Efficient and Fully Abstract Routing of Futures in Object Network Overlays

Mads Dam and Karl Palmskog
School of Computer Science and Communication
KTH Royal Institute of Technology
{mfd,palmskog}@kth.se

Abstract

In distributed object systems, it is desirable to enable migration of objects between locations, e.g., in order to support efficient resource allocation. Existing approaches build complex routing infrastructures to handle object-to-object communication, typically on top of IP, using, e.g., message forwarding chains or centralized object location servers. These solutions are costly and problematic in terms of efficiency, overhead, and correctness. We show how location independent routing can be used to implement object overlays with complex messaging behavior in a sound, fully abstract, and efficient way, on top of an abstract network of processing nodes connected point-to-point by asynchronous channels. We consider a distributed object language with futures, essentially lazy return values. Futures are challenging in this context due to the strong global consistency requirements they impose. The key conclusion is that execution in a decentralized, asynchronous network can preserve the standard, network-oblivious behavior of objects with futures, in the sense of contextual equivalence. To the best of our knowledge, this is the first such result in the literature. We also believe the proposed execution model may be of interest in its own right in the context of large-scale distributed computing.

1 Introduction

The ability to transparently and efficiently relocate objects between processing nodes is a basic prerequisite for many tasks in large-scale distributed systems, including tasks such as load balancing, resource allocation, and management. By freeing applications from the burden of resource management, they can be made simpler, more resilient, and easier to manage, resulting in a lower cost for development, operation and management.

The key problem is how to efficiently handle object and task mobility. Since object locations change dynamically in a mobile setting, some form of application-level routing is needed for inter-object messages to reach their destinations. Various approaches have been considered in the literature; Sewell et al. [36] provide a comprehensive survey. One common imple-
mentation strategy is to use some form of centralized, replicated, or decentralized object location register, either for forwarding or for address lookup [1, 13, 17, 36]. This type of solution requires some form of synchronization to keep registers consistent with physical locations, or else it needs to resort to some form of message relaying, or forwarding. Forwarding by itself is another main implementation strategy used in, e.g., the Emerald system [25], or in more recent systems like JoCaml [10]. Other solutions exist, such as broadcast or multicast search, that are useful for recovery or for service discovery, but hardly efficient as general purpose routing devices in large systems.

In general, we consider a mechanism for object mobility with the following properties desirable:

**Low stretch** In stable state, the ratio between actual and optimal route lengths (costs) should be small.

**Compactness** The space required at each node for storing route information should be small (sublinear in the number of destinations).

**Self-stabilization** Even when started in a transient state, computations should proceed correctly, and converge to a stable state. Observe that this precludes the use of locks.

**Decentralization** To enable scaling to large networks with many objects and tasks, routes and next-hop destinations should be computed in a decentralized fashion, at the individual nodes, and not rely on a centralized facility.

Existing solutions are quite far from meeting these requirements: Location registers (centralized or decentralized) and pointer forwarding regimes both preclude low stretch, and the use of locks precludes self-stabilization.

In earlier work [11], we suggest that the root of the difficulties lies in a fundamental mismatch between the information used for search and identification (typically, object identifiers, OIDs), and the information used for routing, namely, host identifiers, typically IP addresses. If we were to route messages not to the destination location, but instead to the destination object, it should be possible to build object network overlays which much better fit the desiderata laid out above. In previous work [11], we show that this indeed appears to be true (even if the problem of compactness is left for future investigation). The key idea is to use a form of location independent (also known as flat, or name independent) routing [2, 20, 21] that allows messages (RPCs) to be routed directly to the called object, independently of the physical node on which that object is currently executing. Using location independent routing, a lot of the overhead and performance constraints associated with object mobility can be eliminated, including latency and bandwidth overhead due to looking up, querying, updating, and locking object location databases, and overhead due to increased traffic for, e.g., message forwarding.

The language considered previously [11] allows to define a collection of objects communicating by asynchronous RPC, and thus its functionality
is not much different from a core version of Erlang [4], or the Nomadic Pict language studied by Sewell et al. [36]. The question we raise is how program behavior is affected by being run in the networked model, as compared with a more standard, network-oblivious “reference” semantics given in the style of rewriting logic [9]. This comparison is of interest, since the reference semantics is given at a high level of abstraction and ignores almost all aspects of physical distribution, such as location, routing, and message passing. We show that, with a maximally nondeterministic network-aware semantics, and in the sense of contextual equivalence [30], programs exhibit the same behavior in both semantics.

Messaging in our earlier work is very simple. The implicit channel abstraction used in the reference semantics is essentially that of a reliable, unordered communication channel. Messages (method calls) are sent according to the program order, but the order in which they are acted upon is arbitrary. Soundness and full abstraction for the network-aware semantics is therefore an interesting and useful observation, since it allows many conclusions made at the level of abstract program behavior to transfer to a networked realization.

In this paper, we address the question of how sensitive these results are to the type of communication taking place at the abstract level. The overlays considered in our earlier work allow only one type of message, with modest requirements on global consistency. It is of interest to examine also languages allowing more complex communication behavior for objects. To this end, we define the richer language mABS, corresponding essentially to a fragment of the ABS (Abstract Behavioral Specification) language core [23], developed in the EU FP7 HATS project. We show that the conclusions of our previous work remain valid, but with more involved constructions. The extensions result in much more complex object overlays involving futures [6, 12, 14, 27, 28, 39], in effect placeholders for remote method return values, that can be shared among objects, but whose eventual instantiated values need to be kept consistent and propagated correctly to all objects that need them. Future variables are used extensively in many concurrent and distributed high-level languages, libraries, and models, including Java, .NET, Scheme, Concurrent LISP, and Oz, to name just a few. Many versions of futures exist in the literature. Our work uses futures as placeholders for forthcoming computational results, as do Caromel et al. [5] and de Boer et al. [12]. Other models exists, such as the concurrent constraint store model of, e.g., Oz [28, 37].

Futures need a messaging infrastructure to propagate instantiations. Consider a remote method call \( x = \text{obj!m(args)} \). The effect of the call is the creation of two items:

1. A remote thread evaluating method \( m \) with the arguments \( \text{args} \) in \( \text{obj} \).
2. A future that becomes assigned to the variable \( x \). The future is initially uninstantiated, but is intended to become instantiated after the remote call has returned.

This functionality allows long-running tasks to be offloaded to a remote
thread with the main thread proceeding with other tasks. When the return value is eventually needed, the calling thread can request it by performing a `get` operation on the future. If `x` is uninstantiated, this causes the evaluation to block.

One problem is that first-class futures [5], which we employ, can be transmitted as arguments between threads. If `y` is a future variable occurring in `args`, there must be some means for the value eventually assigned to `y` to find its way to the remote thread computing `obj!m(args)`, either by forwarding the value after it becomes available, or by the remote thread querying either the caller or some centralized lookup server for the value of `y`, if and when it is needed. This creates very similar problems to those arising from object migration. Thus, it would seem likely that location independent routing could be useful for propagation of values for futures, and as we show in this paper, indeed this is so. In the case of futures, however, the problems are aggravated: In order for the network-aware implementation to be correct (sound and fully abstract) we must be able to show that future assignments are unique and propagate correctly to all objects needing the assignment, without resorting to solutions that are overly inefficient such as flooding.

Many strategies for future propagation exist in the literature [18, 31]. In this work, we use what Henrio et al. [18] refer to as an eager forward-based strategy, where assignments are propagated along the flow of futures as soon as they are instantiated. Other propagation strategies exist, including strategies that use various forms of location registers, and lazy strategies which request futures only as needed. Either approach may benefit from the use of location independent routing. However, our chosen strategy is particularly suited for decentralized networks, since it has lower propensity of overloading any particular node when object-node allocations are balanced [18].

Our main result is to show that, with full nondeterminism, the abstract, network-oblivious semantics and the network-aware semantics with futures implemented through eager forwarding correspond in the sense of contextual equivalence. To the best of our knowledge, this is the first such result in the literature, and is interesting in itself, as it shows that the network-aware semantics captures the abstract behavior very accurately. Also, it follows that, for the case when a scheduler is added (pruning some execution branches), a similar correspondence holds, but now for barbed simulation instead of barbed bisimulation.

The proof of the main result uses a normal form construction in two stages. First, we show that each well-formed configuration in the network-aware semantics can be rewritten into an equivalent form with optimal routes. The second stage of the normalization procedure then continues rewriting to a form where, in addition, all messages that can be delivered are delivered, and where all objects are migrated to some central node. Correctness of the normalization procedure essentially gives a Church-Rosser like property—that transitions in the network-aware semantics commute with normalization. Normalization brings configurations in
the network-aware semantics close to the form of the reference semantics, and this, then, allows the proof to be completed.

The paper is organized as follows: In Section 3, we first introduce the mABS language syntax, and the network-oblivious reference (type 1) semantics of mABS is given in Section 4. In Section 5, we present type 1 contextual equivalence, i.e., the notion of contextual equivalence adapted to the reference semantics. Then, in Section 6, we turn to the network-aware (type 2) semantics and present the runtime syntax and the reduction rules. We proceed by detailing the well-formedness conditions for the network-aware semantics in Section 7 and adapt contextual equivalence to this semantics in Section 8. We then present the normal-form construction in Section 9, and complete the correctness proof in Section 10. In Section 11, we discuss scheduling, and finally in Section 12, we conclude. Long proofs have been deferred to appendices.

2 Notation

We sometimes use a vectorized notation to abbreviate sequences, for compactness. Thus, \( \pi \) abbreviates a sequence \( x_1, \ldots, x_n \), possibly empty, and \( x_0, \pi \) 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 a 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 CPOs) \( A \) and \( B \) is \( A \times B \) with pairing \( (a, b) \) and projections \( \pi_1 \) and \( \pi_2 \).

3 The mABS Language

We define mABS, short for milli-ABS, a small, distributed, object-based language with asynchronous calls and futures. Its syntax is depicted in Figure 1. The mABS language is an extension of the language \( \mu \)ABS (micro-ABS) of message-passing processes introduced in earlier work \([11]\) with futures used as placeholders for method return values. The language is fairly self-explanatory. A program is a sequence of class definitions, appended with a set of variables \( \pi \) and a “main” statement \( s \), which can use those variables to set up an initial collection of objects. The class hierarchy is flat and fixed. Classes have parameters \( \pi \), local variable declarations \( \gamma \), and methods \( \Gamma \). Methods have parameters \( \pi \), local variable declarations \( \gamma \) and a statement body \( s \). For simplicity, we assume that variables have unique declarations. Expression syntax is left open, but is assumed to include the constant \texttt{self}. We require that expressions are side-effect free. We omit types from the presentation. Types could be added, but they would not affect the results of the paper in any significant way. Statements include standard sequential control structures, and a minimal set of constructs for
\[ x, y \in \text{Var} \quad e \in \text{Exp} \quad C, m \in \text{SID} \]

- \(P ::= \text{CL}(\pi, s)\) Program
- \(CL ::= \text{class }C(\pi)\{\gamma, M\}\) Class definition
- \(M ::= m(\pi)\{\gamma, s\}\) Method definition
- \(s ::= s_1; s_2 \mid x = \text{rhs} \mid \text{skip} \mid \text{while } e \{s\}\) Statement
- \(rhs ::= e \mid \text{new }C(\pi) \mid e!m(\pi) \mid e.get\) Right-hand side

Figure 1: mABS abstract syntax

class Server() { ,
serve(x) { s1, s2, f1, f2, r1, r2,
if small(x) {
    return process(x)
} else {
    s1 = \text{new Server}();
    s2 = \text{new Server}();
    f1 = s1!serve(hi(x));
    f2 = s2!serve(lo(x));
    r1 = f1.get();
    r2 = f2.get();
    return combine(r1, r2)
}
}
}

{, s, f, r,
    s = \text{new Server}();
    f = s!serve(1537);
    r = f.get
}

Figure 2: mABS code sample

asynchronous method invocation, object creation, and retrieval of values associated with futures (get statements).

**Example 3.1.** Assume that \(\text{combine}((\text{hi}(v)), (\text{lo}(v))) = \text{process}(v)\) for integers \(v\). In the class Server in the program in Figure 2, the method serve returns immediately if its argument is small. Otherwise, two new servers are spawned, and the upper and lower tranches delegated to those respective servers. The results are then retrieved, combined, and returned. In the main block, a call to serve on a server object results in a future, stored in the variable \(f\), which is then used to retrieve the actual result, stored in the variable \(r\). The original call spawns more server objects, which, in a network-aware implementation, can move to other nodes to balance load.

## 4 Reference Semantics

We first present an abstract reference semantics for mABS in the style of rewriting logic. The presentation follows our earlier work [11] quite closely. We use the abstract semantics for comparison with the more con-
crete network-aware semantics, which we present later. The semantics uses a reduction relation \( cn \rightarrow cn' \) where \( cn \) and \( cn' \) are configurations, as determined by the runtime syntax in Figure 3. Later on, we introduce different configurations and transition relations, and so use index 1, or refer to, e.g., configurations of “type 1” for this first semantics when we need to disambiguate. With respect to the runtime syntax, \( \subseteq \) is the sub-term relation, and we use disjoint, denumerable sets of object identifiers \( o \in OID \), future identifiers \( f \in FID \), and primitive values \( p \in PVal \). Values \( v \) are either primitive values, OIDs, or FIDs. Lifted values are ranged over by \( v_\bot \in Val_\bot \), and we use \( \sqsubseteq \) for the associated standard partial ordering. We often refer to OIDs and FIDs as names, and subject them binding using bind, which is reminiscent of the binder in the \( \pi \)-calculus \[32\].

