(o', m))$

call-ext: If $o' = \hat{e_1}(a, l) \in Ext$ then
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, e_1!m(\mathbf{e_2}); s) \to \mathsf{t}(o, l, s)\ \mathsf{c}(o, o', m, \hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l}))$

new: $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, x = \mathbf{new}\ C(\mathbf{e}); s) \to \mathsf{bind}\ o'.\mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', init(C, \hat{\mathbf{e}}(a, l)))$

Figure 6: μABS reduction rules part 2

3. Object Existence: Suppose $o \notin \mathit{Ext}$ occurs in cn. Then $\mathsf{o}(o, a) \preceq cn$ for some object environment a
4. Object Nonexistence: Suppose $o \in \mathit{Ext}$. Then $\mathsf{o}(o, a) \not\preceq cn$ for any object environment a
5. Object Binding: Suppose $o \notin \mathit{Ext}$. Then $o \notin \mathit{fn}(cn)$

Well-formedness is important as it ensures that objects, if defined, are defined uniquely. The existence properties are important to make sure that the partitioning of OID's into external and (by extension) internal is meaningful, in that external references are always routed outside the "current configuration".

Proposition 4.3 (WF1 Preservation)**.** *If cn is type 1 wellformed and $cn \rightarrow cn'$ then cn' is type 1 wellformed.*

Proof. Routine. □

Theorem 4.4. *If cn is type 1 reachable then cn is fully type 1 wellformed.*

Proof. It suffices to check that any initial configuration is fully type 1 wellformed, and then use proposition 4.3. □

5 Barbed Equivalence

Our approach to implementation correctness is based on the notion of barbed equivalence [33], a notion of equivalence often used to relate transition systems determined by a reduction semantics, cf. [8, 15, 18]. Our goal is to show that it is possible to remain strongly faithful to the reference semantics, provided all nondeterminism is deferred to be handled by a separate scheduler. This allows to draw strong conclusions also in the case a scheduler is added, as we discuss later. Barbed equivalence requires of a pair of equivalent configurations that the internal transition relation $\rightarrow$ is preserved in both directions, while preserving also a set of external observations. Although weaker than corresponding equivalences such as bisimulation equivalence on labelled transition systems, barbed equivalence in nonetheless of interest for the following two reasons:

1. Barbed equivalence offers a reasonable account of observationally identical behaviour on *closed systems*, i.e. when composition of (in our case) subconfigurations to build larger configurations is not considered because it a) is for some reason not important or relevant, or b) does not offer new observational capabilities.
2. Barbed equivalence can be strengthened in a natural way to *contextual equivalence* [32] by adding to barbed equivalence a natural requirement of closure under context composition. Furthermore, a number of works [21, 34] have established very strong relations between contextual equivalence for reduction oriented semantics and bisimulation/logical relation based equivalences for sequential and higher-order computational models.

It is, however, far from trivial to devise a natural notion of context that works at the level of the network semantics introduced later, and such that the notions of context correspond at both the abstract, reference semantics level we consider at present, and at the network level. For this reason the account of this paper based on barbed equivalence is also a natural stepping stone towards a deeper study of the notion of context in real-world—or at least not overly artificial—networked software systems.

Let $obs = o'!m(\mathbf{v})$. The observation predicate $cn \downarrow obs$ is defined to hold just in case cn can be written in the form

$$\mathsf{bind}\ \mathbf{o}.(cn'\ \mathsf{c}(o', m, \mathbf{v}))\ .$$

The derived predicate $cn \Downarrow obs$ holds just in case $cn \to^* cn' \downarrow obs$ for some cn'.

Let now $\mathcal{R}$ be a binary relation on type 1 well-formed configurations. We are interested in relations with the following properties:

- *Symmetry*: If $cn_1\ \mathcal{R}\ cn_2$ then $cn_2\ \mathcal{R}\ cn_1$
- *Reduction-closure*: If $cn_1\ \mathcal{R}\ cn_2$ and $cn_1 \to cn_1'$ then there exists some cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1'\ \mathcal{R}\ cn_2'$
- *Barb preservation*: If $cn_1\ \mathcal{R}\ cn_2$ and $cn_1 \downarrow obs$ then $cn_2 \Downarrow obs$

We call a relation with these three properties a *type 1 witness relation*.

Definition 5.1 (Type 1 Barbed Equivalence)**.** Let $cn_1 \simeq_1 cn_2$ if, and only if, $cn_1\ \mathcal{R}\ cn_2$ for some type 1 witness relation $\mathcal{R}$.

Barbed equivalence is the reference behavioral identity to which other equivalences are compared in the remainder of the paper.

Example 5.2. Even if this is somewhat out of scope we sketch for completeness a proof that the programs of example 2 and 3 are barbed equivalent. The external OID's is the single `env`. We construct a relation $\mathcal{R}$ relating configurations of the simple server with configurations of the dynamic ring. For the former, a reachable configuration has one of the following three forms:

- The initial configuration with an object with OID $main$ and a task at line 10
- The initial object, a task at l. 11, a `Server` object
- The initial object, a task at l. 12, a `Server` object, and a `Client` object
- The initial object, a `Server` object, a `Client` object, and an invoked `use` task
- The initial object, a `Server` object, a `Client` object, and an invoked `serve` task
- The initial object, a `Server` object, a `Client` object, and an invoked `response` task

All configurations have exactly one observation, namely `env!output`(42). For the dynamic ring, the state space is a little more complex. It contains:

- The initial configuration with an object with OID $main$ and a task at line 20
- The initial object, a task at l. 21, a `Server` object
- The initial object, a task at l. 22, a `Server` object, and a `Client` object
- The initial object, a `Server` object, a `Client` object, and an invoked `use` task
- The initial object, a `Server` object, a `Client` object, and an invoked `serve` task
- The initial object, a `Server` object, a `Client` object, $42 - n$ `Cell` objects, and a `process` task at one of lines 9-12, invoked with arguments `bar(...(bar(42,42),41)...,42`$-n$`)`, OID of `Server`, and `iter` = 42.
- The initial object, a `Server` object, a `Client` object, 42 `Cell` objects, and a `response` task ready to execute `env!output`(42).

It is straightforward though somewhat cumbersome to verify that the relation obtained by relating any of the reachable pairs (cn_1, cn_2) where cn_1 is a configuration of the simple server and cn_2 is a configuration of the dynamic ring is a barbed bisimulation.

An important shortcoming of barbed equivalence as we have introduced it here is that prima facie it does not take program structure, or configuration structure, into account. Thus, there is no guarantee, for instance, that if $cn_1 \simeq_1 cn_2$ then, for all cn, $cn_1\ cn \simeq_1 cn_2\ cn$. With the current static internal/external partitioning of OID's the question is not that meaningful, as e.g. cn and cn' are unable to communicate when both are closed. But closedness is important to equip configurations with clearly defined interfaces. It is possible to refine the notion of wellformedness to better capture composition of configurations. This allows to consider a form of contextual equivalence [32], by extending the three conditions above with a condition of *contextuality*, saying that whenever $cn_1\ \mathcal{R}\ cn_2$ (subject to suitable wellformedness constraints) then $cn_1\ cn\ \mathcal{R}\ cn_2\ cn$. This, however, complicates matters quite considerably when we turn to the network semantics below, and for this reason we defer a proper treatment of network composition and contextual equivalence to a future publication.

6 Network Semantics: Runtime Configurations

The "standard" (type 1) semantics for μABS is quite abstract and does not account for many issues which must be faced by an actual implementation, in particular if the goal is high performance and scalability. For instance:

- The μABS semantics implements a rendez-vous oriented communication model. We want to account for this using a standard buffered asynchronous model
- Accordingly, calls should be replaced by message passing
- The μABS semantics has no concept of proximity or name space. Any two objects, regardless of their "location" can without any overhead or search choose to synchronize at any point. Instead, we want a semantics that is *network aware* in the sense that it brings out proximity and location without unduly constraining the model, for instance to a particular naming discipline, or to a centralized name or location lookup service

Our proposal is to execute μABS objects on a network graph in a fully decentralized and lock free manner where the only means of communication or synchronization is by asynchronous message passing along edges connecting neighbouring nodes, each edge having an associated directional, buffered communication channel. In this section we accordingly introduce a refinement of the standard semantics, a “network semantics”, or type 2 semantics, which adds an explicit network components to the type 1 semantics. The key idea is to use name-based routing, as explained in the introduction. That is, nodes are equipped with explicit routing information allowing messages to be addressed to specific receiving objects, rather than their hosts, which may change. This allows a very simple, fully decentralized, and lock free integration of routing and object migration, as we now begin to demonstrate.

Nodes and Routing The network semantics is presented in rewriting logic style, similar to the type 1 semantics above. We still have configurations cn, but these now have a richer structure. We first introduce two new types of container to reflect the underlying network graph, namely *nodes* and *links*. Node containers have the form

$$\mathsf{n}(u, t)$$

where $u \in \mathit{NID}$ is a primitive *node identifier*, and where t is an associated routing table. Node identifiers (NID’s) take the place of ip addresses in the usual ip infrastructure. For routing we assume a rudimentary Bellman-Ford distance vector (d.v.) routing discipline [39]. More elaborate and practical routing schemes exist that are better equipped for e.g. disconnected operation, and with better combinations of scalability and stretch. However, for the present purpose, the d.v. scheme achieves its purpose. Consequently, a *routing table* t is a partial function associating to the OID’s o "known" to t a pair $t(o) = (u, n)$ where n is the minimum number of hops believed by t to be needed to reach the node hosting o from the current node, and where u is the next hop destination.

Routing Tables Routing tables support the following operations:

- Next hop lookup, $nxt(o,t) = \pi_1(t(o))$: In the context of a node $\mathsf{n}(u,t)$, $nxt(o,t)$ returns a neighbour u' of u to which, according to the current state of u, a message should be sent in order to eventually reach the destination o.

- Update, $upd(t,u,t')$: Updates t by incorporating the routing table t' belonging to a (neighbouring) node u. The update function is defined thus:

$$upd(t,u,t')(o) = \begin{cases} \perp & \text{if } o \notin dom(t) \cup dom(t') \\ t(o) & \text{else, if } o \notin dom(t') \\ (u, \pi_2(t'(o))+1) & \text{else, if } o \notin dom(t) \\ (u, \pi_2(t'(o))+1) & \text{else, if } \pi_1(t'(o)) = u \\ (u, \pi_2(t'(o))+1) & \text{else, if } t'(o) < \pi_2(t(o)) - 1 \\ t(o) & \text{otherwise} \end{cases}$$

  If o is known to neither the current node or to u, the distance estimate to o from the current node is undefined. If it is known to the current node, but not to u, t:s information is unchanged. If it is known to u, but not to the current node, the estimate from the current node becomes 1 plus u:s estimate. Otherwise we may assume that u is known to both the current node and to u. If the minimal route follows the edge between the current node and u, u:s distance estimate plus one is the new distance estimate at the current node, regardless of whether the estimate is improves on the current estimate or not. Otherwise, if u:s estimate improves sufficiently on the estimate at the current node, u:s estimate is incremented and used at the current node. In other circumstances, the distance estimate at the current node is left unchanged.

- Registration, $reg(o,u,t)$: Returns the routing table t' obtained by registering o at u in t, i.e.

$$reg(o,u,t)(o') = \begin{cases} (u,0) & \text{if } o = o' \\ t(o') & \text{otherwise} \end{cases}$$

  The function reg is invoked only when u is the “current" node.

Links, Queues, and Messages Nodes are connected by directed edges, or *links*, of the form

$$\mathsf{l}(u,q,u') \ ,$$

where $u \in NID$ is the source NID, $u' \in NID$ is the sink NID, and where $q \in Q$ is the associated fifo message queue. Queue operations are standard: $enq(msg,q)$ enqueues the message msg onto the tail of q, $hd(q)$ returns the head of q, and $deq(q)$ returns the tail of the q, i.e. q with $hd(q)$ removed. If q is empty ($q = \varepsilon$) then $hd(q)$ and $deq(q)$ are both undefined.

Messages have one of the following three forms:

- call$(o, o', m, \mathbf{v})$: A remote call message originating from object o and addressed to object o', of method m, and with arguments $\mathbf{v}$.
- table(t): A routing table update message. The origin NID is implicit, as the message is dequeued from a link queue with explicit source NID.
- object(cn): An object migration message, where cn is an *object closure*, as explained below.

Call messages are said to be *object bound*, and table and object messages are said to be *node bound*. We define $dst(msg)$, the *destination* of msg to be o' for call messages, and $dst(msg) = \bot$ in the remaining two cases.

The Network Graph Nodes and links induce a directed graph structure $graph(cn)$ in the obvious way, by taking as vertices the NID's u and as edges pairs (u, u') for each link $\mathsf{l}(u, q, u')$. For this to make sense we impose some constraints that apply from now on, to all "global" configurations cn in the type 2 semantics.

1. Unique vertices: There is at most one container $\mathsf{n}(u', t) \in cn$ with $u' = u$
2. Unique edges: For each source-sink pair u, u' there is at most one link $\mathsf{l}(u, q, u')$, for some q
3. Edges connect vertices: If $\mathsf{l}(u, q, u') \in cn$ then $\mathsf{n}(u, t), \mathsf{n}(u', t') \in cn$ for some t, t'
4. Reflexivity: $graph(cn)$ is reflexive, i.e. if $\mathsf{n}(u, t) \in cn$ for some t then $\mathsf{l}(u, q, u) \in cn$ for some q
5. Symmetry: $graph(cn)$ is symmetric, i.e. if $\mathsf{l}(u, q, u') \in cn$ then $\mathsf{l}(u', q', u) \in cn$ for some q'
6. Connectedness: $graph(cn)$ is connected, i.e. if $\mathsf{n}(u, t), \mathsf{n}(u', t') \in cn$ then there is a path in $graph(cn)$ connecting vertices u and u'

Conditions 1 is essential for naming. Condition 2 is important in the present paper, as the present paper focuses on closed systems. Condition 3 simplifies communication but could be lifted in principle. Conditions 4 and 5 are non-essential, but helpful. Finally, condition 6 is essential for routing to stabilize, but many of the results below can be proved without it.

Objects and Tasks In the type 2 semantics *object containers* are now attached to a node u and have the shape

$$\mathsf{o}(o, a, u, q_{in}, q_{out})$$

where $o \in OID$, $a \in OEnv$ as before, and q_{in}, q_{out} is a pair of an ingoing and an outgoing fifo message queue. This object level buffering is not essential, as messages are already buffered at link level, but object level buffering allows a more elegant formalization. It is commonplace in actor languages to consider inbound queues only. Here we find it more elegant to allow an outgoing queue as well, although this is mainly a matter of taste. Tasks $\mathsf{t}(o,l,s)$ are unchanged from the type 1 semantics.

Object Closures Type 2 configurations are built from the four container types introduced above, nodes, links, objects and tasks. It remains to explain object closures. For an object message $\mathsf{object}(cn)$ to be valid, the configuration cn needs to be an *object closure* of the form

$$\mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l_1,s_1)\ \ldots\ \mathsf{t}(o,l_n,s_n)\ .$$

Specifically, if cn is any configuration then $clo(cn,o)$, the *closure* of object o with respect to cn, is the multiset of all type 2 containers of the form either $\mathsf{o}(o',a',u',q'_{in},q'_{out})$ or $\mathsf{t}(o',l',s')$ such that $o'=o$, and $objof(cn)$ is a partial function returning o if all type 2 containers in cn are either objects or tasks, with OID o.

Type 2 Runtime Syntax Reflecting the above description, the type 2 runtime syntax is presented in fig. 7. A pictorial representation of the type 2

$u \in NID$			Node identifier
$t \in RTable$	$=$	$OID \to NID \times \omega$	Routing table
$q \in Q$	$=$	Msg^*	Message queue
$obj \in Obj_2$	$::=$	$\mathsf{o}(o,a,u,q_{in},q_{out})$	Object
$nd \in Nd$	$::=$	$\mathsf{n}(u,t)$	Network node
$lnk \in Lnk$	$::=$	$\mathsf{l}(u,q,u')$	Network link
$ct \in Ct_2$	$::=$	$tsk \mid obj \mid nd \mid lnk$	Runtime container
$cn \in Cn_2$	$::=$	$ct_1 \ldots ct_n$	Configuration
$msg \in Msg$	$::=$	$\mathsf{call}(o,o',f,m,\mathbf{v}) \mid \mathsf{table}(t) \mid \mathsf{object}(cn)$	Message

Figure 7: Type 2 runtime syntax

runtime state is shown in fig. 8. As implicit above, configurations remain multisets, and we write, e.g., $obj \in cn$ if cn can be written as $obj\ cn'$ for some cn'. Tasks are unchanged from fig. 4. We write $\mathsf{t}(cn)$ for the multiset of tasks in cn, i.e. the multiset $\{tsk \mid \exists cn'.cn = tsk\ cn'\}$, and $\mathsf{o}(cn)$ for the multiset of objects in cn, similarly defined. We also write $\mathsf{m}(cn)$ for the multiset $\{msg \mid msg \preceq cn\}$. To avoid explosion of the notation we reuse symbols from the type 1 semantics as far as possible, and resolve them by context.

7 Type 2 Reductions

An important distinction between the standard semantics and the network semantics is the absence of binding. For the standard semantics, name binding plays an key role to avoid clashes between locally generated names.

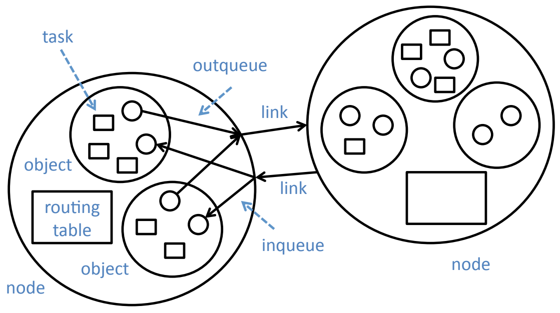

Figure 8: μABS-NET runtime state

However, in a language with NID's this device is no longer needed, as globally unique name can be guaranteed easily by augmenting names with their generating NID. Since all name generation takes place in the context of a given NID, we can simply assume an operation $newo(u)$ that return a new OID, which is globally fresh for the "current configuration". Another important point to note is that all transitions in the type 2 are *fully local*, in the sense that all operations applied, and all conditions determining whether or not a transition is enabled, can be fully determined by inspecting only one node and, possibly, the head of incoming link queues, alternatively by enqueuing messages to the tail of the outgoing queue.