Later, in the type 2 semantics, this type of explicit binding is dropped. We use \( z \) as a generic name variable, and assume throughout that names are uniquely bound. The free names of a configuration \( cn \) is the set \( fn(cn) \), and \( OID(cn) = \{ o \mid \exists a. o(a,a) \sqsubseteq cn \} \) is the set of OIDs of objects occurring in \( cn \). Similarly, \( FID(cn) = \{ f \mid \exists v_\bot. f(f,v_\bot) \sqsubseteq cn \} \) is the set of future identifiers in \( cn \). Standard alpha congruence applies to name binding.

Configurations are “\( \pi \)-scoped” multisets of containers of which there are four types, namely, tasks, objects, futures, and calls. Configuration juxtaposition is assumed to be commutative and associative with unit 0. In addition we assume the standard structural identities \( \text{bind } z.0 = 0 \) and \( \text{bind } z.(cn_1 \text{ bind } z.cn_2) = (\text{bind } z.cn_1) \text{ bind } z.cn_2 \) when \( z \notin fn(cn_2) \). We often use a vectorized notation \( \text{bind } \Xi.cn \) as abbreviation, letting \( \text{bind } \varepsilon.cn = cn \) where \( \varepsilon \) is the empty sequence. The structural identities then allow us to rewrite each configuration into a standard form \( \text{bind } \Xi.cn \) such that each member of \( \Xi \) occurs free in \( cn \), and \( cn \) has no occurrences of the binding operator bind. We use standard forms frequently.

Tasks are used for method body elaboration, and futures are used as centralized stores for assignments to future variables. Task and object envi-

---

\[
\begin{align*}
x & \in \text{Var} & \text{Variable} \\
o & \in \text{OID} & \text{Object identifier} \\
p & \in \text{PVal} & \text{Primitive value} \\
f & \in \text{FID} & \text{Future identifier} \\
v & \in \text{Val} = \text{PVal} \cup \text{OID} \cup \text{FID} & \text{Value} \\
z & \in \text{Name} = \text{OID} \cup \text{FID} & \text{Name} \\
l & \in \text{MEnv} = \text{Var} \cup \{ \text{ret} \} \to \text{Val}_\bot & \text{Task environment} \\
a & \in \text{OEnv} = \text{Var} \cup \{ \text{self} \} \to \text{Val}_\bot & \text{Object environment} \\
tsk & \in \text{Tsk} ::= t(o,l,s) & \text{Task} \\
obj & \in \text{Obj} ::= o(o,a) & \text{Object} \\
fut & \in \text{Fut} ::= f(f,v_\bot) & \text{Future} \\
call & \in \text{Call} ::= c(o,f,m,\Xi) & \text{Call} \\
cn & \in \text{Cn} ::= 0 \mid ct \mid cn\text{ bind } z.cn & \text{Configuration} \\
obs & \in \text{Obs} ::= \text{ext}!m(\Xi) & \text{Observation}
\end{align*}
\]

---

Figure 3: mABS type 1 runtime syntax
environments \( l \) and \( o \), respectively, map task and object variables to values. Task environments are aware of a special variable \( \text{ret} \) that a task can use in order to identify its return future. Upon method invocation, a task environment is initialized using the operation \( \text{locals}(o, f, m, \tau) \) which maps the formal parameters of method \( m \) in the class of \( o \) to the corresponding arguments in \( \tau \), initializes the method local variables to suitable null values, and maps \( \text{ret} \) to \( f \), the return future of the task being created. Object environments are initialized using the operation \( \text{init}(C, v, o) \), which maps the parameters of the class \( C \) to \( v \), the special variable \( \text{self} \) to \( o \), and initializes the object variables as above. In addition to \( \text{locals} \) and \( \text{init} \), the reduction rules presented below use the auxiliary operation \( \text{body}(o, m) \), which retrieves the statement of the shape \( s \) in the definition body for \( m \) in the class of \( o \), and \( J_{e, l}(a, l) \in \text{Val} \) is used for evaluating the expression \( e \) in object environment \( a \) and task environment \( l \).

Calls play a special role in defining the external observations of a configuration \( cn \). Assume an OID \( ext \) representing the “outside world”, not allowed to be bound or defined in any well-formed configuration. An observation, or \( \text{barb} \), is a call of the form \( ext!m(\tau) \), ranged over by \( obs \). Calls that are not external are meant to be completed in the usual reduction semantics style, by internal reaction with the called object, spawning a new task. External calls could be represented directly, without relying on the call container type, by saying that a configuration \( cn \) has the barb \( obs = ext!m(\tau) \) whenever \( cn \) has the shape

\[
\text{bind } \tau \left( cn' \ o(o, a) \ t(o, l, x = e_1!m(\tau_1); s) \right),
\]

where \( [e_1]_{(a, l)} = ext \) and \( [\tau_1]_{(a, l)} = \tau \). However, in a semantics with unordered communication, which is what we are after, consecutive calls should commute, i.e., there should be no observational distinction between executing two method calls with the respective statements

\[
x = e_1!m_1(\tau_1'); y = e_2!m_2(\tau_2'); s
\]

and

\[
y = e_2!m_2(\tau_2'); x = e_1!m_1(\tau_1'); s
\]

This, however, is difficult to reconcile with the representation in (1). To this end, call containers are used for both internal and external calls, allowing configurations like (1) to produce a corresponding container, and then proceed to elaborate \( s \).

We next present the reduction rules. For ease of notation, the rules assume that sequential statement composition is associative with unit skip. The rules in Figure 4 and Figure 5 define the reduction relation. The rules use the notation \( cn \vdash cn' \rightarrow cn'' \) as shorthand for \( cn \ cn' \rightarrow cn \ cn'' \). Figure 4 gives the mostly routine rules for assignment, control structures, and contextual reasoning, and Figure 5 gives the more interesting rules that involve method invocation and object creation. A method call causes a new future identifier to be created, along with its future container, with lifted value initialized to \( \perp \). Future instantiation is done when \textbf{return} statements
are evaluated, and get statements cause the evaluating task to hang until the value associated with the future is defined, and then store that value. Object creation (new) statements cause new objects to be created along with their OIDs in the expected manner.

We note some basic properties of the reduction semantics.

**Proposition 4.1.** Suppose $cn \rightarrow cn'$. Then, the following holds:

1. $\text{fn}(cn') \subseteq \text{fn}(cn)$.

2. If $o(a, a') \not\subseteq cn$, then $o(a, a') \not\subseteq cn'$ for some object environment $a'$.

3. If $f(f, v_\bot) \not\subseteq cn$, then $f(f, v'_\bot) \not\subseteq cn'$ for some $v'_\bot$ such that $v_\bot \subseteq v'_\bot$.

**Proof.** No structural identity, nor any reduction rule, allows an OID or FID to escape its binder. No rules allow object or future containers to be removed. Also, no rules allow futures to be re-instantiated to $\bot$. The results follow. \qed

**Definition 4.2** (Type 1 Initial Configuration, Type 1 Reachable). Consider a program $\mathcal{CL}(\pi, s)$. The program can make calls to a special OID $\text{ext}$, and in this way produce externally observable output. Assume a reserved OID
\( o_{\text{main}} \) distinct from \( ext \), and a reserved FID \( f_{\text{init}} \). A type 1 initial configuration for the program has the shape

\[
\text{cn}_{\text{init}} = \text{bind} \ o_{\text{main}}, f_{\text{init}}.o(o_{\text{main}}, \bot) \ \text{t}(o_{\text{main}}, l_{\text{init}}, s) \ f(f_{\text{init}}, \bot),
\]

where \( l_{\text{init}} \) is the initial task environment assigning suitable default values to the variables in \( \pi \), and \( l_{\text{init}}(\text{ret}) = f_{\text{init}} \). When there is a derivation \( \text{cn}_1 \rightarrow \cdots \rightarrow \text{cn}_n \), we say that \( \text{cn}_n \) is reachable from \( \text{cn}_1 \). If \( \text{cn}_1 = \text{cn}_{\text{init}} \), \( \text{cn}_n \) is said to be type 1 reachable.

**Definition 4.3** (Type 1 Active Future). Let \( \text{cn} \) be a type 1 configuration. The future identifier \( f \) is active for the object \( o \) in \( \text{cn} \) if one of the following holds:

1. There is an object container \( o(o, a) \preceq \text{cn} \) such that \( a(x) = f \) for some \( x \).
2. There is a task container \( \text{t}(o, l, s) \preceq \text{cn} \) such that \( l(x) = f \) for some \( x \).
3. There is a call container \( c(o, f', m, \overline{\pi}) \preceq \text{cn} \) and \( f' = f \) or \( f \) occurs in \( \overline{\pi} \).
4. There is a future identifier \( f' \) that is active for \( o \) in \( \text{cn} \), and \( f(f', f) \preceq \text{cn} \).

**Definition 4.4** (Type 1 Well-formedness). A configuration \( \text{cn} \) is type 1 well-formed (WF1) if \( \text{cn} \) satisfies:

1. **OID Uniqueness**: If \( o(a_1, a_1) \) and \( o(a_2, a_2) \) are distinct object container occurrences in \( \text{cn} \), then \( a_1 \neq a_2 \).
2. **Task-Object Existence**: If \( \text{t}(o, l, s) \preceq \text{cn} \), then \( o(o, a) \preceq \text{cn} \) for some object environment \( a \).
3. **Call Uniqueness**: If \( c(o_1, f_1, m_1, \overline{\pi}_1) \) and \( c(o_2, f_2, m_2, \overline{\pi}_2) \) are distinct call container occurrences in \( \text{cn} \), then \( f_1 \neq f_2 \).
4. **Future Uniqueness**: If \( f(f_1, v_{\bot, 1}) \) and \( f(f_2, v_{\bot, 2}) \) are distinct future container occurrences in \( \text{cn} \), then \( f_1 \neq f_2 \).
5. **Single Writer**: If \( \text{t}(o_1, l_1, s_1) \) and \( \text{t}(o_2, l_2, s_2) \preceq \text{cn} \) are distinct task container occurrences in \( \text{cn} \) such that \( l_1(\text{ret}) = f_1 \) and \( l_2(\text{ret}) = f_2 \), then \( f_1 \neq f_2 \), \( f(f_1, \bot) \preceq \text{cn} \), and \( f(f_2, \bot) \preceq \text{cn} \), and additionally, if \( c(o, f, m, \overline{\pi}) \preceq \text{cn} \), then \( f \neq f_1 \) and \( f \neq f_2 \).
6. **Future Existence**: If \( f \) is active for \( o \) in \( \text{cn} \) or \( f(f', f) \preceq \text{cn} \), then \( f(f, v_{\bot}) \preceq \text{cn} \); if \( f(f, \bot) \preceq \text{cn} \), then there is either a call container \( c(o', f, m, \overline{\pi}) \preceq \text{cn} \) with \( o' \in \text{OID}(\text{cn}) \), or a task container \( \text{t}(o', l, s) \preceq \text{cn} \) such that \( l(\text{ret}) = f \).

Well-formedness is important, as it ensures that objects and futures, if defined, are defined uniquely, and that, e.g., tasks are defined only along with their accompanying object. The Single Writer property reflects the fact that only the task that was spawned along with some given future is able to assign to that future, and hence, if the task has not yet returned, the
future remains uninstantiated. Future Existence ensures that there are corresponding future containers for FIDs accessible to objects, and that those containers either carry values or have the potential of carrying a value.

**Proposition 4.5** (WF1 Preservation). Let $cn$ be a configuration. Then, the following holds:

1. If $cn$ is a type 1 initial configuration, then $cn$ is WF1.
2. If $cn$ is WF1 and $cn \rightarrow cn'$, then $cn'$ is WF1.
3. If $cn$ is type 1 reachable, then $cn$ is WF1.

**Proof.** The first two properties hold by inspection of the definitions and the rules. The last property holds by way of the first property and repeated application of the second property.

---

### 5 Type 1 Contextual Equivalence

Our approach to implementation correctness uses contextual equivalence [30]. The goal is to show that it is possible in a network-aware setting to remain strongly faithful to the reference semantics, provided all nondeterminism is deferred to a separate scheduler. This allows drawing strong conclusions also in the case where a scheduler is added, as we discuss in Section 11. Contextual equivalence requires of a pair of equivalent configurations, firstly, that the internal transition relation $\rightarrow$ is preserved in both directions, and secondly, that the relation is preserved when adding a context configuration, all while preserving a set of external observations. A number of works [22, 33] 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.

Let $obs = ext!m(\pi)$. The observation predicate $cn \downarrow obs$ is defined to hold just in case $cn$ can be written in the form

$$ bind \exists. (cn' c (ext, f, m, \pi)) . $$

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

**Definition 5.1** (Type 1 Witness Relation, Type 1 Contextual Equivalence). Let $\mathcal{R}$ range over binary relations on WF1 configurations. The relation $\mathcal{R}$ is a type 1 witness relation, if $cn_1 \mathcal{R} cn_2$ implies

1. **Reduction Closure**: If $cn_1 \rightarrow cn'_1$, then $cn_2 \rightarrow^* cn'_2$ for some $cn'_2$ such that $cn'_1 \mathcal{R} cn'_2$.
2. **Context Closure**: If $cn_1$ is WF1, then $cn_2$ is WF1 and $cn_1 \mathcal{R} cn_2$.
3. **Barb Preservation**: If $cn_1 \downarrow obs$, then $cn_2 \downarrow obs$.
Additionally, the converse properties must hold with $R^{-1}$ for $R$ above\(^1\). We define type 1 contextual equivalence, $\simeq_1$, as the union of all type 1 witness relations. Additionally, we say that the WF1 configurations $cn_1$ and $cn_2$ are type 1 contextually equivalent whenever $cn_1 \simeq_1 cn_2$, i.e., whenever $cn_1 \mathcal{R} cn_2$ for some type 1 witness relation $\mathcal{R}$.