First, the rules in fig. 5 apply with the following two minor modifications:

- Rule ctxt-2 is dropped as name binding is dropped from the type 2 runtime syntax
- Rule wfield is modified in the obvious way to read: If $x \in dom(a)$ then $\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to \mathsf{o}(o, a[\hat{e}(a, l)/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$

The remaining reduction rules are presented in fig. 9. The rules are naturally divided into subgroups:

- The rules t-send and t-rcv are concerned with the exchange of routing tables
- The three rules msg-send, msg-rcv and msg-route are used to manage message passing, i.e. reading a message from a link queue and transferring it to the appropriate object in-queue, and dually, reading a message from an out-queue and transferring it to the attached link queue. Finally, messages are routed to the next link, if the destination object does not reside at the current node. In rule msg-rcv note that the receiving node is not required to be present. This, however, will be enforced by the well-formedness condition later.

t-send: $\mathsf{n}(u,t) \vdash \mathsf{l}(u,q,u') \to \mathsf{l}(u, enq(\text{table}(t),q),u')$

t-rcv: If $hd(q) = \text{table}(t')$ then $\mathsf{l}(u',q,u)\ \mathsf{n}(u,t) \to \mathsf{l}(u', deq(q), u)\ \mathsf{n}(u, upd(t,u',t'))$

msg-send: If $hd(q_{out}) = msg$, $dst(msg) = o'$ and $nxt(o',t) = u'$ then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u,q,u')\ \mathsf{o}(o,a,u,q_{in},q_{out}) \to \mathsf{l}(u, enq(msg,q),u')\ \mathsf{o}(o,a,u,q_{in},deq(q_{out}))$

msg-rcv: If $hd(q) = msg$ and $dst(msg) = o$ then
$\mathsf{l}(u',q,u)\ \mathsf{o}(o,a,u,q_{in},q_{out}) \to \mathsf{l}(u',deq(q),u)\ \mathsf{o}(o,a,u,enq(msg,q_{in}),q_{out})$

msg-route: If $hd(q) = msg$, $dst(msg) = o$ and $nxt(o,t) = u'' \neq u$ then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u',q,u)\ \mathsf{l}(u,q',u'') \to \mathsf{l}(u',deq(q),u)\ \mathsf{l}(u,enq(msg,q'),u'')$

msg-delay-1: If $hd(q) = msg$, $dst(msg) = o$ and $nxt(o,t) \uparrow$ then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u',q,u)\ \mathsf{l}(u,q',u) \to \mathsf{l}(u',deq(q),u)\ \mathsf{l}(u,enq(msg,q'),u)$

msg-delay-2: If $hd(q_{out}) = msg$, $dst(msg) = o'$, and $nxt(o',t) \uparrow$ then
$\mathsf{n}(u,t) \vdash \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{l}(u,q,u) \to \mathsf{o}(o,a,u,q_{in},deq(q_{out}))\ \mathsf{l}(u,enq(msg,q),u)$

call-send: Let $o' = \hat{e_1}(a,l)$, $\mathbf{v} = \hat{\mathbf{e_2}}(a,l)$ in
$\mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,e_1!m(\mathbf{e_2});s) \to$
$\quad \mathsf{o}(o,a,u,q_{in},enq(\text{call}(o,o',m,\mathbf{v}),q_{out}))\ \mathsf{t}(o,l,s)$

call-rcv: If $hd(q_{in}) = \text{call}(o',o,m,\mathbf{v})$ then
$\mathsf{o}(o,a,u,q_{in},q_{out}) \to \mathsf{o}(o,a,u,deq(q_{in}),q_{out})\ \mathsf{t}(o,locals(o,m,\mathbf{v}),body(o,m))$

new-2: Let $o' = newo(u)$ in
$\mathsf{o}(o,a,u,q_{in},q_{out}) \vdash \mathsf{n}(u,t)\ \mathsf{t}(o,l,x = \mathbf{new}\ C(\vec{e});s) \to$
$\quad \mathsf{n}(u,reg(o',u,t))\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',init(C,\hat{\mathbf{e}}(a,l)),u,\varepsilon,\varepsilon)$

obj-send: Let $cn' = clo(cn,o)$ in
$\mathsf{n}(u,t)\ \mathsf{l}(u,q,u')\ cn \to \mathsf{n}(u,reg(o,u',t))\ \mathsf{l}(u,enq(\text{object}(cn'),q),u')\ (cn - cn')$

obj-rcv: If $hd(q) = \text{object}(cn')$ then
$\mathsf{l}(u',q,u)\ \mathsf{n}(u,t) \ \to \mathsf{l}(u',deq(q),u)\ \mathsf{n}(u,reg(objof(cn'),u,t))\ cn'$

Figure 9: Type 2 reduction rules

- The two rules msg-delay-1 and msg-delay-2 are used to handle the case where routing tables have not yet stabilized. For instance it may happen that updates to the routing tables have not yet caught up with object migration. In this case, a message may enter an out-queue without the hosting nodes routing table having information about the message's destination (rule msg-delay-2). Another case is where a node receives a message on a link without knowing where to forward it (rule msg-delay-1). This situation is particularly problematic as a blocked message may prevent routing table updates to reach the hosting node, thus causing deadlock. The solution we propose is to use the network self-loop as a buffer for temporarily unroutable messages.

- The rules call-send and call-rcv produce and consume call messages in a pretty obvious way.

- The rule new-2 handles object creation, including registration of the new object at the local node.

- The final two rules concern object migration. Of these, obj-send is a *global* rule in that it is not allowed to be used in subsequent applications of the ctxt-1 rule. In this way we can guarantee that only complete object closures are migrated. In rule obj-send, $cn - cn'$ is multiset difference.

We emphasize again that all of the above rules are strictly local and appeal only to mechanisms directly implementable at link level: Tests and simple datatype manipulations taking place at a single node, or accesses to a single nodes link layer interface. The "global" property appealed to above for the migration rules is merely a formal device to enable an elegant treatment of object closures.

The reduction rules can be optimized in several ways. For instance, object self-calls are always routed through the "network interface", i.e. the hosting nodes self-loop. This is not necessary. It would be possible to add a rule to directly spawn a handling task from a self call without affecting the results of the paper.

We note some elementary properties of the type 2 semantics.

Proposition 7.1. *Suppose that* $cn \to cn'$*.*

1. *If* $\mathsf{n}(u,t) \preceq cn$ *then* $\mathsf{n}(u,t') \preceq cn'$ *for some* t'
2. *If* $lnk = \mathsf{l}(u,q,u') \preceq cn$ *then* $\mathsf{l}(u,q',u') \preceq cn'$ *for some* q'
3. *If* $obj = \mathsf{o}(o,a,u,q_{in},q_{out}) \preceq cn$ *then there is an object*

$$obj' = \mathsf{o}(o',a',u',q'_{in},q'_{out}) \preceq cn'$$

(the derivative *of* obj *in* cn'*) such that* $o' = o$*,* $u' = u$*, and for all* x*, if* $a(x)\downarrow$ *then* $a'(x)\downarrow$*.*

Proof. By inspecting the rules. $\square$

We then turn to initial configurations. Let a program $\boldsymbol{CL}\{\mathtt{x}, s\}$ be given.

Definition 7.2 (Type 2 Initial Configuration)**.** A *type 2 initial configuration* has the shape

$$cn_{init} = cn_{graph}\ \mathsf{o}(main, \bot, u_{init}, \varepsilon, \varepsilon)\ \mathsf{t}(main, l_{init}, s)$$

where

- $\mathsf{o}(main, \bot)\ \mathsf{t}(main, l_{init}, s)$ is a type 1 initial configuration,
- cn_{graph} is a configuration consisting only of nodes and links, with empty link queues,
- u_{init} names a node $\mathsf{n}(u_{init}, t_{init})$ in cn_{graph},
- $t_{init}(main) = (u_{init}, 0)$, and $t_{init}(o) = \bot$ for $o \neq main$, and
- $t(o) = \bot$ for all routing tables $t \neq t_{init}$ and OID's o in cn_{init}.

8 Well-formedness

The well-formedness conditions need to be augmented somewhat for the type 2 semantics. We can observe first that the graph well-formedness conditions of the previous section are clearly preserved by all transitions.

Definition 8.1 (Type 2 Wellformedness)**.** A type 2 configuration cn is *type 2 wellformed* (WF2) if cn satisfies:

1. *OID uniqueness*: Suppose
$$\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}), \mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq cn$$
are distinct object occurrences. Then $o_1 \neq o_2$
2. *Object-Node existence*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \in cn$ then $\mathsf{n}(u, t) \in cn$ for some t.
3. *Task-Object Existence*: If $\mathsf{t}(o, l, s) \preceq cn$ then $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ for some a, u, q_{in}, q_{out}
4. *Object Existence*: Suppose $o \notin Ext$ occurs in cn. Then $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ for some a, u, q_{in}, q_{out}
5. *Object Nonexistence*: Suppose $o \in Ext$. The $\mathsf{o}(o, a, u, q_{in}, q_{out}) \not\preceq cn$ for any a, u, q_{in}, q_{out}
6. *Buffer cleanliness*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ and $msg \preceq q_{in}$ or $msg \preceq q_{out}$ then msg is object bound. Moreover, if $msg \preceq q_{in}$ then $dst(msg) = o$
7. *Local Routing Consistency, 1*: If $\mathsf{n}(u, t), \mathsf{o}(o, a, u, q_{in}, q_{out}) \in cn$ then $nxt(o, t) = (u, 0)$
8. *Local Routing Consistency, 2*: If $\mathsf{n}(u, t) \preceq cn$ and $\pi_1(nxt(o, t)) = u'$ then there is a link $\mathsf{l}(u, q, u') \preceq cn$

Most conditions are straightforward. For 8.1.6 observe that only object bound messages (for in-queues, with messages appropriately addressed) enter the object queues. Buffer cleanliness is needed to prevent the formation of contexts that are deadlocked because an in- or out-queue contains messages of the wrong type. For 8.1.7 the requirement should hold only when the object is *not* in transit (the object is in transit when it is contained in a message queue, i.e. when $\preceq$ holds, but not $\in$), as otherwise the object may be on the wire away from node u, and u:s routing table will then have been updated. This is not a concern in 8.1.8 as nodes and links never move. WF2 Preservation is very easily verified:

Lemma 8.2 (WF2 Preservation)**.** *If cn is type 2 well-formed and $cn \rightarrow_2 cn'$ then cn' is type 2 wellformed.* □

As above we note that then all type 2 reachable states are well-formed.

Corollary 8.3. *If cn is type 2 reachable then cn is fully type 2 wellformed.*

Proof. First check that initial configurations are type 2 wellformed and closed and then use lemma 8.2. □

9 Type 2 Barbed Equivalence

We next adapt the notion of barbed equivalence to the type 2 setting. The only difficulty is to define the type 2 correlate of the observation predicate. We take the point of view that an observation $obs = o!m(\mathbf{v})$ is enabled at a configuration cn if a corresponding call message call$(o', o, m, \mathbf{v})$ is located at the head of one of the object output queues in cn. More precisely, the type 2 observability predicate is $cn \downarrow obs$, holding if and only if cn has the following shape;

$$cn = cn' \ \mathsf{o}(o', a, u, q_{in}, q_{out}) \tag{2}$$

and $hd(q_{out})$ is defined and equal to call$(o', o, m, \mathbf{v})$.