We establish some well-known, elementary properties of contextual equivalence for later reference.

**Proposition 5.2.** The identity relation is a type 1 witness relation. $\simeq_1$ is a type 1 witness relation. If $R, R_1, R_2$ are type 1 witness relations then so is

1. $R^{-1}$,
2. $R^\ast$, and
3. $R_1 \circ R_2 \circ R_1$.

**Proof.** See Appendix 1.

We conclude that $\simeq_1$ has the expected basic property.

**Proposition 5.3.** $\simeq_1$ is an equivalence relation.

**Proof.** The result follows from Proposition 5.2. For transitivity, in particular, we use Proposition 5.2.3.

### 6 Network-Aware Semantics

We now turn to the second, main part of the paper where we address the problem of efficiently executing mABS programs on an abstract network graph using the location independent routing scheme alluded to in Section 1. The approach follows closely that for the network-aware semantics introduced in earlier work [11], with the important difference that method return values via futures are now included. In addition to the naming, routing, and object migration issues already addressed previously, the additional challenge is to ensure that futures are correctly assigned and propagated at the network level.

In the network-aware semantics, we assume an explicitly given network “underlay”: A network of nodes and directional links with which message buffers are associated, modeling a concrete network structure with asynchronous point-to-point message passing. Object execution is localized to each node. At the outset, nodes know only of their “own” objects, but as routing information is propagated, inter-node object-to-object message delivery becomes possible. Objects can migrate between neighboring nodes. When this is done is not addressed here; we discuss possible decentralized adaptation strategies, that in effect impose a scheduler, in other work [29]. The propagation of routing information will automatically lead to routing tables becoming up-to-date. How and when this is done is again left to a

---

\(^1\)The usual explicit symmetry requirement is slightly too strong for our purpose.
scheduler. Method calls to an object can be issued if a task has access to the OID of the object, and the associated message can be delivered once a route to the callee becomes known.

For the mABS language, the network-aware semantics must be extended to cover also return values and futures. We use an eager forward-based strategy [5, 18] for handling futures. The strategy’s central idea is that, whenever an object shares a future identifier, the object assumes an obligation to send the associated value to the object with which the future identifier is shared. Initially, the value may be unavailable, requiring the use of forwarding lists for futures stored in the object state. Our objective is to prove that this approach is sound and fully abstract for our network-aware semantics, even though routing may be in an unstable state.

Example 6.1. An example of an execution, which illustrates future propagation and its interaction with routing, is shown in Figure 6. In configuration 1 in the upper left corner, a call to method \( m \) with argument \( f \), a future identifier, is about to be sent from object \( o_1 \) residing on node \( u_1 \) to object \( o_2 \) on node \( u_3 \). The following events then take place as the system evolves into configuration 2 in the upper right corner:
• A new future \( f' \) is created at object \( o_1 \), as a placeholder for the return value of the call to method \( m \).

• The forwarding list for \( f \) at \( o_1 \) is augmented to include \( o_2 \).

• The call is routed to \( o_2 \), and a new task computing the statement \( \text{body}(o_2, m) \) is spawned at \( o_2 \), associated with \( f' \).

• \( o_2 \) is augmented with \( f \) and \( f' \) as placeholders, and the forwarding list for \( f' \) at \( o_2 \) is augmented to include \( o_1 \).

The scheduler now decides to migrate \( o_2 \) from \( u_3 \) to \( u_2 \). No action is required other than regular routing table updates, as the forward pointers keep referencing the same objects. Finally, the future identifier \( f \) is returned by the task at \( o_2 \), as shown in the lower left configuration 3. In the lower right configuration 4, the value of the future \( f' \), namely, \( f \), has been sent to \( o_1 \), where it becomes assigned to the variable \( y \). The obligation of \( o_1 \) to forward the value of \( f \) to \( o_2 \) has therefore become redundant and could be removed without affecting object behavior, but for simplicity we do not include such garbage collection in the semantics. The price for this omission is additional messaging, but the latency remains the same, since \( o_1 \) is able to discover an assignment to \( f \) at \( o_1 \) when it is first made.

6.1 Runtime Syntax

In Figure 7, we present the network-aware mABS runtime syntax, i.e., the shape of the runtime state. Recall from Section 4 that we reuse symbols

\[
\begin{align*}
\text{u} &\in \text{NID} & \text{Node identifier} \\
\text{a} &\in OEnv_2 & (\text{Var} \cup \{\text{self}\} \rightarrow \text{Val}_\perp) \times (\text{FID} \rightarrow (\text{Val}_\perp \times (\text{OID} \text{ list}))_\perp) \quad & \text{Object environment} \\
\text{t} &\in RTable & \text{Routing table} \\
\text{q} &\in Q & \text{Message queue} \\
\text{obj} &\in \text{Obj}_2 & o(a, a, u, q_{in}, q_{out}) \quad & \text{Object} \\
\text{nd} &\in \text{Nd} & n(u, t) \quad & \text{Network node} \\
\text{lnk} &\in \text{Lnk} & l(u, q, u') \quad & \text{Network link} \\
\text{ct} &\in \text{Ct}_2 & \text{tsk} \mid \text{obj} \mid \text{nd} \mid \text{lnk} \quad & \text{Container} \\
\text{cn} &\in \text{Cn}_2 & \text{ct}_1 \ldots \text{ct}_n \quad & \text{Configuration} \\
\text{msg} &\in \text{Msg} & \text{call}(o, a', f, m, v) \mid \text{future}(o, f, v) \mid \text{table}(t) \mid \text{object}(cn) \quad & \text{Message}
\end{align*}
\]

Figure 7: mABS type 2 runtime syntax

as much as possible and use indices to disambiguate. Thus, for instance, \( \text{Obj}_1 \) is the set \( \text{Obj} \) of the type 1 semantics in Figure 3, and \( \text{Obj}_2 \) is the corresponding set in Figure 7. We adopt the same syntactical conventions as in Section 4. Tasks are unchanged from Figure 3. We write \( t(cn) \) for the multiset of tasks in \( cn \), i.e., the multiset \( \{\text{tsk} \mid \text{tsk} \preceq cn\} \), and \( o(cn) \) for
the multiset of objects in $cn$, similarly defined. We also write $m(cn)$ for the multiset $\{msg \mid msg \preceq cn\}$.

We proceed to explain the different types of containers and the operations on them, concentrating on the treatment of futures. For a more detailed explanation of other features, in particular routing, we refer to our earlier work [11].

Network and Routing The nodes and links in a configuration $cn$ induce a network graph $\text{graph}(cn)$, which contains a vertex $u$ for each node container $n(u,t)$ and an edge $(u,u')$ for each link $(l(u,q,u'))$. The reduction semantics given later does not allow identifiers in nodes or links to be changed, so in the context of any given transition (or, execution), the network graph remains constant. Note that there is no a priori guarantee that the network graph is a well-formed graph. For the remainder of the paper, we therefore impose some constraints on the well-formedness of the network graph, namely, that (i) there is at least one vertex, (ii) endpoints of edges exist, (iii) vertices and edges are uniquely determined, (iv) the network graph is reflexive and symmetric, and (v) the network graph is connected. For routing, we adopt a simple Bellman-Ford distance vector discipline [38]. For a routing table $t$, $t(o) = (u,n)$ indicates that, as far as $t$ is concerned, there is a path from the current node (the node to which $t$ is attached) to the object $o$ with distance $n$, that first visits the node $u$. We only count hops to compute distance, for simplicity. A more realistic routing scheme associates weights with edges, reflecting latency or capacity constraints. Next hop lookup is performed by the operation $\text{nxt}(o,t) = \pi_1(t(o))$ where $\pi_1$ is the first projection. There is also an operation $\text{upd}$ for updating a routing table $t$ by a routing table $t'$ received from a neighboring node $u$, defined by

$$\text{upd}(t,u,t')(o) = \left\{ \begin{array}{ll} \bot & \text{if } o \notin \text{dom}(t) \cup \text{dom}(t') \\ t(o) & \text{else, if } o \notin \text{dom}(t') \\ (u,\pi_2(t(o)) + 1) & \text{else, if } o \notin \text{dom}(t) \\ (u,\pi_2(t'(o)) + 1) & \text{else, if } \pi_1(t'(o)) = u \\ (u,\pi_2(t(o)) + 1) & \text{else, if } \pi_2(t'(o)) + 1 < \pi_2(t(o)) \\ t(o) & \text{otherwise.} \end{array} \right.$$  

Finally, there is an operation $\text{reg}(o,u,t,n)$ that returns the routing table $t'$, obtained by registering the object identifier $o$ at $t'$'s current node $u$ with distance $n$, i.e., such that

$$\text{reg}(o,u,t,n)(o') = \left\{ \begin{array}{ll} (u,n) & \text{if } o = o' \\ t(o) & \text{otherwise.} \end{array} \right.$$  

Message Queues FIFO message queue operations are standard: $\text{enq}(msg,q)$ enqueues a message $msg$ onto the tail of the queue $q$, $\text{hd}(q)$ returns the head of $q$, and $\text{deq}(q)$ returns the tail of the $q$, i.e., $q$ with $\text{hd}(q)$ removed. If $q$ is empty, then $\text{hd}(q) = \text{deq}(q) = \bot$.

Messages The network-aware semantics uses four forms of messages:
• call(o, o', f, m, v): A call message, corresponding to a call container in the reference semantics, where o is the identifier of the callee and o' the identifier of the caller, respectively.

• future(o, f, v): A future instantiation message, meant to inform object o that the future f has been instantiated to the value v.

• table(t): A routing table update message, with the routing table t.

• object(cn): A message containing an object closure cn of the form o(a, u, q_in, q_out) t(o, l_1, s_1) ... t(o, l_n, s_n), as explained in more detail below, used for migrating objects and their tasks across nodes. Note that containers inside object closure messages in a link queue q are included in the subterm relation ⪯ for the configuration in which the link with q resides.

Call and future instantiation messages are said to be object bound, while routing table messages and object messages are node bound. We define dst(msg), the destination of msg, to be o for a call message or a future message as defined above, and ⊥ in the remaining two cases.

Objects and Object Environments Object containers o(a, u, q_in, q_out) are now attached to a node u and equipped with an ingoing (q_in) and an outgoing (q_out) FIFO message queue, and the notion of object environment is refined to take futures into account in a localized manner. In the type 2 semantics, object environments a are now augmented by mapping futures f to pairs (v⊥, o), where:

• v⊥ is the lifted value currently associated with f at the current object, and

• o is a forwarding list, containing the identifiers of the objects subscribing to instantiations of f at the current object.

For instance, if a(f) = (⊥, o_1 o_2), the future f is as yet uninstantiated (at the object to which a belongs), and, if f eventually does become instantiated, the instantiation must be forwarded to o_1 and o_2. Forwarding does not necessarily happen in the given order, since we consider forwarding lists modulo associativity and commutativity with the empty list ε as unit.

We introduce some notation and auxiliary operations to help manipulating object environments:

• a(x) abbreviates π_1(a)(x), and a(f) abbreviates π_2(a)(f).

• a[v/x] is a with π_1(a) replaced by the expected update. Similarly, a[v/f] updates π_2(a) by mapping f to the pair (v, π_2(a(f))), i.e., the assigned value is updated and the forwarding list remains unchanged. If f /∈ dom(π_2(a)), then a[v/f](f) = (v, ε), i.e., the update to the value takes effect. Finally, we use a[(v, o)/f] for the expected update where both the value and the forwarding list are updated.
fw(π, o, a) updates π2(a) by, for each future f occurring in π, adding o to the forwarding list of a(f), i.e., by mapping f to either (⊥, o) if a(f) ↑, or (π1(a(f)), o π2(a(f))) otherwise.

init(C; π, o) returns an initial object environment, by mapping the formal parameters of C to π, and self to o; this operation is unchanged from the reference semantics.

init(π, o) augments a by mapping each FID f in π which is uninitialized in a (i.e., such that f \not\in \text{dom}(a)) to (⊥, ε).

As a consequence of these changes, futures are eliminated as containers in the type 2 runtime syntax. In other respects, the type 2 runtime syntax is unchanged: Syntactical conventions that are not explicitly modified in the type 2 syntax above remain the same. In particular, we continue to assume the commutativity and associativity properties of configuration juxtaposition, now with the empty list of containers as unit.

Object Closures  Object closures wrap objects with their active tasks. We use the following operations:

- clo(cn, o), the closure of object o with respect to the configuration cn, is the multiset of all type 2 containers in cn of the form o(o′, a′, u′, q′ in, q′ out) or t(o′, l′, s′), such that o′ = o.

- oidof(cn), a partial function returning the OID o, if all the type 2 containers in cn are objects and tasks with OID o.

- place(cn, u) places all object containers in the configuration cn at the node u, i.e., cn and place(cn, u) are identical, except that each object container o(o′, a′, u′, q′ in, q′ out) in cn is replaced by an object container o(o′, a′, u, q′ in, q′ out) in place(cn, u).

6.2 Reduction Semantics

An important distinction between the reference semantics and the network-aware semantics is the absence of binding. For the standard semantics, name binding plays an important role in avoiding clashes between locally generated names. However, in a language with node identifiers (NIDs) this device is no longer needed, as globally unique names can be guaranteed easily by augmenting names with their generating NID. Since all name generation in the mABS type 2 semantics below takes place in the context of a given NID, we can simply assume the existence of two operations newf(u) and newo(u), that return a new future and a new OID, respectively, that is globally fresh for the “current context”, with newo(u) distinct from ext.

We now present the mABS type 2 reduction rules. The first part, in Figure 8, is carried over from the type 1 semantics in Figure 4, with some minor modifications. First, rule CTXT-2 is dropped, since name binding is dropped from the type 2 runtime syntax. Second, the rules WFIELD, IF-TRUE, IF-FALSE, WHILE-TRUE, and WHILE-FALSE are straightforwardly modified to account for
the new runtime shape of objects. The remaining reduction rules are given in Figure 9; these rules can be divided into groups as per below.

Routing  The first set of rewrite rules, $t$-send and $t$-rcv, are concerned with the exchange of routing tables, which only takes place between distinct adjacent nodes.

Message Passing  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. If the destination object does not reside at the current node, the message is routed to the next link. In rule msg-rcv, note that the receiving node is not required to be present. This, however, is enforced by the well-formedness condition for the network graph, which prohibits output links.

Unstable Routing  The three rules msg-delay-1, msg-delay-2, and msg-delay-3 are used to handle the cases where routing tables have not yet stabilized, or a message is unroutable. 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 object out-queue without the hosting node’s routing table having information about the message’s destination (rule msg-delay-2). Another case is when 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 a deadlock. The solution we propose, which is implicit in the rules, is to use the network self-loop links, included in all well-formed networks, as buffers for unroutable mes-
The four rules in Table 9: mABS type 2 reduction rules, part 2

sages that may become routable. Rule MSG-DELAY-3 allows messages on this link to be shuffled.

Producing and Consuming Messages The four rules CALL-SEND-2, CALL-RCV-2, FUT-SEND, and FUT-RCV produce and consume method call and future
instantiation messages, respectively. A method call causes a local future identifier to be created and passed along with the call message. Upon receiving the call, the callee first initializes the received futures it does not already know about, and then augments the resulting local object environment to enact forwarding for the received future to the caller, when possible. Observe that it may be the case that the callee already knows about the return future of the call; since message order is not assumed to be preserved, a later call with the original return future may overtake the earlier call. The eventual return value becomes associated with the return future by the assignment to the constant \( \text{ret} \) during initialization of the task’s local environment. The rule fut-send lets future instantiations be forwarded to objects in the forwarding list whenever the future is instantiated to a value locally, and fut-rcv causes the receiving object to update its local environment accordingly. A future may itself be instantiated to a future, making it necessary to update the local forwarding list whenever fut-send is used.

Language Constructs The three rules \( \text{ret-2}, \text{get-2}, \text{and new-2} \) handle the corresponding language constructs. Return statements cause the corresponding future to be instantiated, as explained above. Get statements read the value of the future, provided it has received a value, and new statements cause a new object to be created, initialized, and registered at the local node. The arguments provided to the new object may contain future identifiers, whose values must be duly forwarded to the new object by augmenting the old object’s forwarding lists.

Object Registration and Migration The rule obj-reg registers a new object on the node on which it has been placed. The final two rules concern object migration. Of these, the rule obj-send is global 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. To remove an object closure \( \text{cn}' \) from a configuration \( \text{cn} \) for migration, we take the multiset difference \( \text{cn} - \text{cn}' \).

It is important to note that all of the above rules are strictly local and appeal only to mechanisms directly implementable at the link level, i.e., they correspond to tests and simple datatype manipulations taking place at a single node, or accesses to a single node’s link layer interface. The “global” property appealed to above for migration 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 node’s self-loop link. 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 6.2.** Suppose that \( \text{cn} \rightarrow \text{cn}' \). Then, the following holds:

1. If \( n(u, t) \preceq \text{cn} \), then \( n(u, t') \preceq \text{cn}' \) for some \( t' \).
2. If \( l(u, q, u') \preceq cn \), then \( l(u, q', u') \preceq cn' \) for some \( q' \).

3. If \( o(a, u, q_{in}, q_{out}) \preceq cn \), then there is an object \( o(a', u', q'_{in}, q'_{out}) \preceq cn' \) (the derivative of the object in \( cn' \)), such that for all variables \( x \), if \( o(x) \downarrow \), then \( o'(x) \downarrow \), and for all future identifiers \( f \), if \( o'(f) \downarrow \), then \( o'(f) \downarrow \), and if \( \pi_1(o(f)) \downarrow \), then \( \pi_1(o'(f)) \downarrow \).

4. If \( t(o, l, s) \preceq cn \) and \( l(\text{ret}) = f \), then either there is a task \( t(o, l', s') \preceq cn' \) (the derivative of the task in \( cn' \)), such that \( \text{dom}(l) \subseteq \text{dom}(l') \), and \( l'(\text{ret}) = f \), or there is an object \( o(a, u, q_{in}, q_{out}) \preceq cn' \) such that \( \pi_1(a(f)) \downarrow \).

Proof. By straightforward induction on the reduction relation. \( \square \)

**Definition 6.3** (Type 2 Initial Configuration, Type 2 Reachable). Consider a program \( \overline{CL} \{ \pi, s \} \), using the special OID \( \text{ext} \) to communicate with the outside the world. Assume a reserved OID \( o_{\text{main}} \), distinct from \( \text{ext} \), and a reserved FID \( f_{\text{init}} \). A type 2 initial configuration for the program has the shape

\[
\text{cn}_{\text{init}} = o(o_{\text{main}}, a_{\text{init}, 2}, u_{\text{init}}, \varepsilon, \varepsilon) \cup (o_{\text{main}}, t_{\text{init}}, s) \text{ cn}_{\text{graph}},
\]

where

- \( a_{\text{init}, 2} = \bot[\bot, \varepsilon]/f_{\text{init}}, \)
- \( l_{\text{init}} \) is unchanged from the definition of type 1 initial configurations,
- \( \text{cn}_{\text{graph}} \) is a configuration consisting only of nodes and links, inducing a well-formed network graph,
- \( \text{cn}_{\text{graph}} \) contains a node \( n(u_{\text{init}}, t_{\text{init}}), \)
- \( t_{\text{init}}(o_{\text{main}}) = (u_{\text{init}}, 0) \), and \( t_{\text{init}}(o) = \bot \) for all OIDs \( o \) distinct from \( o_{\text{main}}, \)
- \( t(o) = \bot \) for all routing tables \( t \neq t_{\text{init}} \) in \( \text{cn}_{\text{graph}} \) and for all OIDs \( o \), and
- \( l(u, q, u') \preceq \text{cn}_{\text{graph}} \) implies \( q = \varepsilon \), for all \( u \) and \( u' \).

When there is a derivation \( \text{cn}_1 \rightarrow \cdots \rightarrow \text{cn}_n \), we say that \( \text{cn}_n \) is reachable from \( \text{cn}_1 \). If \( \text{cn}_1 \) is a type 2 initial configuration, \( \text{cn}_n \) is said to be type 2 reachable.

### 7 Type 2 Well-formedness

Well-formedness becomes more complex in the case of the network-aware semantics, since account must be taken of, e.g., queues, messages in transit, and routing, to ensure that, e.g., multiple objects are never given identical names, and that futures are never assigned inconsistent values, as detailed below. A particularly delicate matter concerns the way future instantiations are propagated. The well-formedness condition needs to ensure that either
all objects that may at some time need the value of a future can also eventually receive it, or else no object is able to do so (in which case the task for which the future is a placeholder for the return value fails to terminate). This is the “future liveness” property in Definition 7.5 below. To this end, we first define when a future identifier is assigned in a configuration and active for an object.

Definition 7.1 (Future Assignment). We say that a configuration \( cn \) assigns the value \( v \) to \( f \) if there is an object container \((o, a, u, q_{\text{in}}, q_{\text{out}}) \preceq cn\), such that either \( \pi_1(a(f)) = v \) or there is a message \( \text{future}(o, f, v) \preceq cn \). If there is no such value \( v \), we say that \( f \) is \textit{unassigned} in \( cn \).

Definition 7.2 (Type 2 Active Future). Let \( cn \) be a type 2 configuration. The future identifier \( f \) is \textit{active} for the object with identifier \( o \) in \( cn \) if one of the following holds:

1. There is an object container \((o, a, u, q_{\text{in}}, q_{\text{out}}) \preceq cn\) such that \( a(x) = f \) for some variable \( x \).

2. There is a task container \((t, o, l, s) \preceq cn\) such that \( l(x) = f \) for some \( x \).

3. There is a call message \( \text{call}(o, o', f', m, v) \) in transit in \( cn \), and \( f' = f \) or \( f \) occurs in \( v \).

4. There is a future identifier \( f' \) that is active for \( o \), and \( cn \) assigns \( f \) to \( f' \).

Thus, \( f \) is active for \( o \) if \( f \) occurs in the environment of the object or one of its tasks, the object is about to receive a method call with \( f \) as the return future or as one of its arguments, or \( f' \) is active for \( o \) and \( f' \) has been instantiated to \( f \) somewhere.

We next define which objects are due to be notified by eventual future instantiations.

Definition 7.3 (Notification Path). Fix a type 2 configuration \( cn \), an object container \((o, a, u, q_{\text{in}}, q_{\text{out}}) \preceq cn\), and an OID \( o' \in \text{OID}(cn) \). Let \( n \) be a nonnegative integer. Inductively, \( o \) is \textit{on the notification path of} \( f \) \textit{by} \( o' \) \textit{in} \( n \) \textit{steps}, if \( n \) is the least number such that one of the following conditions hold:

1. \( n = 0 \), \( o' = o \), and \( \pi_1(a(f)) = v \).

2. \( n = 1 \), \( o' = o \), and there is a future message \( \text{future}(o, f, v) \preceq cn \).

3. \( n = 1 \), \( o' = o \), and there is a task \( t(o, l, s) \preceq cn \) with \( l(\text{ret}) = f \).

4. \( n = 2 \), \( o' = o \), and there is a call message \( \text{call}(o, o'', f, m, v) \preceq cn \).

5. \( n = 4 \), and there is a call message \( \text{call}(o', o, f, m, v) \preceq cn \).

6. \( n = n' + 2 \), with \( n' \) a nonnegative integer, and there is an object \((o'', a', u', q_{\text{in}'}, q_{\text{out}'}) \preceq cn\) such that \( o \in \pi_2(a'(f)) \), and \( o' \) is on the notification path of \( f \) by \( o' \) in \( n' \) steps.
7. \( n = 2n' + n'' \), with \( n' \) and \( n'' \) nonnegative integers, if \( o \) is on the notification path of \( f' \) by \( o'' \) in \( n' \) steps, \( cn \) assigns \( f \) to \( f' \), and \( o'' \) is on the notification path of \( f \) by \( o' \) in \( n'' \) steps.

Say that \( o \) is on the notification path of \( f \), if \( o \) is on the notification path of \( f \) by some \( o' \) in some number of steps.

Intuitively, the number of steps in a notification path is the number of events that need to take place before a future becomes assigned at the object, considering task evaluation a single event. For instance, if \( o' \) has a task evaluating \( f \) and has added \( o \) to the forwarding list for \( f \), it takes three steps for the value of \( f \) to reach \( o \): the evaluation of the task, the sending of a future message, and the reception of that future message by \( o \).

Condition 1 is the base case when \( f \) has already been instantiated. Condition 2 holds if a future has been resolved and a future message is in transit to \( o \). Condition 3 holds if \( o \) is due to receive the return value of \( f \) from one of its pending tasks. Condition 4 holds if a call to \( o' \) has been sent off from \( o \) with return future \( f' \) and \( o' \) is then on the notification path of \( f \) since the call message is guaranteed to be received at the callee’s site (if at all) and the forwarding list there updated. Condition 5 holds if \( o \) has been inserted into a forwarding list for \( f \) at closer distance to a “source”. Condition 6 holds if \( f \) is assigned to another future \( f' \), \( o'' \) is a “source” of \( f' \) for \( o \), and \( o'' \) is due to receive the return value for \( f' \), which can then reach \( o \) through forwarding list additions in transitions using the rule FUT-SEND.

We prove that if \( o \) is on the notification path of \( f \), then in the next configuration \( o \) remains on the notification path of \( f \), without increasing the number of steps.

**Lemma 7.4.** Fix a configuration \( cn \) and an object \( o(a, u, q_{in}, q_{out}) \prec cn \). If \( o \) is on the notification path of \( f \) by \( o' \) in \( n \) steps in the configuration \( cn \) and \( cn \rightarrow cn' \), then \( o \) is on the notification path of \( f \) by \( o' \) in at most \( n \) steps in \( cn' \).

**Proof.** See Appendix 2.

We can now finally state the conditions of type 2 well-formedness.

**Definition 7.5 (Type 2 Well-formedness).** A type 2 configuration \( cn \) is type 2 well-formed (WF2) if \( cn \) satisfies:

1. **OID Uniqueness:** If \( o(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \) and \( o(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \) are distinct object container occurrences in \( cn \), then \( o_1 \neq o_2 \).

2. **Task-Object Existence:** If \( t(o, l, s) \preceq cn \), then \( o(a, u, q_{in}, q_{out}) \preceq cn \) for some \( a, u, q_{in}, \) and \( q_{out} \).