There are other ways of defining the observability predicate that may be more natural. For instance one may attach external OID's to specific NID's and restrict observations to those NID's accordingly. It is also possible to add dedicated output channels to the model, and route external calls to those. None of these design choices have any effect on the subsequent results, however, but add significant notational overhead, particular in the latter case.

With the observation predicate set up, the weak observation predicate is derived as in section 5, and, as there, we define a *type 2 witness relation* as a relation that satisfies symmetry, reduction closure, and barb preservation. Thus:

Definition 9.1. Type 2 Barbed Equivalence Let $cn_1 \simeq_2 \ cn2$ if and only if $cn_1 \ \mathcal{R} \ cn_2$ for some type 2 witness relation $\mathcal{R}$.

In fact, for the purpose of this paper there in no real need to distinguish between the type 1 and type 2 equivalences, and hence we conflate the notions of witness relation and barbed equivalences, by letting the type of the configuration arguments be determined by the context, and use $\simeq$ as the generic notion.

10 Normal Forms

The goal is prove that if cn_1 and cn_2 are initial type 1 and type 2 configurations, respectively, for the same program, then $cn_1 \simeq cn_2$. The key to the proof is a normal form lemma for the type 2 semantics saying, roughly, that any well-formed type 2 configuration can be rewritten, using a subset of the rules as detailed below, into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, and where all objects have been moved to a single node. We prove this in two steps. First we prove a stabilization result, that non-self links can be emptied of messages and routing tables normalized to induce messaging paths with unit stretch. This allows the second normalization step to empty also object queues and migrate all objects to a single node. Once this is done we can prove correctness by exhibit a map representing each type 1 configuration as a canonical type 2 configuration, using normalization to help prove reduction preservation in both direction. Then only barb preservation is needed to complete the correctness argument.

10.1 Stabilization

We first show that each configuration can be rewritten using the transition rules into a form for which routing is stable and all queues are empty, except for *external* messages, i.e. messages msg addressed to an object $o \in Ext$. By well-formedness we then know that no object $\mathsf{o}(o', a', u', q'_{in}, q'_{out}) \preceq cn$ with $o' = o$ exists. In the context of a configuration cn call a *proper link* any link $\mathsf{l}(u, q, u')$ for which $u \neq u'$.

Definition 10.1 (Stable Routing, External Queued Messages)**.** Let cn be a well-formed type 2 configuration.

1. cn has *stable routing*, if for all $\mathsf{n}(u, t)$, $\mathsf{o}(o, a, u', q_{in}, q_{out}) \preceq cn$, if $nxt(o, t) = u''$ then there is a minimal length path from u to u' which visits u''
2. cn has *external link messages only*, if $\mathsf{l}(u, q, u') \in cn$ and $msg \preceq q$ implies $u = u'$ and msg is external.

The strategy for performing the rewriting is to first empty link queues as far as possible as we simultaneously exchange routing tables to converge to a configuration with stable routing. This first stage is accomplished using algorithm 1 in fig. 10 where we hide uses of ctxt-1 to allow the transition rules to be applied to arbitrary containers. Observe that we have no intention to use alg. 1 or any of the later algorithms in this section to do actual computing in the type 2 semantics. "Real" network computing using the type 2 semantics requires more sophisticated approaches. The algorithms considered here do not need to be effective or "local": We only need to exhibit *some* strategy for producing a configuration with the desired result, allowing us to prove the desired normal form results.

Proposition 10.2. *Algorithm 1 terminates.*

Proof. See appendix A. □

Write $\mathcal{A}_1(cn) \rightsquigarrow cn'$ if the configuration cn' is a possible result of applying algorithm 1 to cn. We then say that cn' is in *stable form*. Stable forms are almost unique, but not quite, since routing may stabilize in different ways, and since this (plus the generally nondeterministic scheduling of rules in alg. 1) may cause messages to enter object input queues at different times. Let $\mathsf{t}_1(cn) = \{tsk \mid tsk \preceq cn\}$ and let $\mathsf{o}_1(cn)$ be the multiset of object containers $ct = \mathsf{o}(o, a, u, q_{in}, q_{out})$ in cn such that either $ct \in \mathsf{o}(cn)$, or else $\mathsf{o}(o, a, u', q_{in}, q_{out})$ is in transit in cn from some u' to u (since then, after applying alg. 1, u will host the object). Finally, let $\mathsf{m}_1(cn)$ be the multiset of external messages in transit in cn, or of messages occurring in an object in- or out-queue.

Proposition 10.3. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$ then*

```
Algorithm 1: Stabilize routing and read internal link messages
Input Type 2 wellformed configuration $cn$ on a connected network graph
Output Configuration with stable routing and external link messages only
repeat
    Use t-send on each proper link in $cn$ to broadcast routing tables to all
    neighbours ;
    repeat
        Use t-rcv to dequeue one message on a link in $cn$
    until t-rcv no longer enabled ;
    Use msg-rcv, msg-route, msg-delay-1, obj-rcv to dequeue one message
    from each link, if possible
until link queues contain only external messages, and routing is stable
```

Figure 10: Algorithm 1– Stabilize routing and empty link queues of internal messages

1. $\mathit{graph}(cn) = \mathit{graph}(cn')$
2. cn' *has stable routing*
3. cn' *has external link messages only*
4. $\mathsf{t}(cn') = \mathsf{t}_1(cn)$
5. $\mathsf{o}(cn') = \mathsf{o}_1(cn)$
6. $\mathsf{m}(cn') = \mathsf{m}_1(cn)$

Proof. Property 1 and 2 are immediate. Property 3 and 4 can be read out of the termination proof. For the remaining three properties observe first that t_1, o_1, and m_1 are all invariant under the transitions used in algorithm 1. The equations follows by noting that only external messages (and so no object closures) are in transit in cn'. □

Prop. 10.3 shows the "almost uniqueness" property alluded to above. The normal form property suggested by prop. 10.3 motivates a notion of equivalence "up to stabilization" defined below.

Definition 10.4 ($\equiv_1$)**.**

1. Let $cn_1\ \mathcal{R}_1\ cn_2$ if and only if cn_1 and cn_2 are both type 2 well-formed, $\mathit{graph}(cn_1) = \mathit{graph}(cn_2)$, $\mathbf{z}_1 = \mathbf{z}_2$, $\mathsf{t}_1(cn_1) = \mathsf{t}_1(cn_2)$, $\mathsf{o}_1(cn_1) = \mathsf{o}_1(cn_2)$, and $\mathsf{m}_1(cn_1) = \mathsf{m}_1(cn_2)$.
2. Let $cn_1 \equiv_1 cn_2$ if there are cn_1', cn_2' such that

$$\mathcal{A}_1(cn_1) \rightsquigarrow cn_1'\ \mathcal{R}_1\ cn_2' \leftsquigarrow \mathcal{A}_1(cn_2)$$

Prop. 10.3 together with termination of $\mathcal{A}_1$ allows the existential quantifiers in def. 10.4.2 to be exchanged by universal ones.

Algorithm 2: Normalization

Input Type 2 well-formed configuration cn on a connected network graph
Output Configuration in type 2 normal form

```
fix a NID $u$ ;
run alg. 1 ;
repeat
  while some object queue is nonempty {
    use msg-send, msg-delay-2, call-rcv to dequeue one message from each
      nonempty object queue } ;
  while an object exists not located at $u$ {
    use obj-send to send the object towards $u$ } ;
    run alg. 1
until all objects are located at $u$ and queues contain only external messages
```

Figure 11: Algorithm 2 – Normalization

Corollary 10.5. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$ then $cn \equiv_1 cn'$*

Proof. We have $\mathcal{A}_1(cn) \rightsquigarrow cn' \ \mathcal{R}\ cn' \leftsquigarrow \mathcal{A}_1(cn')$. □

Lemma 10.6. *$\equiv_1$ is reduction closed*

Proof. See appendix A. □

Proposition 10.7. *$\equiv_1$ is a type 2 witness relation*

Proof. See appendix A. □

Corollary 10.8. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$ then $cn \simeq cn'$*

Proof. By prop. 10.7 and corollary 10.5. □

10.2 Normalization

When then turn to the second normalization step, to empty object queues and migrate all object closures to a central node. The normalization procedure is algorithm 2 shown in fig. 11. Let $\mathcal{A}_2(cn) \rightsquigarrow cn'$ if cn' is a possible result of applying algorithm 2 to cn. Initially a node u_0 is chosen towards which all objects will migrate during normalization. Normalization is performed in cycles, each cycle starting and ending in a stable configuration. In each cycle first object in- and out-queues are emptied. Then, objects not yet at u_0 are migrated one step toward u_0. Routing is not needed for this. It is sufficient to know that migration toward u_0 is possible.

Proposition 10.9. *Algorithm 2 terminates*

Proof. See appendix A. □

We then turn to normal forms and define first a couple of auxiliary operations. Let $\mathrm{t}_2(cn)$ be the multiset of method containers $tsk = \mathrm{t}(o, l, s)$ such that one of the following cases apply:

- tsk is a task container in cn.
- There is a message call$(o', o, m, \mathbf{v})$ in transit, $o \notin Ext$, $l = locals(o, m, \mathbf{v})$ and $s = body(o, m)$.

Let $\mathsf{o}_2(cn)$ be the multiset of object containers $\mathsf{o}(o, a, u, \varepsilon, \varepsilon)$ for which the following apply:

- $u = u_0$
- There is an object container $obj = \mathsf{o}(o, a', u', q_{in}, q_{out}) \preceq cn$
- $a'(x) = a(x)$ for all variables x

Also say that cn has *external messages only*, if object queues are empty and link queues in cn contain only external messages.

Definition 10.10 (Normal Form)**.** A well-formed configuration cn is in *normal form*, if

1. cn has stable routing
2. cn has external messages only
3. $\mathsf{t}(cn) = \mathsf{t}_2(cn)$
4. $\mathsf{o}(cn) = \mathsf{o}_2(cn)$
5. $\mathsf{m}(cn) = \mathsf{m}_1(cn)$

Proposition 10.11. *Suppose cn is well-formed. If $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then*

1. *cn' is in normal form*
2. *$graph(cn) = graph(cn')$*
3. *$\mathsf{t}_2(cn) = \mathsf{t}(cn')$*
4. *$\mathsf{o}_2(cn) = \mathsf{o}(cn')$*
5. *$\mathsf{m}_1(cn) = \mathsf{m}(cn')$*

Proof. See appendix A. □

Similar to prop. 10.4, prop. 10.11 justifies a notion of normal form equivalence as follows.

Definition 10.12 ($\equiv_2$)**.**
1. Let $cn_1\ \mathcal{R}_2\ cn_2$ if and only if cn_1 and cn_2 are both well-formed, $graph(cn_1) = graph(cn_2)$, $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$, $\mathsf{o}_2(cn_1) = \mathsf{o}_2(cn_2)$, and $\mathsf{m}_1(cn_1) = \mathsf{m}_1(cn_2)$.
2. Let $cn_1 \equiv_2 cn_2$ if and only if there are cn_1', cn_2' such that

$$\mathcal{A}_2(cn_1) \rightsquigarrow cn_1'\ \mathcal{R}_2\ cn_2' \leftsquigarrow \mathcal{A}_2(cn_2)$$

Clearly, $\equiv_2$ identifies more extended configurations than $\equiv_1$.

Corollary 10.13. $\equiv_1 \subseteq \equiv_2$

Proof. If $\mathsf{t}_1(cn_1) = \mathsf{t}_1(cn_2)$ then $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$ and similar for o_1 and o_2. The result follows. $\square$

We also obtain that normalization respects normal form equivalence.

Corollary 10.14. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then $cn \equiv_2 cn'$*

Proof. By prop. 10.11. $\square$

The proof of reduction closure follows that of lemma 10.6 quite closely.

Lemma 10.15. *$\equiv_2$ is reduction closed.*

Proof. See appendix A. $\square$

Proposition 10.16. *$\equiv_2$ is a type 2 witness relation*

Proof. Similar to the proof of prop. 10.7. $\square$

It follows that $cn_1 \equiv_2 cn_2$ implies $cn_1 \simeq cn_2$.

Corollary 10.17. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then $cn \simeq cn'$*

Proof. None of the rules used in alg. 2 affects the shape of the normal form. Thus, if $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then $cn \equiv_2 cn'$. But then $cn \simeq cn'$, by prop. 10.16. $\square$

11 Correctness

The goal is to prove soundness and full abstraction of the network semantics, i.e. that for any two type 1 configurations $\mathsf{bind}\ \mathbf{o}.cn$, $\mathsf{bind}\ \mathbf{o}'.cn'$ in standard form, $\mathsf{bind}\ \mathbf{o}.cn \simeq \mathsf{bind}\ \mathbf{o}'.cn'$ if and only if $down(cn_1) \simeq down(cn_2)$. However, since we have set up the semantics such that $\simeq$ applies without modification at both type 1 and type 2 levels it suffices to prove that $\mathsf{bind}\ \mathbf{o}.cn \simeq down(cn)$.

To accomplish this we represent each type 1 configuration as a type 2 configuration in normal form. We first fix an underlying graph represented as a well-formed type 2 configuration cn_{graph} and a distinguished UID u_0 in this graph, similar to the way initial configurations are defined in section 6. Thus, cn_{graph} consists of nodes and links only, each node u in cn_{graph} has the form (u, t), and each link has the form (u, ε, u'). The routing tables t are defined later. Defining a suitable representation map is a little cumbersome. A first complication is that names in the type 1 semantics (which includes the binder) need to be related to names in the type 2 semantics, which does not include the binder, but on the other hand has different generator functions (the function $newo$). For external names this is not a problem, but for bound names some form of name representation map is useful to connect the two types of names. Accordingly, we fix an injective *name representation map* rep, taking names o in the type 1 semantics to names $rep(o)$ in the type 2 semantics. For convenience we extend the name representation map rep to external names $o \in \mathit{Ext}$ by $rep(o) = o$, to arbitrary values by $rep(p) = p$, to task environments by $rep(l)(x) = rep(l(x))$ and similarly for object environments. The only slight complication in defining the mapping $down$ is that we need an operation to send a type 1 call container as a message in the type 2 semantics. This is done by the operation $send$ which sends a call container originating at o onto object o:s output queue as follows:

$$\begin{aligned} & send(\mathsf{c}(o, o', m, \mathbf{v}), \mathsf{o}(o, a, u, q_{in}, q_{out})\ cn) \\ & \quad = \quad \mathsf{o}(o, a, u, q_{in}, enq(\mathsf{call}(o, o', m, \mathbf{v}), q_{out}))\ cn \end{aligned} \tag{3}$$

We can then define the type 2 representation of the type 1 configuration $\mathsf{bind}\ \mathbf{o}.cn$ (leaving routing tables to be defined shortly) as the extended configuration $down(cn)(cn_{graph})$ where $down$ is defined by induction on the structure of cn as follows:

- $down(0)(cn) = cn$
- $down(cn_1\ cn_2) = down(cn_1) \circ down(cn_2)$
- $down(\mathsf{t}(o, l, s))(cn) = \mathsf{t}(rep(o), rep(l), s)\ cn$
- $down(\mathsf{o}(o, a))(cn) = \mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon)\ cn$
- $down(\mathsf{c}(o, o', m, \mathbf{v}))(cn) = send(\mathsf{c}(rep(o), rep(o'), m, \boldsymbol{rep}(\mathbf{v})), cn)$

In other words, we represent type 1 configurations by first assuming some underlying network graph, and then mapping the containers individually to type 2 level. The only detail remaining to be fixed above is the routing tables. For u_0 the initial routing table, t_0, needs to register all objects in cn_0, i.e.

$$t_0 = reg(g(o_0), u_0, reg(g(o_1), u_0, reg(\cdots, reg(g(o_m), u_0, \bot)) \cdots))$$

where $o_0, \ldots, o_m$ are the OID's in cn_0'. For nodes $\mathsf{n}(u, t)$ where $u \neq u_0$ we let t be determined by some stable routing. This is easily computed using alg. 1, and we leave out the details. This completes the definition of $down(cn)$.

Lemma 11.1 (Up and Down Property)**.** *Let* $\mathsf{bind}\ \mathbf{z}.cn$ *be type 1 well-formed in standard form.*

1. *If* $\mathsf{bind}\ \mathbf{z}.cn \to \mathsf{bind}\ \mathbf{z'}.cn'$ *then* $down(cn) \to^* \circ \simeq down(cn')$
2. *If* $down(cn) \to cn''$ *then for some* $\mathbf{z'}$*,* cn'*,* $\mathsf{bind}\ \mathbf{z}.cn \to^* \mathsf{bind}\ \mathbf{z'}.cn'$ *and* $cn'' \simeq down(cn')$

Proof. See appendix B. □

We can now prove the main result of this first part of the paper.

Theorem 11.2 (μABS Implementation Correctness)**.** *For all wellformed type 1 configurations* cn *on connected network graphs,* $cn \simeq down(cn)$

Proof. Define $\mathcal{R}$ by

$$\mathcal{R} = \{(cn, cn') \mid down(cn) \simeq cn'\} \tag{4}$$

We show that $\mathcal{R}$ is a witness relation.

First for reduction-closure in both directions: If $cn_1 \mathcal{R} cn_2$ then $down(cn_1) \simeq cn_2$. If $cn_1 \to cn_1'$ then by the Up and Down Property, 1, $down(cn_1) \to^* cn' \simeq down(cn_1')$. We get that $cn_2 \to^* cn_2'$ such that $cn' \simeq cn_2'$. But then $cn_1' \mathcal{R} cn_2'$. Conversely, if $cn_2 \to cn_2'$ then $down(cn_1) \to^* cn'$ and $cn' \simeq cn_2'$. By the Up and Down Property, $cn_1 \to^* cn_1'$ and $cn' \simeq down(cn_1')$. But then $cn_1' \mathcal{R} cn_2'$ as desired.

Barb Preservation is very direct, also in both directions: Assume $cn_1 \mathcal{R} cn_2$. Then $cn_1 \downarrow o!m(\mathbf{v})$ if and only if $down(cn_1) \Downarrow o!m(\mathbf{v})$ if and only if $cn_2 \Downarrow o!m(\mathbf{v})$. This completes the proof. □

12 Discussion

We have presented a sound and fully abstract semantics for a rudimentary object language, in terms a network-based execution model. Thanks in part to a novel explicit mixing of messaging and routing we are able to present the model at a level where it could in principle be implemented in a provably correct fashion directly on top of silicon, or integrated in a hypervisor such as Xen [3], assuming reliable link layer functionality only.

Soundness and full abstraction is a useful validation that the network semantics induces the same behaviour on μABS programs as the reference semantics. The network semantics, however, lacks a scheduler to determine e.g. when to migrate objects and how to schedule threads on single nodes. Such a scheduler will resolve nondeterministic choices left open in the network semantics presented here. Once such a scheduler is added, soundness and full abstraction is lost. We can, however, easily adapt the results presented in this paper to a notion of barbed *simulation*, obtained by instead of requiring preservation of observations and reductions in both directions, requiring preservation only in one. Then correctness for barbed simulation, that $down(cn)$ simulates bind $\mathbf{o}.cn$, is obtained as a corollary.

Substantial work has been going on in the HATS project on the ABS language [22] and its extensions, for instance towards software product lines [35]. Johnsen et al [23] suggests an extension of ABS with deployment components for resource management. We are mainly interested in the μABS language as an example. Essentially, however, our work is language independent, and we could apply the approach presented here to a version of core Erlang with minor changes only. Some details would be different, in particular the treatment of Erlang's pattern match-based message reception construct. The changes, however, would be local only, and so make little essential difference.