3. **Object-Node Existence:** If \( o(a, u, q_{in}, q_{out}) \preceq cn \), then \( n(u, l) \preceq cn \) for some \( t \).

4. **Buffer Cleanliness:** If \( o(a, u, q_{in}, q_{out}) \preceq cn \) and \( msg \preceq q_{in} \) or \( msg \preceq q_{out} \), then \( msg \) is object bound. Additionally, if \( msg \preceq q_{in} \), then \( \text{dst}(msg) = o \).
5. Local Routing Consistency: If \( n(u, t) \preceq cn \) and \( \text{nxt}(o, t) = u' \), then there is a link \( l(u, q, u') \preceq cn \).

6. Call Uniqueness: If \( \text{call}(o_1, o'_1, f_1, m_1, \overline{v}) \) and \( \text{call}(o_2, o'_2, f_2, m_2, \overline{v}) \) are distinct call message occurrences in \( cn \), then \( f_1 \neq f_2 \).

7. Future Uniqueness: If \( cn \) assigns both \( v_1 \) and \( v_2 \) to \( f \), then \( f_1 \neq f_2 \).

8. Single Writer: If \( t(o_1, l_1, s_1) \) and \( t(o_2, l_2, s_2) \) are distinct task container occurrences in \( cn \) such that \( l_1(\overline{\text{ret}}) = f_1 \) and \( l_2(\overline{\text{ret}}) = f_2 \), then \( f_1 \neq f_2 \) and both \( f_1 \) and \( f_2 \) are unassigned in \( cn \), and if \( \text{call}(o, o', f, m, \overline{v}) \preceq cn \), then \( f \neq f_1 \) and \( f \neq f_2 \).

9. External OID: \( \text{ext} \notin \text{OID}(cn) \), and if \( n(u, t) \preceq cn \), then \( \text{ext} \notin \text{dom}(t) \).

10. Future Liveness: \( o \in (o, a, u, q_{\text{in}}, q_{\text{out}}) \preceq cn \), and either \( f \) is active for \( o \) in \( cn \), \( a(f) \downarrow \), or \( cn \) assigns \( f \) to \( f' \) and \( o \) is on the notification path of \( f' \), then \( o \) is on the notification path of \( f \).

Conditions 7.5.1 to 7.5.5 and 7.5.9 are inherited from our preceding work [11]. 7.5.4 is needed to prevent the formation of contexts that are deadlocked because an in- or out-queue contains messages of the wrong type. The rationale behind 7.5.8 is that mABS enforces a single-writer discipline on instantiated values of futures. Once a future has been instantiated through the evaluation of a return statement, the task is “garbage collected” in the rule \( \text{ret-2} \). 7.5.9 ensures that messages to \( \text{ext} \) are only transported to reflexive links, where they remain. Condition 7.5.10 is the future propagation property discussed above. We use the designation Future Liveness not to indicate a guarantee that \( f \) will eventually be instantiated at an object, but to indicate that, if eventually \( f \) is instantiated somewhere, a notification path to the object exists along which the instantiation can be propagated.

**Lemma 7.6** (WF2 Preservation). Let \( cn \) be a configuration. Then, the following holds:

1. If \( cn \) is a type 2 initial configuration, then \( cn \) is WF2.

2. If \( cn \) is WF2 and \( cn \rightarrow cn' \), then \( cn' \) is WF2.

3. If \( cn \) is type 2 reachable, then \( cn \) is WF2.

**Proof.** See Appendix 2.

An easy but important consequence of type 2 well-formedness is that assignments to futures cannot be updated with new values.

**Proposition 7.7.** Suppose that \( cn \) is WF2 and \( cn \rightarrow cn' \). If \( cn \) assigns \( v \) to \( f \) and \( cn' \) assigns \( v' \) to \( f \), then \( v = v' \).

**Proof.** Since \( cn \) is WF2, if \( cn \) assigns \( v \) to \( f \), there cannot be a task \( t(o, l, s) \preceq cn \) such that \( t(\overline{\text{ret}}) = f \). But the only way of assigning \( v' \neq v \) to \( f \) is through \( \text{ret-2} \). Hence, the result follows.
8 Type 2 Contextual Equivalence

We adapt the notion of contextual equivalence to the type 2 setting as in the preceding work [11]. The initial problem is to define the type 2 correlate of the observation predicate. Say a configuration $cn$ has the observation, or $\text{barb, obs} = \text{ext}\!\!m(\overline{v})$ if a corresponding call message $\text{call}(\text{ext}, o, f, m, \overline{v})$ is located at the head of one of the self-loop link queues in $cn$. More precisely, the type 2 observability predicate $cn \downarrow \text{obs}$ holds whenever $cn$ has the shape

$$cn = cn' l(u, q, u),$$

and $\text{hd}(q)$ is defined and equal to $\text{call}(\text{ext}, o, f, m, \overline{v})$.

It might be thought that the FIFO queue discipline goes against the treatment of external calls in the type 1 semantics, since there, an external call container, once created, will remain as an inert element of all subsequent configurations. Thus, once, say, five call containers have been created, all five calls can be observed at all configurations from that point onwards. This is obviously not the case in the type 2 semantics. On the other hand, in that semantics, external call messages can always be shuffled on a reflexive link using the rule $\text{msg-delay-3}$, allowing specific calls to reach the head of the queue.

There are other ways of defining the observability predicate that may be more natural. For instance, one may attach external OIDs to specific NIDs and restrict observations to those NIDs 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, particularly in the latter case.

For context closure, a context is any configuration $cn$ containing only object and task containers. Thus, contexts do not affect the underlying network graph. This definition is used, since, firstly, it is objects and tasks that induce computational behavior, and secondly, allowing contexts to augment the underlying graph by adding new nodes and links requires a much more complex account of network composition and well-formedness that we prefer to leave to future work.

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 reduction closure, and barb preservation, with context closure defined as follows: If $cn_1 \mathcal{R} cn_2$, $cn$ is a context, and $cn_1 \mathcal{cn}$ is WF2, then $cn_2 \mathcal{cn}$ is WF2, and $cn_1 \mathcal{cn} \mathcal{R} cn_2 \mathcal{cn}$. Thus:

**Definition 8.1 (Type 2 Contextual Equivalence).** Let $cn_1 \approx_2 cn_2$ whenever $cn_1 \mathcal{R} cn_2$ for some type 2 witness relation $\mathcal{R}$. It is straightforward to relate a type 2 context with a type 1 context. Given a type 2 context $cn$, produce a type 1 configuration by first mapping pointwise an object container $o(a, u, q_{in}, q_{out}) \preceq cn$ to $o(a)$, each message $\text{call}(o, o', f, m, \overline{v})$ in $q_{in}$ and $q_{out}$ to a call container $c(o, f, m, \overline{v})$, and each task $\text{tsk} \preceq cn$ to itself. Then, for all $f$ active for some $o$ in $cn$, add a future
container \( f(v) \) if \( cn \) assigns \( v \) to \( f \), and \( f(\bot) \) if \( f \) is unassigned in \( cn \). The resulting configuration is well-defined as long as object queues only contain messages that are object bound, which is the case for configurations that are WF2. With this proviso, we do not need to distinguish between the type 1 and type 2 equivalences, and hence we conflate the respective notions of witness relation and contextual equivalence, by letting the type of the configuration arguments be determined by the context, and use \( \simeq \) as the generic notion.

9 Normal Forms

An mABS program can be run from an initial state in either the type 1 or the type 2 semantics. We want to show that the type 1 behavior of a program is preserved in the type 2 semantics, in the sense that the type 1 and type 2 initial states are contextually equivalent.

The key to the proof is a normal form lemma for mABS saying, roughly, that any well-formed type 2 configuration can be rewritten into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, where all futures that are assigned a value somewhere are assigned a value everywhere the value might be needed (by well-formedness, this value is unique), and where all objects have been moved to a single node. We perform this rewriting using two algorithms. The first algorithm stabilizes routing and empties link queues, leaving only external (“unroutable”) messages. The second algorithm, which uses the first, empties object queues, propagates futures, and moves all objects to a single node.

9.1 Stabilization

In the scope of a configuration \( cn \), we call a link \( l(u,q,u') \preceq cn \) proper whenever \( u \neq u' \), and say that a message \( msg \) is routable whenever \( \text{dst}(msg) \in \text{OID}(cn) \), and unroutable otherwise.

Definition 9.1 (Stable Routing). Let \( cn \) be a type 2 configuration. We say that \( cn \) has stable routing, if for all \( n(u,t), o(a,a',u',q_{in},q_{out}) \preceq cn \), if \( \text{nxt}(a,t) = u'' \), then there is a minimal length path from \( u \) to \( u' \) in \( \text{graph}(cn) \) which visits \( u'' \).

Definition 9.2 (External Link Messages). Let \( cn \) be a type 2 configuration. We say that \( cn \) has external link messages, if \( l(u,q,u') \preceq cn \) and \( msg \preceq q \) implies \( msg \) is object bound and unroutable.

To converge to a configuration with stable routing, the idea is to empty link queues as far as possible, and let nodes simultaneously exchange routing tables. This is accomplished using Algorithm 1, where we hide uses of \( \text{ctxt-1} \) to allow the transition rules to be applied to arbitrary containers. Write \( A_1(cn) \rightarrow cn' \) if \( cn' \) is a possible result of applying Algorithm 1 to \( cn \). The resulting configuration is almost unique, but not quite, since routing may stabilize in different ways.
Algorithm 1: Stabilize routing and process link messages

**Input** A type 2 well-formed configuration \( cn \)

**Output** A configuration with stable routing and external link messages, reachable from \( cn \)

**repeat**
use OBJ-REG on each object not in transit;
use T-SEND on each proper link to broadcast routing tables from all nodes to their neighboring nodes;
**repeat**
use T-RCV to dequeue one message on a link
until T-RCV can no longer be used;
once for each link, if possible, use MSG-RCV, MSG-ROUTE, MSG-DELAY-1, or OBJ-RCV, or otherwise, use MSG-DELAY-3 on self-loop links with external messages at the queue head
until routing has stabilized and there are only external link messages

Figure 10: Algorithm 1

**Proposition 9.3.** Algorithm 1 terminates.

**Proof.** See Appendix 3.

Let \( o_1(cn) \) be the multiset of object containers where, if it is the case that \( o(o, a, u, q_{in}, q_{out}) \leq cn \), there is a corresponding container \( o(o, a, u', q'_{in}, q'_{out}) \), such that the NID \( u' \) is that of the receiving node if the object is in transit in \( cn \), or otherwise \( u' = u \), and additionally, all messages \( msg \) in link queues such that \( dst(msg) = o \) have been enqueued in some fixed order in \( q_{in} \) to produce \( q'_{in} \). Define the relation \( \equiv_1 \) to hold between multisets of object containers when there is a one-to-one mapping where elements only possibly differ in how in-queue messages are ordered. Finally, let \( m_1(cn) \) be the multiset of both external messages and routable messages in \( cn \).

**Definition 9.4** (Stable Form). A WF2 configuration \( cn \) is in **stable form**, if \( cn \) has stable routing, \( o(cn) \equiv_1 o_1(cn) \), and \( m(cn) = m_1(cn) \).

**Proposition 9.5.** If \( A_1(cn) \rightsquigarrow cn' \), then

1. \( cn' \) is reachable from \( cn \),
2. \( cn' \) is in stable form,
3. \( \text{graph}(cn') = \text{graph}(cn') \),
4. \( t(cn') = t(cn) \),
5. \( o(cn') \equiv_1 o_1(cn) \), and
6. \( m(cn') = m_1(cn) \).

**Proof.** Properties 1, and 3 are immediate. Property 2 can be read out of the termination proof. For the remaining three properties, observe first that t,
o₁, and m₁ are all invariant under the transitions used in Algorithm 1. The equations then follow by noting that only externally-addressed messages (and so no object closures or routing tables) are in transit in links in cₙ′.

Proposition 9.5 makes precise the “almost unique” property alluded to above. The normal form property suggested by Proposition 9.5 motivates a notion of equivalence “up to stabilization”, defined below.

**Definition 9.6** (≡₁). Define a relation R₁ on type 2 configurations such that cn₁ R₁ cn₂ whenever graph(cn₁) = graph(cn₂), t(cn₁) = t(cn₂), o(cn₁) ≡₁ o(cn₂), and m(cn₁) = m(cn₂). Let cn₁ ≡₁ cn₂ if cn₁ and cn₂ are WF2 and there are cn₁’ and cn₂’ such that A₁(cn₁) ⇝ cn₁’ R₁ cn₂’ ⇝ A₁(cn₂).

Proposition 9.5, together with the termination of A₁, allows the existential quantifiers in Definition 9.6 to be exchanged for universal ones.

**Corollary 9.7.** If A₁(cn) ⇝ cn’, then cn ≡₁ cn’.  

**Proof.** We have A₁(cn) ⇝ cn’ R₁ cn’ ⇝ A₁(cn’).

**Lemma 9.8.** ≡₁ is reduction closed.

**Proof.** See Appendix 3.

**Lemma 9.9.** ≡₁ is context closed.

**Proof.** See Appendix 3.

**Proposition 9.10.** ≡₁ is a type 2 witness relation.

**Proof.** See Appendix 3.

**Corollary 9.11.** If A₁(cn) ⇝ cn’, then cn ≍₂ cn’.

**Proof.** By Proposition 9.10 and Corollary 9.7.

### 9.2 Normalization

We turn to the second procedure, which empties object queues, propagates futures, and migrates all objects and their tasks to a single node. The procedure, Algorithm 2, is shown in Figure 11. Write A₂(cn) ⇝ cn’ if cn’ is a possible result of applying Algorithm 2 to cn. Initially, a node u is chosen towards which all objects will migrate during normalization. Normalization is then performed in cycles, with each cycle starting and ending in a configuration in stable form. In each cycle, one message is read from an object’s in- and out-queues. By well-formedness, object queues contain only call and future messages. Receptions of future messages may cause object environments to instantiate futures. This may cause new future instantiation messages to be enabled. Accordingly, those messages are generated and delivered to an object out-queue. Once this is done, objects not yet at u will be migrated.

**Proposition 9.12.** Algorithm 2 terminates.
Algorithm 2: Normalization

**Input** A type 2 well-formed configuration $cn$

**Output** A configuration in normal form, reachable from $cn$

fix a NID $u$ for a node in $cn$ ;
run Algorithm 1 ;
repeat
   while some object queue is nonempty
      use MSG-SEND, MSG-DELAY-2, CALL-RCV-2, or FUT-RCV to dequeue one
      message from each nonempty object queue ;
      repeat
         use FUT-SEND to send future instantiation messages
      until FUT-SEND can no longer be used
   end ;
   while an object $o$ exists not located at $u$
      use OBJ-SEND to send $o$ towards $u$
   end ;
run Algorithm 1
until all objects are located at $u$, all object queues are empty, and there are
only external link messages

Figure 11: Algorithm 2

**Proof.** See Appendix 3.

We next turn to normal forms, and define first a couple of auxiliary operations.

Let $t_2(cn)$ be the multiset of task containers $tsk = t(o, l, s)$ such that

- $tsk \leq cn$, or
- there is a routable message $\text{call}(o, o', f, m, v)$ in transit in $cn$, such that
  $l = \text{locals}(o, f, m, v)$ and $s = \text{body}(o, m)$.

Let $m_2(cn)$ be the multiset of external messages in queues in $cn$. Let $o_2(cn)$
be the multiset of object containers $o(o, a, u', \epsilon, \epsilon)$ for which all of the following holds:

- $u' = u$.
- There is an object container $o(o, a', u'' , q_{in}, q_{out}) \leq cn$.
- $a(x) = a'(x)$, for all variables $x$, and $a(\text{self}) = a'(\text{self})$.
- $a(f) = (v, \epsilon)$ if $cn$ assigns $v$ to $f$, and $o$ is on the notification path of $f$
in $cn$.
- $a(f) \uparrow$, if $f$ is unassigned in $cn$, $a'(f) \uparrow$, and there is no $f'$ such that $cn$
  assigns $f$ to $f'$ and, additionally, $o$ is on the notification path of $f'$. 

[29]
• \( a(f) = (\bot, \emptyset) \) if \( f \) is unassigned in \( cn \), and \( a'(f) \downarrow \) or there is an \( f' \) such that \( cn \) assigns \( f \) to \( f' \) and, additionally, \( o \) is on the notification path of \( f' \), with \( o \) defined as follows. Let \( f_1, \ldots, f_n \) be all future assignments such that, for all \( f_i \), \( o \) is on the notification path of \( f_i \), and either \( cn \) assigns \( f_i \) to \( f \), or there is a sequence of assignments of futures \( f_j \) starting from \( f_i \) leading to \( f \) in \( cn \). Let \( o_i \) be the forwarding list such that \( o_i = \pi_2(\pi_1(a'(f_i))) \) if \( a'(f_i) \downarrow \), and \( o_i = \varepsilon \) if \( a'(f_i) \uparrow \). Let \( \bar{\sigma} \) be the forwarding list such that \( \bar{\sigma} = \pi_2(\pi_1(\pi_2(\pi_1(\pi_2(\pi_1(a'(f_i))))))) \) if \( a'(f) \downarrow \), and \( \bar{\sigma}' = \varepsilon \) if \( a'(f) \uparrow \). Then, if \( \text{call}(o, o', f, m, \bar{\sigma}) \preceq cn \) for some \( o', m, \) and \( \bar{\sigma} \), set \( \bar{\sigma} = \bar{\sigma}_1 \cdots \bar{\sigma}_n \bar{\sigma}' \), and set \( \bar{\sigma} = \bar{\sigma}_1 \cdots \bar{\sigma}_n \bar{\sigma}' \) otherwise.

The intent of these functions is to describe the effects of running Algorithm 2 on a configuration, the details of which we make precise in the following definition and proposition.

**Definition 9.13 (Normal Form).** A WF2 configuration \( cn \) is in normal form, if

1. \( cn \) has stable routing,
2. \( t(cn) = t_2(cn) \),
3. \( o(cn) = o_2(cn) \), and
4. \( m(cn) = m_2(cn) \).

**Proposition 9.14.** If \( cn \) is WF2 and \( A_2(cn) \rightsquigarrow cn' \), then the following holds:

1. \( cn' \) is reachable from \( cn \).
2. \( cn' \) is in normal form.
3. \( \text{graph}(cn) = \text{graph}(cn') \).
4. \( t_2(cn) = t(cn') \).
5. \( o_2(cn) = o(cn') \).
6. \( m_2(cn) = m(cn') \).

**Proof.** See Appendix 3.

Proposition 9.14 motivates the following definition of normal form equivalence.

**Definition 9.15 (\( \equiv_2 \)).** Define a relation \( R_2 \) on type 2 configurations such that \( cn_1 R_2 cn_2 \) whenever \( \text{graph}(cn_1) = \text{graph}(cn_2), t(cn_1) = t(cn_2), o(cn_1) = o(cn_2), \) and \( m(cn_1) = m(cn_2) \). Let \( cn_1 \equiv_2 cn_2 \) if \( cn_1 \) and \( cn_2 \) are WF2 and there are \( cn_1' \) and \( cn_2' \) such that \( A_2(cn_1) \rightsquigarrow cn_1' R_2 cn_2' \rightsquigarrow A_2(cn_2) \).

Clearly, \( \equiv_2 \) relates more extended configurations than \( \equiv_1 \).

**Corollary 9.16.** \( \equiv_1 \subseteq \equiv_2 \).
Proof. If $cn_1 \equiv_1 cn_2$, the two configurations have the same task containers, and the same object bound messages. In addition, there is a one-to-one mapping between object containers where identifiers and object environments coincide. The result follows by noting that any remaining differences between the containers will disappear after running Algorithm 2.

We also obtain that normalization respects normal form equivalence.

**Corollary 9.17.** If $A_2(cn) \leadsto cn'$, then $cn \equiv_2 cn'$.

Proof. We have $A_2(cn) \leadsto cn' R_2 cn' \leadsto A_2(cn')$.

The proof of reduction closure follows that of Lemma 9.8 quite closely.

**Lemma 9.18.** $\equiv_2$ is reduction closed.

Proof. See Appendix 3.

**Lemma 9.19.** $\equiv_2$ is context closed.

Proof. See Appendix 3.

**Proposition 9.20.** $\equiv_2$ is a type 2 witness relation.

Proof. Similar to the proof of Proposition 9.10.

**Corollary 9.21.** If $A_2(cn) \leadsto cn'$, then $cn \simeq_2 cn'$.

Proof. None of the rules used in Algorithm 2 affect the shape of the normal form. Thus, if $A_2(cn) \leadsto cn'$, then $cn \equiv_2 cn'$. But then, $cn \simeq_2 cn'$ by Proposition 9.20.

**10 Correctness**

In this Section, we prove the correctness of the network-aware semantics by mapping a well-formed type 1 configuration $\text{bind } z$ in standard form to a well-formed type 2 configuration $\text{net}(cn)$ close to normal form, with an arbitrary, but well-formed, underlying network graph. After establishing that the mapping $\text{net}$ produces reasonable configurations, we prove that the two configurations are contextually equivalent.

**10.1 The Representation Map**

We first fix a well-formed network graph represented as a configuration $cn_{\text{graph}}$, containing a node with distinguished NID $u_0$, similarly to the way initial configurations are defined in Section 6. Thus, $cn_{\text{graph}}$ consists of nodes and links only, with each node $u$ in $cn_{\text{graph}}$ having the form $n(u,t)$, and each link having the form $l(u,\varepsilon,u')$. The routing tables $t$ are defined later.
Representing Names and Values  To represent names, one complication is that names in the type 1 semantics need to be related to identifiers in the type 2 semantics, which does not include the binding construct of the type 1 semantics, but on the other hand has two generator functions (the operations newf and newo). This prevents the name relation from being modeled using the identity relation. To address this, we assume that names in the type 1 semantics are really symbolic, connected to concrete identifiers used in the type 2 semantics by means of an injective name representation map \( \text{rep} \), taking internal names \( f, o \) in the type 1 semantics to names \( \text{rep}(f), \text{rep}(o) \) in the type 2 semantics. For convenience, we extend the name representation map \( \text{rep} \) to arbitrary values and task environments as follows:

- \( \text{rep}(\text{ext}) = \text{ext} \)
- \( \text{rep}(p) = p, \text{if } p \in \text{PVal} \)
- \( \text{rep}(l)(x) = \text{rep}(l(x)) \)
- \( \text{rep}(l)(\text{ret}) = \text{rep}(l(\text{ret})) \)

Representing Object Environments  A problem in extending \( \text{rep} \) to object environments is that object environments in the type 2 semantics must be defined partially in terms of the type 1 environments (for object variables), and partially in terms of the future containers available in the “root configuration”, since the type 1 semantics uses future containers in place of forwarding lists stored in object environments. To this end, we first define an auxiliary operation \( \text{oenvmap}(cn, \wp, \text{rep}) : \text{FID} \rightarrow \text{Val}_\perp \) on triples of type 1 configurations \( cn \), pools \( \wp \) of OID/FID constants, and name representation maps \( \text{rep} \), as a function which gathers together assignments to futures as determined by the future containers in \( cn \):

- \( \text{oenvmap}(0, \wp, \text{rep})(f) = \perp \)
- \( \text{oenvmap}(\text{tsk}, \wp, \text{rep})(f) = \perp \)
- \( \text{oenvmap}(\text{obj}, \wp, \text{rep})(f) = \perp \)
- \( \text{oenvmap}(\text{call}, \wp, \text{rep})(f) = \perp \)
- \( \text{oenvmap}(f(f, v_\perp), \wp, \text{rep})(f') = \text{if } \text{rep}(f) = f' \text{ then } \text{rep}(v_\perp) \text{ else } \perp \)
- \( \text{oenvmap}(\text{bind } o. cn, \wp \cup \{o'\}, \text{rep})(f) = \text{oenvmap}(cn, \wp, \text{rep}[o'/o])(f) \)
- \( \text{oenvmap}(\text{bind } f.cn, \wp \cup \{f'\}, \text{rep})(f'') = \text{oenvmap}(cn, \wp, \text{rep}[f'/f])(f'') \)
- \( \text{oenvmap}(cn_1 cn_2, \wp, \text{rep})(f) =\text{oenvmap}(cn_1, \wp, \text{rep})(f) \sqcup \text{oenvmap}(cn_2, \wp, \text{rep})(f) \)

Fix now a root type 1 configuration \( cn_0 \) and a large enough pool \( \wp_0 \) of names (proportional to the size of \( cn_0 \), and computed using newf\( (u_0) \) and newo\( (u_0) \) to conform to our naming policy). Assume that \( cn_0 \) is in standard form, i.e., \( cn_0 = \text{bind } \overline{\text{z}}.cn_0' \) where \( cn_0' \) does not have binders. Fix \( g = \text{oenvmap}(cn_0, \wp_0, \perp) \) and \( cn_{\text{graph}} \) as above. We can now extend \( \text{rep} \) to object environments as follows:
\[\pi_1(\text{rep}(a))(\text{self}) = \text{rep}(\pi_1(a))(\text{self})\]
\[\pi_1(\text{rep}(a))(x) = \text{rep}(\pi_1(a))(x)\]
\[\pi_2(\text{rep}(a))(f) = \begin{cases} (v, \varepsilon) & \text{if } g(f) = v \\ (\perp, \text{OID}(cn_0)) & \text{otherwise} \end{cases}\]

Since we have left the nature of expressions unspecified, we need to additionally assume that the representation map commutes with the expression semantics, i.e., for all \(\epsilon, a,\) and \(l\), we assume the following equation holds:

\[\text{rep}\left([e]_{(a,l)}\right) = [\text{rep}(a),\text{rep}(l)]\]  \hspace{1cm} (2)

**Proposition 10.1.** Let \(cn_0\) be a type 1 well-formed root configuration in standard form, and \(\wp_0\) be a pool as above. Then, \(\text{rep}(a)(f) = (v, \varepsilon)\) if and only if \(f(f,v) \preceq cn_0\).