Much work has been done on object/component mobility in the π-calculus tradition [29], and on the implementation of high-level object or process-oriented languages in terms of more efficiently implementable low level calculi. In [36], following earlier work on Pict [31], Fournet's distributed join-calculus [16], and the JoCaml programming language [9], a compiler is implemented and proved correct for Nomadic Pict, a prototype language with very similar functionality to our μABS language: principally asynchronous message passing between named, location-oblivious processes. The target language extends Pierce and Turner's Pict language with synchronous local communication and asynchronous message passing between located processes. In comparison with [36] the use of name-based routing allows us to use barbed equivalence in place of coupled simulation [30] and as a consequence obtain a simpler correctness proof, due to the need for locking in the central forwarding server scheme used in [36]. JoCaml also uses forward chaining, along with an elaborate mechanism to collapse the forwarding chains. In the Klaim project [4] compilers were implemented and proved correct for several variants of the Klaim language, using the Linda tuple space communication model and a centralized name server to identify local tuple servers. The Oz kernel language [38] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [27] adds to this a notion of locality with separate failure and mobility semantics, but no real distribution or communication semantics is given (long distance communication is reduced to explicit manipulation of located agents, in the style of Ambient calculus [7]).

Our correctness proof uses reduction semantics and barbed equivalence. This is rather standard in the process algebra literature, cf. [15, 8, 18]. Both Sewell et al [36] and Fournet et al [17] use coupled simulation in order to handle problems related to preemptive choice. This complication does not arise in our work, whence barbed equivalence suffices. However, barbed equivalence is mostly useful for closed systems modeling where the stimuli to which an observed system is to be exposed must be given up front, as part of the initial configuration. A structural account based on some form of contextual equivalence [32], or on bisimulation equivalence along with a labelled transition semantics instead, would be more suitable. Work in this direction is currently going on. Replacing our reference reduction-based semantics with a labelled transition semantics is fairly straightforward. The bigger challenge is to develop a suitable structural account at the network level, allowing partially defined configurations to be composed.

Another direction for future work is to extend the μABS language. Experiments in this directions are going on. In [11] we extend μABS with futures (aka promises [28]) as placeholders for return values. This extension turns out to be not at all trivial. In other directions it is of interest to consider models with node power-on/power-off, in order to model system with adaptive power consumption, as well as various forms of node failure, along similar lines as [14]. The model can be used as a platform for language-based studies of load balancing and resource adaptation. We have extended the network semantics reported here to the full core ABS language [22] and implemented a multi-core simulation engine. This work is reported in [10]. There we show how the network semantics presented here can be split into a language interpreter layer and a language independent node controller layer, not unlike the meta-actors of [26], and we show how different resource allocation heuristics can be used to optimize object-to-node placement for different simple applications.

Further down the line it is of interest to examine the potential practical implications of our approach. This presents additional challenges, including garbage collection and buffer management, in particular as the network semantics we have presented above uses unbounded buffers. Also the routing scheme must be reconsidered. Distance vector routing suffers from well-known and fundamental scalability and security problems, and needs consideration in light of recent progress on compact routing (cf. [40, 37]).

References

[1] K. Arnold. *The Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[2] B. Awerbuch and D. Peleg. Online tracking of mobile users. *J. ACM*, 42(5):1021–1058, 1995.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[4] L. Bettini, V. Bono, R. D. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In *Global Computing: programming Environments, Languages, Security and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer-Verlag, 2003.

[5] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In R. Guérin, R. Govindan, and G. Minshall, editors, *SIGCOMM*, pages 265–276. ACM, 2005.

[6] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. Rofl: routing on flat labels. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 363–374, New York, NY, USA, 2006. ACM.

[7] L. Cardelli and A. D. Gordon. Mobile Ambients. *Theoretical Computer Science, vol. 240, no 1*, 2000.

[8] G. Castagna, J. Vitek, and F. Z. Nardelli. The Seal calculus. *Inf. Comput.*, 201(1), 2005.

[9] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for objective-caml. In *Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, ASAMA '99, pages 22–, Washington, DC, USA, 1999. IEEE Computer Society.

[10] M. Dam, A. Jafari, A. Lundblad, and K. Palmskog. ABS-NET: Fully decentralized runtime adaptation for distributed objects. Manuscript, 2013.

[11] M. Dam and K. Palmskog. Efficient and fully abstract routing of futures in object network overlays. Manuscript, 2013.

[12] M. J. Demmer and M. Herlihy. The arrow distributed directory protocol. In S. Kutten, editor, *DISC*, volume 1499 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 1998.

[13] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Softw., Pract. Exper.*, 21(8):757–785, 1991.

[14] J. Field and C. A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In J. Palsberg and M. Abadi, editors, *POPL*, pages 195–208. ACM, 2005.

[15] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proc. 23rd ACM Symp. Principles of Programming Languages (POPL)*, pages 372–385. ACM Press, 1996.

[16] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR '96, pages 406–421, London, UK, UK, 1996. Springer-Verlag.

[17] C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, TCS '00, pages 348–364, London, UK, UK, 2000. Springer-Verlag.

[18] A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. *Mathematical Structures in Computer Science*, 13(3):371–408, 2003.

[19] D. Havelka, C. Schulte, P. Brand, and S. Haridi. Thread-based mobility in oz. In *Proceedings of the Second international conference on Multiparadigm Programming in Mozart/Oz*, MOZ'04, pages 137–148, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments*

and technologies, CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[21] A. Jeffrey and J. Rathke. Contextual equivalence for higher-order pi-calculus revisited. *Logical Methods in Computer Science*, 1(1), 2005.

[22] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[23] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. A formal model of object mobility in resource-restricted deployment scenarios. In F. Arbab and P. Ölveczky, editors, *Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*, pages 187–. Springer-Verlag, 2012.

[24] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, Feb. 1988.

[25] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[26] Y. Kwon, S. Sundresh, K. Mechitov, and G. Agha. Actornet: an actor platform for wireless sensor networks. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *AAMAS*, pages 1297–1300. ACM, 2006.

[27] M. Lienhardt, A. Schmitt, and J.-B. Stefani. Oz/K: A kernel language for component-based open programming. In *GPCE'07: Proceedings of the 6th international conference on Generative Programming and Component Engineering*, pages 43–52, New York, NY, USA, 2007. ACM.

[28] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.

[29] R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput*, 100(1):1–40,41–77, 1992.

[30] J. Parrow and P. Sjödin. Designing a multiway synchronization protocol. *Computer Communications*, 19(14):1151–1160, 1996.

[31] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[32] A. M. Pitts. Howe's method for higher-order languages. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge Tracts in Theoretical Computer Science*, chapter 5, pages 197–232. Cambridge University Press, Nov. 2011.

[33] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[34] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, Jan. 2011.

[35] I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.

[36] P. Sewell, P. T. Wojciechowski, and A. Unyapoth. Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Transactions on Programming Languages and Systems*, 34, April 2010.

[37] A. Singla, P. B. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy. Scalable routing on flat names. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 20:1–20:12, New York, NY, USA, 2010. ACM.

[38] G. Smolka. The definition of kernel oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, 1995.

[39] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[40] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, pages 1–10, New York, NY, USA, 2001. ACM.

[41] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum. Algorithmic design of the globe wide-area location service. *Comput. J.*, 41(5):297–310, 1998.

Appendix A: Proofs for Section 10

Proposition 10.2 *Algorithm 1 terminates*

Proof. In each iteration of the outermost loop of alg. 1, exactly one message is enqueued on each proper link, and at least one message is dequeued (from all link queues), so the sum of messages in transit in link queues does not exceed its initial value. The rules msg-rcv, msg-delay-1, obj-rcv cause messages to leave the link queues, except for external messages, which are copied onto the self-loop queues. If the link queues have only routing table messages the algorithm terminates in that iteration. So if the algorithm fails to terminate it must be because msg-route is from some point n_0 onwards applied in each iteration of the outermost loop. From n_0 onwards, no messages other than table updates are delivered (to the receiving node, or to the receiving object). In particular, no object messages can be in transit on a link from that point onwards. We show that then routing tables must at some point stabilize. At point n_0 (as all other points) each node u has $t(o) = (u, 0)$ whenever o:s host is u, by def. 8.1.7. Let m_0 be the length of the largest link queue at the point from which no messages are delivered. After $n_0 + m_0 + 1$ iterations, each node u has received at least one table update from each of its neighbours u', and the last table update applied to u has $t(o) = 0$. As result, at point $n_0 + m_0 + 1$ each node u has $t(o) = (u', 1)$ whenever the host of o is u' and the minimal length path from u to u' has length 1. The entry of u:s routing table for o will not change from that point onwards. We say that those entries are *stable*. Proceeding, let m_1 be the length of the largest link queue at at point $n_0 + m_0 + 1$. After $n_0 + m_0 + 1 + m_1 + 1$ iterations each routing table entry with length 2 (or less) will be stable. In the limit each entry will be stable. It follows that algorithm 1 must terminate, since, once routing has stabilized, rule msg-route-ext can only be applied a finite number of times before the message will be delivered.

The only detail remaining to be checked is that a message can always be read from a link, but table and object messages can always be delivered, and call and future messages can also always be delivered, if nothing else to the self loop, in case the routing table has not yet been updated, or if the message is external and the destination object is not known to the routing table. This is the only case where msg-delay-1-ext is used, in fact. This completes the argument. $\square$

Lemma 10.6 $\equiv_1$ *is reduction closed*

Proof. Assume that $cn_1 \to cn_1'$ and $cn_1 \equiv_1 cn_2$. We find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \equiv_1 \mathbf{z}_2' : cn_2'$. The proof is by cases on the rewriting rule applied. The details are a bit messy, but straightforward. For rules not among call-send, call-rcv, if

$$\mathcal{A}_1(cn_1) \rightsquigarrow cn_{1,1} \; \mathcal{R}_1 \; cn_{2,1} \leftarrow\!\!\rightsquigarrow \mathcal{A}_1(cn_2) \quad (5)$$

and $\mathcal{A}_1(cn_1') \rightsquigarrow cn_{1,1}'$ then we obtain $cn_{1,1}' \; \mathcal{R}_1 \; cn_{1,1}$ by prop. 10.3 and since the correspondences between tasks, objects, and call and future messages are maintained in pre- and poststates. For the two remaining rules, we proceed:

call-send: We may assume that

$$cn_1 = cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) \quad (6)$$
$$cn_1' = cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},enq(msg,q_{out}))\ \mathsf{t}(o,l,s) \quad (7)$$

where $\mathbf{v}=\hat{\mathbf{e_2}}(a,l)$, $msg = \mathsf{call}(o,o',m,\mathbf{v})$, $o'=\hat{e_1}(a,l)$. We get

$$\mathcal{A}_1(cn_1) \rightsquigarrow cn_1''\ \mathcal{R}_1\ cn_2'' \leftsquigarrow \mathcal{A}_1(cn_2) \quad (8)$$

for some choice of cn_1'', etc. By prop. 10.3,

$$\mathsf{o}(o,a,u,q_{in},q_{out}), \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) \in cn_1'' \quad (9)$$

and hence, by the definition of $\mathcal{R}_1$,

$$\mathsf{o}(o,a,u,q_{in},q_{in},q_{out}), \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) \in cn_2'' \quad (10)$$

as well. But then it follows that the configuration cn_2 can mimick the call-send step by cn_1 by first stabilizing to cn_2'' and then performing the call-send step, obtaining cn_2'.

call-rcv: Can be proved by the same strategy. □

Proposition 10.7 $\equiv_1$ *is a type 2 witness relation*

Proof. Symmetry is immediate, and reduction closure follows by lemma 10.6. For barb preservation, from $cn_1 \equiv_1 cn_2$ we get

$$\mathcal{A}_1(cn_1) \rightsquigarrow cn_1'\ \mathcal{R}\ cn_2' \leftsquigarrow \mathcal{A}_1(cn_2)\ . \quad (11)$$

If $cn_1 \downarrow o!m(\mathbf{v})$ then cn_1 has the shape $cn_1''\ \mathsf{o}(o,a,u,q_{in},q_{out})$ and $hd(q_{out}) = \mathsf{call}(o',o,m,\mathbf{v})$. By prop. 10.3 and the details of alg. 1, since o is external and not routable, the message $\mathsf{call}(o',o,m,\mathbf{v})$ occurs in the self-loop queue on u. By the definition of $\equiv_1$, $\mathsf{call}(o',o,m,\mathbf{v})$ occurs in a self-loop of cn_2'. A few reductions exposes $\mathsf{call}(o',o,m,\mathbf{v})$ at the head of that queue, at configuration cn_2'', say. Then $cn_2' \rightarrow^* cn_2''$ and $\mathcal{A}_1(cn_2) \rightarrow^* cn_2'$ we obtain $cn_2 \Downarrow o!m(\mathbf{v})$ and the proof is complete. □

Proposition 10.9 *Algorithm 2 terminates*

Proof. Routing is stable after each run of alg. 1, and none of the rules applied in the first inner loop affect routing stability. Also, after the first run of alg. 1, links contain only external calls. Whenever an object out-queue is nonempty, one of msg-send or msg-delay-2 will be enabled. By Buffer Cleanliness, call-rcv will be applicable if the object in-queue is nonempty, decreasing in-queue size by one. Thus, when the first while loop is exited, object queues are empty. The second while terminates when all objects not yet at u_0 have been put on the wire. At the end of each outer loop, routing is stabilized and link queues emptied (except for external messages).

Once emptied, out-queues remain empty. In-queues may contain messages at the start of the second iteration, but after that, only external messages remain in either link or object queues, except for object closures, which are consumed once they reach u_0. $\Box$

Proposition 10.11 *Suppose cn is well-formed. If $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then*

1. *cn' is in normal form*
2. *$graph(cn) = graph(cn')$*
3. *$\mathsf{t}_2(cn) = \mathsf{t}(cn')$*
4. *$\mathsf{o}_2(cn) = \mathsf{o}(cn')$*
5. *$\mathsf{m}_1(cn) = \mathsf{m}(cn')$*

Proof. Property 10.11.2 follows from prop. 10.3.1.

For property 10.11.3 observe first that the function t_2 is invariant under transitions used in alg. 2. On termination of alg. 2 only external messages are in transit, and since no rule causes a task to be modified, 10.11.3 follows.

For 10.11.4 let $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \in \mathsf{o}(cn')$. We need to show that

$$\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \in \mathsf{o}_2(cn) \ .$$

By definition, $q_{in,2} = q_{out,2} = \varepsilon$. Also, $u_2 = u_0$. We know that there is an object container $\mathsf{o}(o, a', u', q_{in}, q_{out}) \preceq cn$, as there is a 1-1 correspondence between object containers in pre- and poststate for each transition used in alg. 2. We also know that $a'(x) = a_2(x)$ for all x.

For 10.11.5 the property holds as it does so already for alg. 1.

We finally need to prove 10.11.1. Property 10.10.1 is trivial, as each run of alg. 2 ends with a run of alg. 1, and alg. 1 ensures that cn' has stable routing. Property 10.10.2 holds since alg. 1 ensures external link messages only, and since on termination, alg. 2 ensures empty object queues. For 10.10.3 the result follows since only external messages are in transit in cn'. For 10.10.4, if $obj = \mathsf{o}(o, a, u, \varepsilon, \varepsilon)$ satisfies the properties defining o_2 above then, referring to those conditions, $u = u' = u_0$, $a' = a$, $q_{in} = \varepsilon = q_{out}$, and $obj \in cn'$ as needed to be shown. Finally, 10.10.5 holds since it does so already for alg. 1. $\Box$

Lemma 10.15 *$\equiv_2$ is reduction closed.*

Proof. Assume that $cn_1 \to cn_1'$ and $cn_1 \equiv_2 cn_2$. We find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \equiv_2 cn_2'$. As above the proof is by cases on the rewrite rule. We can assume that cn_2 is in normal form, by 12 and transitivity of $\equiv_2$. For rules that do not affect $\mathsf{t}_2(cn_1)$, $\mathsf{o}_2(cn_1)$, or $\mathsf{m}_1(cn_1)$ the result is trivial. Rules in fig. 5 commute directly, i.e. the same rule applied to cn_1 can be applied to cn_2, in the same way. This follows since cn_2 is in normal form. Rules such as msg-send, msg-rcv that ship around messages between object and link queues are also very easy to prove, by reference to prop. 10.11. For the remaining cases:

call-send: We may assume that

$$\begin{aligned} cn_1 &= cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) && (12)\\ cn'_1 &= cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},enq(msg,q_{out}))\ \mathsf{t}(o,l,s) && (13)\end{aligned}$$

where $\mathbf{v} = \hat{\mathbf{e_2}}(a,l)$, $msg = \mathsf{call}(o,o',m,\mathbf{v})$, $o' = \hat{e_1}(a,l)$, and $o' \notin Ext$. By 10.11, since cn_2 is in normal form, we obtain

$$\mathcal{A}_2(cn_1) \rightsquigarrow cn''_1\ \mathcal{R}_2\ cn_2 \tag{14}$$

for some choice of cn''_1. By prop. 10.11 we obtain that

$$cn_2 = cn_{2,1}\ \mathsf{o}(o,a',u,\varepsilon,\varepsilon\ \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) \tag{15}$$

where $a'(x) = a(x)$ for variables x. It follows that cn'_2 can be chosen so that

$$cn'_2 = cn_{2,1}\ \mathsf{o}(o,a',u,\varepsilon,enq(msg,\varepsilon))\ \mathsf{t}(o,l,s), \tag{16}$$

and

$$\mathcal{A}_2(cn'_1) \rightsquigarrow cn_{1,3} \tag{17}$$

with $cn_{1,3} = cn_{1,4}\ \mathsf{o}(o,a',u,\varepsilon,enq(msg,\varepsilon))\ \mathsf{t}(o,l,s)$ such that $cn_{1,4}\ \mathcal{R}_2\ cn_{2,1}$. It follows that $cn_2 \to cn'_2$ and $cn'_1 \equiv_2 cn'_2$, as desired.

call-rcv: In this case we get

$$\begin{aligned} cn_1 &= cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},q_{out}) && (18)\\ cn'_1 &= cn_{1,1}\ \mathsf{o}(o,a,u,deq(q_{in}),q_{out})\ \mathsf{t}(o,locals(o,m,\mathbf{v}),body(o,m)) && (19)\end{aligned}$$

where $hd(q_{in}) = \mathsf{call}(o',o,m,\mathbf{v})$. Again using prop. 10.11 with cn_2 in normal form we get that

$$\mathcal{A}_2(cn_1) \rightsquigarrow cn''_1\ \mathcal{R}_2\ cn_2 \tag{20}$$

for some choice of cn''_1, and cn''_1 can be written as

$$cn''_1 = cn'''_1\ \mathsf{o}(o,a',u,\varepsilon,\varepsilon)\ \mathsf{t}(o,locals(o,m,\mathbf{av}),body(o,m)) \tag{21}$$

where a' is as in the previous case. Now using prop. 10.11 we obtain

$$cn_2 = cn_{2,1}\ \mathsf{o}(o,a',u,\varepsilon,\varepsilon)\ \mathsf{t}(o,locals(o,m,\boldsymbol{av}),body(o,m))\ , \tag{22}$$

and $cn''_1\ \mathcal{R}_2\ cn_2$, completing the case.