**Proof.** By well-formedness, the future container, if it exists, is unique. Pick a name representation map \(\text{rep}\). Then, \(\text{envmap}(cn_0, \wp_0, \text{rep})(f)\) is defined and equal to \(v\) if and only if \(f(f,v) \preceq cn_0\). This is easily seen by induction on the structure of \(cn_0\). \(\square\)

**Representing Call Containers** Another complication is that we need an operation to represent a type 1 call container as a message in the type 2 semantics. Compared to type 1 call containers, type 2 call messages additionally contain the OID of the caller, which is added to the future forwarding list when the message is received. Since future forwarding lists are already extended maximally, it is possible to use the callee’s OID as the caller OID when mapping containers to messages without affecting behaviour, and put the message in the self-loop queue for the node \(u_0\). This is done in the obvious way by the operation \(\text{send}\), as follows:

\[
\text{send}(\text{call}(o,o',f,m,v), l(u_0, q, u_0) cn) = l(u_0, \text{enq}(\text{call}(o,o',f,m,v)), q, u_0) cn
\]

**Representing Configurations** Given a name representation map \(\text{rep}\), we can now define the representation of a type 1 configuration as a transformer on type 2 configurations with the mapping \(\text{net}\), as follows:

\[
\begin{align*}
\text{net}(cn_1, cn_2, \text{rep}) &= \text{net}(cn_1, \text{rep}) \circ \text{net}(cn_2, \text{rep}) \\
\text{net}(0, \text{rep})(cn) &= cn \\
\text{net}(t(o, l, s), \text{rep})(cn) &= t(\text{rep}(o), \text{rep}(l), s) cn \\
\text{net}(o(o, a), \text{rep})(cn) &= o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) cn \\
\text{net}(f(f, v_\perp), \text{rep})(cn) &= cn \\
\text{net}(c(o, f, m, \overline{\tau}), \text{rep})(cn) &= \\
&= \text{send}(\text{call}(\text{rep}(o), \text{rep}(a), \text{rep}(f), m, \text{rep}(\overline{\tau})), cn)
\end{align*}
\]
In other words, we represent type 1 configurations by first assuming some underlying network graph, and then distributing the (centralized) assignments to futures in each object environment with the trivial forwarding lists, and then, once this is done, mapping the containers individually to the type 2 level.

**Defining Routing Tables** The only detail remaining to be addressed from above concerns the routing tables. For the node with NID $u_0$, the initial routing table, $t_0$, needs to have all objects in $cn_0$ registered, i.e.,

$$t_0 = \text{reg}(\text{rep}(o_1), u_0, \text{reg}(\text{rep}(o_2), u_0, \text{reg}(\cdots, \text{reg}(\text{rep}(o_n), u_0, \bot) \cdots )))$$

where $\text{OID}(cn_0) = \{o_1, \ldots, o_n\}$. For nodes $n(u, t)$ where $u \neq u_0$, we let $t$ be determined by some stable routing. This is easily computed using Algorithm 1, and we leave out the details.

**Defining the Map** We can now finally define the representation map.

**Definition 10.2** (Representation Map $\text{net}$). Let a network graph configuration $cn_{\text{graph}}$ and a name representation map $\text{rep}$ be given for a WF1 configuration $\pi.cn$ in standard form. Then, the type 2 representation of the configuration is $\text{net}(cn) = \text{net}(cn, \text{rep})(cn_{\text{graph}})$.

The most basic property that we expect to hold about the representation map is that it produces type 2 well-formed configuration given well-formed type 1 configurations.

**Proposition 10.3.** If $\pi.cn$ is a WF1 configuration in standard form, then $\text{net}(cn)$ is WF2.

**Proof.** See Appendix 4.

### 10.2 Contextual Equivalence under Extension

Note that in the representation map, future-value mappings and forwarding lists are overapproximated when compared to the type 2 semantics, where future instantiations are only forwarded to and instantiated at objects that are on the notification path of the associated futures. We need to ensure that observable behavior does not change for well-formed configurations when future maps are extended in this way. To do this, we make precise the notion of extension for objects, and ultimately prove that configurations are contextually equivalent to their extended counterparts. Thus, a configuration produced by the mapping captures the behavior of a minimally extended type 2 configuration produced from an initial configuration.

**Definition 10.4** (Forwarding Resolved). We say that $o'$ is forwarding resolved for the future $f$ in the configuration $cn$ at the object $o$, if there is an object container $o(a, u, q_{in}, q_{out}) \preceq cn$, such that either $o' \in \pi_2(a(f))$, or one of the following holds:
1. $cn$ assigns $f$ to $p \in PVal$ and there is a container $o(o', a', u', q_{in}, q_{out}) \preceq cn$ such that $\pi_1(a'(f)) = p$.

2. $cn$ assigns $f$ to $f' \in FID$ and there is a container $o(o', a', u', q_{in}, q_{out}) \preceq cn$ such that $\pi_1(a'(f)) = f'$ and, additionally, $o'$ is forwarding resolved for $f'$ in $cn$ at $o$.

**Definition 10.5** (Extended Object Container). Let $cn$ and $cn'$ be configurations. An object container $o(a, u, q_{in}, q_{out}) \preceq cn$ extends an object container $o(o', a', u', q_{in}', q_{out}') \preceq cn'$ if $o = o'$, $u = u'$, $q_{in} = q_{in}'$, $q_{out} = q_{out}'$, and

1. $a'(self) = a(self)$, and for variables $x$, $a'(x) = a(x)$,
2. if $\pi_1(a'(f)) = v$, then $\pi_1(a(f)) = v$,
3. if $o'' \in \pi_2(a'(f))$, then $o''$ is forwarding resolved for $f$ in $cn$ at $o$,
4. if $\pi_1(a(f)) = v$, then $cn'$ assigns $v$ to $f$, and if additionally $v \in FID$, then $o$ is on the notification path of $v$ in $cn$, and
5. if $o'' \in \pi_2(a(f))$, then either $o'' \in \pi_2(a'(f))$ or both $o'' \in OID(cn')$ and $o$ is on the notification path of $f$ in $cn$.

**Definition 10.6** (Extended Configuration). A configuration $cn$ extends another configuration $cn'$, written $cn \preceq cn'$, if there is a one-to-one mapping of object containers $obj \preceq cn$ to object containers $obj' \preceq cn'$, such that $obj$ extends $obj'$, and additionally, $\text{graph}(cn) = \text{graph}(cn')$, $\text{t}(cn) = \text{t}(cn')$, and $\text{m}(cn) = \text{m}(cn')$.

**Definition 10.7** (Notification Path Traversal). Let $cn$ be a configuration such that $o$ is on the notification path of $f$ by $o'$ in $n$ steps. Then, this notification path traverses $o_{path}$ if

1. $n = 0$, $o' = o$, and $o_{path} = o$.
2. $n = 1$, $o' = o$, there is a future message $\text{future}(o, f, v) \preceq cn$, and $o_{path} = o$.
3. $n = 1$, $o' = o$, there is a task $t(o, l, s) \preceq cn$, and $o_{path} = o$.
4. $n = 2$, $o' = o$, there is a call message $\text{call}(o, o'', f, m, \tau) \preceq cn$, and $o_{path} = o$.
5. $n = 4$, and there is a call message $\text{call}(o', o, f, m, \tau) \preceq cn$, and $o_{path} = o'$ or $o_{path} = o'$.
6. $n = n' + 2$, and there is an object $o(o'', a', u', q_{in}, q_{out}) \preceq cn$ such that $o \in \pi_2(a'(f))$, and $o''$ is on the notification path of $f$ by $o'$ in $n'$ steps, and either $o_{path} = o''$, or the notification path from $o''$ to $o'$ traverses $o_{path}$.
7. \( n = 2n' + n'' \), and \( o \) is on the notification path of \( f' \) by \( o'' \) in \( n' \) steps, \( c_n \) assigns \( f \) to \( f' \), and \( o'' \) is on the notification path of \( f \) by \( o' \) in \( n'' \) steps, and \( o_{path} \) either traverses the path of \( f' \) from \( o \) to \( o'' \), or the path of \( f \) from \( o'' \) to \( o' \).

**Proposition 10.8.** Let \( cn_1 \) and \( cn_2 \) be WF2 configurations such that \( cn_1 \succeq cn_2 \). Then, if \( o \) is on the notification path of \( f \) by \( o_2 \) in \( n \) steps in \( cn_2 \), \( o \) is on the notification path of \( f \) by some \( o_1 \) in at most \( n \) steps in \( cn_1 \), such that the notification path from \( o \) to \( o_2 \) in \( cn_2 \) traverses \( o_1 \).

**Proof.** See Appendix 4.

**Proposition 10.9.** If \( cn_1 \) and \( cn_2 \) are WF2 configurations such that \( cn_1 \succeq cn_2 \) and there are \( cn_1' \) and \( cn_2' \) such that \( A_2(cn_1) \leadsto cn_1' \), and \( A_2(cn_2) \leadsto cn_2' \), then \( cn_1' \succeq cn_2' \).

**Proof.** See Appendix 4.

We define a binary relation, \( \equiv_2 \), that is a slight weakening of \( \equiv_2 \), and therefore includes \( \equiv_2 \), which relates WF2 configurations that, after running Algorithm 2, yield a pair of configurations such that one extends the other.

**Definition 10.10** (\( \equiv_2 \)). Let \( cn_1 \equiv_2 cn_2 \) if \( cn_1 \) and \( cn_2 \) are WF2 configurations and there are \( cn_1' \) and \( cn_2' \) such that it is either the case that \( A_2(cn_1) \leadsto cn_1' \succeq cn_2' \leadsto A_2(cn_2) \), or \( A_2(cn_2) \leadsto cn_2' \succeq cn_1' \leadsto A_2(cn_1) \).

**Lemma 10.11.** \( \equiv_2 \) is reduction closed.

**Proof.** See Appendix 4.

**Lemma 10.12.** \( \equiv_2 \) is context closed.

**Proof.** See Appendix 4.

**Proposition 10.13.** \( \equiv_2 \) is a type 2 witness relation.

**Proof.** See Appendix 4.

**Lemma 10.14.** Suppose that \( cn' \) is WF2, and \( cn \succeq cn' \). Then, \( cn \) is WF2 as well, and \( cn \simeq_2 cn' \).

**Proof.** See Appendix 4.

### 10.3 Correctness Result

We obtain a key lemma allowing us to relate transitions in the network-oblivious and the network-aware semantics under \( \simeq_2 \).

**Lemma 10.15.** Let bind \( \exists cn \) be a WF1 configuration in standard form.

1. If bind \( \exists cn \to bind \exists' cn' \), then for some \( cn'' \), \( net(cn) \to^* cn'' \simeq_2 net(cn') \).
2. If $\text{net}(cn) \rightarrow cn''$, then for some $\bar{x}$ and $cn'$, bind $\bar{x}.cn \rightarrow^* \text{bind } \bar{x}.cn'$ and $cn'' \preceq_2 \text{net}(cn')$.

Proof. See Appendix 4.

We can now state the main result.

**Theorem 10.16** (Correctness of the Type 2 Semantics). For all well-formed type 1 configurations bind $\bar{x}.cn$ in standard form, bind $\bar{x}.cn \simeq \text{net}(cn)$.

Proof. See Appendix 4.

\section{11 Scheduling}

The type 2 semantics is highly nondeterministic. The semantics says nothing about how frequently routing tables are to be exchanged, when messages should be passed between the different queues, when future messages are to be sent, or when, and to where, objects are to be transmitted. Resolving these choices is a crucial tradeoff between management overhead and performance. For instance, if routing tables are exchanged at a very high frequency, routing can be always assumed to be in stable state. This ensures short end-to-end routes, but at the expense of a large management and messaging overhead. This raises the question of how to determine these parameters, something which we address in more detail in related work [29].

Regardless of how, a real implementation needs to resolve these choices. This is tantamount to eliminating nondeterminism from the type 2 semantics, essentially by removing transitions. Thus, in a sense, Theorem 10.16 achieves more than is called for, as soundness and full abstraction a priori applies only to the type 2 semantics with all transitions included.

We view a scheduler abstractly as a predicate on histories in the following way. Let a scheduled execution be any sequence $\rho = cn_1 \cdots cn_n$ such that $cn_i \rightarrow cn_{i+1}$ for all $i : 1 \leq i < n$ where all $cn_i$ are well-formed type 2 configurations. A scheduler is a predicate $S$ on such sequences, with the property that

1. $S(\langle cn \rangle)$ for all configurations $cn$, where $\langle cn \rangle$ is the execution consisting only of $cn$ (a scheduler kicks in only when an execution is started), and

2. if $S(cn_1 \cdots cn_n)$ and $cn_n \rightarrow cn_{n+1}$ for some $cn_{n+1}$, then it holds that $S(cn_1 \cdots cn_n cn_{n+1})$ for exactly one $cn_{n+1}$.

Then, a scheduled type 2 semantics is a transition system on executions $\rho = cn_1 \cdots cn_n$ such that $\rho \rightarrow \rho'$ if and only if $\rho' = cn_1 \cdots cn_n cn_{n+1}$ and $S(\rho')$.

Define now the barbed simulation preorder $\preceq$ on executions by requiring the existence of a witness relation $\mathcal{R}$ which satisfies reduction closure, context closure, and barb preservation (where $cn_1 \cdots cn_n \downarrow \text{obs}$ whenever $cn_n \downarrow \text{obs}$) but not necessarily the converse properties. We immediately obtain from Theorem 10.16 the following corollary:
Corollary 11.1. For all well-formed type 1 configurations bind $\pi.cn$ in standard form,

$$\langle \text{net}(cn) \rangle \sqsubseteq \langle \text{bind } \pi.cn \rangle.$$  

Proof. It suffices to note that if $\rho = cn_1 \ldots cn_n \rightarrow \rho' = cn_1 \ldots cn_{n+1}$, then $cn_n \rightarrow cn_{n+1}$, and if $\rho \downarrow \text{obs}$, then $cn_n \downarrow \text{obs}$ as well. \qed

12 Concluding Remarks

The contribution of the present paper has been to show that, using location independent routing, it is possible to devise novel and elegant network-based execution models for object-oriented languages with fairly sophisticated features such as futures, and with attractive properties regarding correctness, performance, and scalability.