The remaining cases ret-2, get-2, new-2, and the object migration rules are proved in a similar fashion as the above. □

Appendix B: Proofs for Section 11

Lemma 11.1 (Up and Down Property). *Let* bind $\mathbf{z}.cn$ *be type 1 well-formed in standard form.*

1. *If* $\mathsf{bind}\ \mathbf{z}.cn \to \mathsf{bind}\ \mathbf{z'}.cn'$ *then* $down(cn) \to^* \circ \simeq down(cn')$

2. *If* $down(cn) \to cn''$ *then for some* $\mathbf{z'}$*,* cn'*,* $\mathsf{bind}\ \mathbf{z}.cn \to^* \mathsf{bind}\ \mathbf{z'}.cn'$ *and* $cn'' \simeq down(cn')$

Proof. 1. First note that each transition in fig. 5 immediately translates into a corresponding transition at type 2 level, and moreover, the resulting type 2 configuration is in normal form. For the remaining transitions we proceed by cases:

call: As cn is type 1 well-formed we can write cn in standard form as

$$cn = \mathsf{bind}\ \mathbf{z}.cn_1\ \mathsf{o}(o,a)\ \mathsf{o}(o',a')\ \mathsf{t}(o,l,x = e_1!m(\mathbf{e_2});s)\ , \tag{23}$$

and cn' as

$$\begin{aligned} cn' \quad = \quad & \mathsf{bind}\ \mathbf{z}.\mathsf{bind}\ f.cn_1\ \mathsf{o}(o,a)\ \mathsf{o}(o',a')\ \mathsf{t}(o,l,s) \\ & \qquad \mathsf{t}(o', \mathit{locals}(o',m,\hat{\mathbf{e_2}}(\mathbf{a},\mathbf{l})), \mathit{body}(o',m)) \end{aligned}$$

where $o' = \hat{e_1}(a,l)$. Let $\mathbf{v} = \hat{\mathbf{e_2})}(\mathbf{a},\mathbf{l})$. Fix cn_{graph} and u_0 as above. We get

$$\begin{aligned}
down(cn) \quad & = \quad down(cn_1)(cn_{graph})\ \mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon) \\
& \qquad \mathsf{o}(rep(o'), rep(a'), u_0, \varepsilon, \varepsilon) \\
& \qquad \mathsf{t}(rep(o), rep(l), x = e_1!m(\mathbf{e_2}); s) \\
& \to \quad down(cn_1)(cn_{graph})\ \mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \\
& \qquad\quad enq(\mathsf{call}(rep(o'), m, rep(\mathbf{v})), \varepsilon)) \\
& \qquad \mathsf{o}(rep(o'), rep(a'), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(rep(o), rep(l), s) \\
& \to^* \circ \simeq \quad down(cn_1)(cn_{graph}) \\
& \qquad \mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon) \\
& \qquad \mathsf{o}(rep(o'), rep(a'), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(rep(o), rep(l), s) \\
& \qquad \mathsf{t}(rep(o'), \mathit{locals}(rep(o'), m, \hat{\mathbf{e_2}}(\boldsymbol{rep}(\mathbf{a}), \boldsymbol{rep}(\mathbf{l})))), \\
& \qquad\quad \mathit{body}(o', m) \\
& \qquad \text{(By normalization and corollary 10.17)} \\
& = \quad down(cn_1)(cn_{graph})\ \mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon) \\
& \qquad \mathsf{o}(rep(o'), rep(a'), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(rep(o), rep(l), s) \\
& \qquad \mathsf{t}(rep(o'), rep(\mathit{locals}(o', m, \hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l}))), \mathit{body}(o', m)) \\
& = \quad down(cn')
\end{aligned}$$

call-ext: We can write cn and cn' as

$$\begin{aligned} cn \quad & = \quad \mathsf{bind}\ \mathbf{z}.cn_1\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = e_1!m(\mathbf{e_2});s) \\ cn' \quad & = \quad \mathsf{bind}\ \mathbf{z}.cn_1\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s)\ \mathsf{c}(o,o',m,\mathbf{v}) \end{aligned}$$

where $o' = \hat{e_1}(a,l) \in \mathit{Ext}$ and $\mathbf{v} = \hat{\mathbf{e_2}}\mathbf{(a,l)}$. We obtain:

$$\begin{aligned}
\mathit{down}(\mathit{cn}) &= \mathit{down}(\mathit{cn}_1)(\mathit{cn}_{\mathit{graph}})\ \mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u_0, \varepsilon, \varepsilon) \\
&\qquad \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l), x = e_1!m(\mathbf{e_2}); s) \\
&\to \mathit{down}(\mathit{cn}_1)(\mathit{cn}_{\mathit{graph}})\ \mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u_0, \varepsilon, \\
&\qquad\quad \mathit{enq}(\mathsf{call}(\mathit{rep}(o'), m, \mathit{rep}(\mathbf{v})), \varepsilon))\ \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l), s) \\
&= \mathit{down}(\mathit{cn}')
\end{aligned}$$

new: Let

$$\begin{aligned}
\mathit{cn} &= \mathsf{bind}\ \mathbf{z}.\mathit{cn}_1\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = \mathbf{new}C(\mathbf{e}); s) \\
\mathit{cn}' &= \mathsf{bind}\ \mathbf{z}.\mathsf{bind}\ o'.\mathit{cn}_1\ \mathsf{o}(o,a)\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', \mathit{init}(C, \vec{e}(a,l)))
\end{aligned}$$

We can write $\mathit{down}(\mathit{cn}_1)(\mathit{cn}_{\mathit{graph}})$ as $\mathit{cn}_1'\ \mathsf{n}(u_0,t_0)$. Also let $o'' = \mathit{newo}(u_0)$. Note that $o'' = \mathit{rep}(o')$. We calculate:

$$\begin{aligned}
&\mathit{down}(\mathit{cn}) \\
&\quad= \mathit{cn}_1'\ \mathsf{n}(u_0,t_0)\ \mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l), x = \mathbf{new}\ C(\mathbf{e}); s) \\
&\quad\to \mathit{cn}_1'\ \mathsf{n}(u_0, \mathit{reg}(\mathit{rep}(o'), u_0, t_0))\ \mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u_0, \varepsilon, \varepsilon) \\
&\qquad\quad \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l)[\mathit{rep}(o')/x], s)] \\
&\qquad\quad \mathsf{o}(\mathit{rep}(o'), \mathit{init}(C, \hat{\mathbf{e}}(\mathit{rep}(a), \mathit{rep}(l))), u_0, \varepsilon, \varepsilon) \\
&\quad= \mathit{cn}_1'\ \mathsf{n}(u_0, \mathit{reg}(o', u_0, t_0))\ \mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l[o'/x]), s)] \\
&\qquad\quad \mathsf{o}(\mathit{rep}(o'), \mathit{rep}(\mathit{init}(c, \hat{\mathbf{e}}(a,l))), u_0, \varepsilon, \varepsilon) \\
&\quad= \mathit{down}(\mathit{cn}')
\end{aligned}$$

2. We proceed by cases on the type 2 rule applied to derive $\mathit{down}(\mathit{cn}) \to \mathit{cn}''$. Rules in fig 5 are immediate since in those cases cn' can be found such that $\mathit{down}(\mathit{cn}') = \mathit{cn}''$. For rules among t-send, t-rcv, msg-send, msg-rcv, msg-route, msg-delay-1, msg-delay-2, call-rcv, obj-send, obj-rcv, using prop. 10.11, def. 10.12, and prop. 10.16 we can choose $\mathbf{o'} = \mathbf{o}$ and $\mathit{cn}' = \mathit{cn}$. For the remaining two rules:

call-send: We get that cn, $\mathit{down}(\mathit{cn})$ and cn'' have the shapes

$$\begin{aligned}
\mathit{cn} &= \mathit{cn}_1\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,e_1!m(\mathbf{e_2}); s) \\
\mathit{down}(\mathit{cn}) &= \mathit{down}(\mathit{cn}_1)(\mathit{cn}_{\mathit{graph}})\ \mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u, \varepsilon, \varepsilon) \\
&\qquad \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l), e_1!m(\mathbf{e_2}); s) \\
\mathit{cn}'' &= \mathit{down}(\mathit{cn}_1)(\mathit{cn}_{\mathit{graph}})\ \mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u, \varepsilon, \\
&\qquad \mathit{enq}(\mathsf{call}(\mathit{rep}(o), \mathit{rep}(o'), m, \boldsymbol{rep}(\mathbf{v})), \varepsilon))\ \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l), s)
\end{aligned}$$

where $\hat{e_1}(a,l) = o'$ and $\hat{\mathbf{e_2}}\mathbf{(a,l)} = \mathbf{v}$. We get two cases depending on whether $o' \in \mathit{Ext}$ or not. Suppose first the latter. Then cn_1 can be written as $\mathit{cn}_1'\ \mathsf{o}(o',a')$, and we can pick

$$\mathit{cn}' = \mathit{cn}_1'\ \mathsf{o}(o,a)\ \mathsf{o}(o',a')\ \mathsf{t}(o,l,s)\ \mathsf{t}(o', \mathit{locals}(o',m,\mathbf{v}), \mathit{body}(o',m))$$

We then use prop. 10.11 to conclude that $down(cn') \simeq cn''$, as desired. In case $o' \in Ext$ we get instead

$$cn' \quad = \quad cn_1\ \mathsf{o}(o, a)\ \mathsf{t}(o, l, s)\ \mathsf{c}(o, o', m, \mathbf{v})$$

and as above, $down(cn_1) = cn''$

new-2: We find cn, $down(cn)$ and cn'' as above of the shapes

$$\begin{aligned} cn &= cn_1\ \mathsf{o}(o, a)\ \mathsf{t}(o, l, x = \mathbf{new} C(\mathbf{e}); s) \\ down(cn) &= cn'_1\ \mathsf{n}(u_0, t_0)\ \mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon) \\ &\quad \mathsf{t}(rep(o), rep(l), x = \mathbf{new}\ C(\mathbf{e}); s) \\ cn'' &= cn'_1\ \mathsf{n}(u_0, reg(rep(o'), u_0, t_0))\ \mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon) \\ &\quad\quad \mathsf{t}(rep(o), rep(l)[rep(o')/x], s)] \\ &\quad\quad \mathsf{o}(rep(o'), init(C, \mathbf{v}), u_0, \varepsilon, \varepsilon) \end{aligned}$$

where $o' = newo(u_0)$, $\hat{\mathbf{e}}(\mathbf{a}, \mathbf{l}) = \mathbf{v}$, and $down(cn_1)(cn_{graph}) = cn'_1\ \mathsf{n}(u_0, t_0)$. We pick

$$cn' \quad = \quad cn_1\ \mathsf{o}(o, a)\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', init(C, \mathbf{v}))$$

and get $down(cn') = cn''$. □