This paper focuses on correctness. In other ongoing work [29], we study the use of the model presented here adapted in detail to the ABS language core [23], to investigate decentralized runtime adaptability of objects. The language in our work on adaptability is more practical in that expressions and expression evaluation are both fully defined, variable types are declared, and it includes a type system which guarantees that well-typed programs have safety properties that go beyond well-formedness as defined here. Another difference is that the unit of concurrency is a single object rather than a task, which gives a more intuitive programming model that avoids data races. However, as demonstrated by Henrio et al. [19], an object model with concurrent tasks can still be easy to use and be efficiently executed. Our adaptability-oriented network-aware semantics differs from the type 2 semantics in that network and object configurations are completely separate, but can perform labelled transitions, and synchronize on complementary labelled transitions to transfer data between a node and an object. This makes the interfaces between the network-layer and the object-layer much more clear than in the present work. We have also developed a Java-based simulator which implements the adaptability-oriented network-aware semantics and provides various ways of controlling object migration. We report on experimental results on decentralized performance adaptation for load balancing and latency management, with promising results.

The correctness analysis is based on contextual equivalence, similar to other past work mostly belonging to the $\pi$-calculus school of process algebra [8, 15, 16]. A closely related precursor is Nomadic Pict [36]. In comparison with that work, we obtain a simpler and in our opinion more elegant correctness treatment, chiefly because our solution obviates the need for locking and consequently preemption, which has well-known detrimental consequences in a bisimulation-oriented setting. Other related works include JoCaml [10], which also uses forward chaining, along with an elaborate mechanism to collapse the forwarding chains. In the Klaim project [1], compilers are 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 lan-
guage [37] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [26] 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 the Ambient Calculus [3]). Past correctness analyses for languages with futures have been carried out, e.g., by Caromel et al. [7], who prove a confluence result for their language of asynchronous sequential processes, however without an explicit treatment of distribution, communication, and routing.

Substantial work has been going on in the HATS project on the ABS language and its extensions, for instance towards software product lines [34]. Johnsen et al. [24] have studied an extension of ABS with deployment components that can be used as a reflection mechanism for explicit performance management. The use of reflection allows many aspects of resource control to be brought into the ABS framework and managed as part of the regular application development process. By contrast, our focus is on algorithmics and the automation of the performance adaptation processes, and for this type of application reflection is not a main concern. The cobox concurrency programming model described by Schäfer [35], closely related to ABS, has a Java implementation which handles message passing to remote objects via RMI, but which does not support object mobility.

Scalability is not fully resolved in the present work. We use a rather naïve distance vector routing scheme based on Bellman-Ford. Distance vector routing has unit stretch but is not compact: Routing tables need to contain on the order of one entry per object identifier in the system. This is no problem for networks of moderate size, but for scalability, other routing schemes are needed, as outlined in the introduction. The Bellman-Ford scheme has other well-known drawbacks that arise in the case of intermittent network partitioning.

Besides more scalable and robust routing, we see two main directions for future work. The first is to examine richer language semantics, specifically with respect to more dynamicity. In ongoing work, we are studying power control: Adding an explicit knob to the network-aware semantics for turning nodes on and off. Further down the line, it is of interest to consider both crash failures and Byzantine failures. The second, parallel, avenue is to study performance adaptation in richer and more realistic settings. In our work on adaptability [29], our only management knob is object migration, and the management objective is to obtain good load balancing combined with good clustering properties. However, a real implementation, in particular when combined with network dynamicity, will have many more management knobs such as buffer size, processor load, and power control. How to effectively control object network performance in such a multi-dimensional setting is a significant future challenge.
References


Appendix 1: Proofs for Section 5

Proposition 5.2. The identity relation is a type 1 witness relation. Contextual equivalence is a type 1 witness relation. If \( R, R_1, R_2 \) are type 1 witness relations then so is

1. \( R^{-1} \),
2. \( R^* \), and
3. \( R_1 \circ R_2 \circ R_1 \).

Proof. The identity relation is trivially reduction closed, context closed, and barb preserving, and is its own converse. To prove \( \simeq_1 \) is a type 1 witness relation, suppose \( cn_1 \simeq_1 cn_2 \). Then, \( cn_1 R cn_2 \) for some type 1 witness relation. If \( cn_1 \rightarrow cn'_1 \), we have \( cn'_2 \) such that \( cn_2 \rightarrow^* cn'_2 \) and \( cn'_1 R cn'_2 \). But then \( cn'_1 \simeq cn'_2 \), and we have shown reduction closure. For context closure, assume \( cn_1 cn \) is WF1; then \( cn_1 cn R cn_2 cn \) and consequently \( cn_1 cn \simeq_1 cn_2 cn \). For barb preservation, if \( cn_1 \downarrow obs \), then \( cn_2 \downarrow obs \), since \( R \) is barb preserving. The converse arguments are completely symmetric. 

For property 1, it suffices to note that \( (R^{-1})^{-1} = R \). Reflexive, transitive closure in property 2 follows by a straightforward inductive argument. For property 3, if \( cn_1 R_1 \circ R_2 \circ R_1 cn_2 \) then \( cn_1 R_1 cn_2 \circ R_2 cn_1 R_1 cn_2 \) for some \( cn_{1,2}, cn_{2,1} \). For reduction closure, assume \( cn_1 \rightarrow cn'_1 \). Then, \( cn_{1,2} \rightarrow^* cn'_{1,2} \) and \( cn'_1 R_{1,2} cn'_{2,1} \). But then, \( cn_{2,1} \rightarrow^* cn'_{2,1} \) and \( cn'_{1,2} R_{2,1} cn'_{2,1} \). Consequently, \( cn_2 \rightarrow^* cn'_2 \) and \( cn'_{2,1} R_{1,2} cn'_2 \), whereby \( cn'_1 R_1 \circ R_2 \circ R_1 cn'_2 \). For context closure, assume \( cn_1 cn \) is WF1. Then, \( cn_{1,2} cn \) is WF1 and \( cn_1 cn R cn_{1,2} cn \), and so on, yielding that \( cn_{2,2} cn \) is WF1 and \( cn_1 cn R_1 \circ R_2 \circ R_1 cn_2 cn \). For barb preservation, assume \( cn_1 \downarrow obs \). Then, \( cn_{1,2} \rightarrow^* cn'_{1,2} \downarrow obs \), allowing us to find \( cn'_{2,2} \) such that \( cn'_{1,2} R_{1,2} cn'_{2,2} \), whereby \( cn' \rightarrow^* cn'_{2,2} \downarrow obs \), and so on, yielding that \( cn_2 \downarrow obs \), as needed. Again, the converse arguments are symmetric.

\( \square \)

Appendix 2: Proofs for Section 7

Lemma 7.4. Fix a configuration \( cn \) and an object \( o(a, o, u, q_{in}, q_{out}) \leq cn \). If \( o \) is on the notification path of \( f \) by \( o' \) in \( n \) steps in the configuration \( cn \) and \( cn \rightarrow cn' \), then \( o \) is on the notification path of \( f \) by \( o' \) in at most \( n \) steps in \( cn' \).

Proof. The proof is by induction on \( n \). We follow the case analysis in Definition 7.3.

Case 1: We obtain that \( \pi_1(a(f)) \downarrow \), and then, by Proposition 6.2.3, we find \( o(a, a', u', q_{in}', q_{out}') \leq cn' \) such that \( \pi_1(a'(f)) \downarrow \).

Case 2: Either future(o, f, v) \leq cn' or we find o(a, a', u', q_{in}', q_{out}') \leq cn' such that \( \pi_1(a'(f)) \downarrow \).

Case 3: There are two options by Proposition 6.2.4: Either we find a task \( t(a, l', s') \leq cn' \) with \( l'(\text{ret}) = f \), or else we find o(a, a', u', q_{in}', q_{out}') such that \( \pi_1(a'(t(\text{ret}))) \downarrow \).
Case 4: Either \( \text{call}(o,o',f,m,\pi) \preceq cn' \) or we find \( t(o,l,s) \preceq cn' \) with \( l(\text{ret}) = f \).

Case 5: Either \( \text{call}(o',o,f,m,\pi) \preceq cn' \) or we find \( o(o',a',u',q_{\text{in}}',q_{\text{out}}') \preceq cn' \) such that \( t(o',l',s') \preceq cn' \) with \( l'(\text{ret}) = f \), and \( o \in \pi_2(a'(f)) \).

Case 6: We find an object \( \text{obj}'' = o(o'',a'',u'',q_{\text{in}}'',q_{\text{out}}'') \preceq cn' \) such that \( o'' \) is on the notification path of \( f \) by \( o' \) in \( n - 1 \) steps in \( cn' \). By Proposition 6.2.3, we find the derivative \( \text{obj}''' = o(o''',a''',u''',q_{\text{in}}''',q_{\text{out}}''') \preceq cn' \) of \( \text{obj}'' \), and by the induction hypothesis \( o'' \) is also on the notification path of \( f \) by \( o' \) in \( cn' \), now in some \( n'' \leq n - 2 \) steps. By inspection of the rules we see that either \( \pi_2(a''(f)) \) is a suffix of \( \pi_2(a'(f)) \), or else there is a message \( \text{future}(o,f,\pi_1(a''(f))) \preceq cn' \). In either case we can conclude.

Case 7: We find an object \( \text{obj}_1 = o(o_1,a_1,u_1,q_{\text{in}}_1,q_{\text{out}}_1) \preceq cn' \) such that \( o_1 \) is on the notification path of \( f' \) by \( o'' \) in \( n' \) steps in \( cn' \), and an object \( \text{obj}'' = o(o'',u'',q_{\text{in}}'',q_{\text{out}}'') \preceq cn' \) such that \( o'' \) is on the notification path of \( f \) by \( o' \) in \( n'' \) steps, with \( n = 2n' + n'' \), where \( f' \) is assigned to \( f \) in \( cn' \). In the next step, by the induction hypothesis, \( o'' \) can be on the notification path of \( f \) by \( o' \) in the same or fewer steps, and the length of the notification path of \( f' \) for \( o_1 \) does not increase as in the previous case. In either case we can conclude. \( \square \)

**Lemma 7.6.** Let \( cn \) be a configuration. Then, the following holds:

1. If \( cn \) is a type 2 initial configuration, then \( cn \) is WF2.

2. If \( cn \) is WF2 and \( cn \rightarrow cn' \), then \( cn' \) is WF2.

3. If \( cn \) is type 2 reachable, then \( cn \) is WF2.

**Proof.** Property 1 follows straightforwardly from Definition 6.3 and Definition 7.5. For property 2, we consider each transition rule in turn.

OID Uniqueness: In all rules except new-2 there is a one-to-one correspondence between object container occurrences in \( cn \) and object container occurrences in \( cn' \). This is sufficient to conclude. For new-2, it is sufficient to note that \( o' \) is a freshly generated OID.

Task-Object Existence, Object-Node Existence: The properties follow since neither nodes nor objects are ever removed. In the first instance, objects can only be created when the node is present, and in the second instance tasks can only be created when the object is present.

Buffer Cleanliness: We check that only object-bound messages enter in- and out-queues. This concerns the rules msg-rcv, call-send-2, fut-send, msg-delay-1, and msg-delay-2 only. The check is routine.

Local Routing Consistency: Easily proved by case analysis on rules.

Call Uniqueness: In all rules except call-send-2, call messages are the same in \( cn \) and \( cn' \), or present in \( cn \) and removed in \( cn' \). This is sufficient to conclude. For call-send-2, it is sufficient to note that \( f \) is a freshly generated FID.

Future Uniqueness: We only need to consider rules which assign a non-\( \perp \) value to futures. This happens in rules fut-send, fut-rcv and ret-2. The
former two rules are immediate, and for \textsc{ret-2} we use the assumption that \(cn\) satisfies 7.5.5.

Single Writer: Again, we only need to consider the rules \textsc{fut-send}, \textsc{fut-rcv} and \textsc{ret-2}. Since for the former two rules, \(cn\) assigns \(v\) to \(f\) if and only if \(cn'\) does so, only \textsc{ret-2} remains, which is immediate.

External OID: Assuming fresh identifiers for objects are never equal to \(ext\), the check is routine.

Future Liveness: Let \(o(o', o', u', q_{in}, q_{out})\) be the derivative of \(o(o, a, u, q_{in}, q_{out})\) in \(cn\), and assume that \(f\) is active for \(o\) for the configuration \(cn'\). Either \(a'(f)\downarrow\), or else a call message \(call(o', o, f', m, v)\) is in transit in \(cn'\) and \(f\) occurs in \(\tau\). We proceed by cases on the transition rule leading to \(cn'\). Any rule that does not directly affect any of the conditions in Definition 7.2 or Definition 7.3 immediately allows to conclude that \(f\) is active for \(o\) also in \(cn\). By the induction hypothesis, we can conclude that \(o\) is on the notification path of \(f\) in \(cn\), and then \(o\) is on the notification path of \(f\) also in \(cn'\), since the only exception in Lemma 7.4 is when the environment of \(o\) is updated. For the remaining rules, there are the following cases to consider:

\textsc{call-send-2}: Assume first that \(o\) is the sending object. Either \(f\) is the newly introduced future in which case \(o\) is on the notification path from \(f\) according to 7.3.5, since a call message is in transit from \(o\) to \(o'\) with return future \(f\). If \(f\) is another future which is active for \(o\) in \(cn\) then \(f\) is also active for \(o\) in \(cn\). By the induction hypothesis, \(o\) is on the notification path from \(f\) in \(cn\). Then \(o\) is also on the notification path from \(f\) in \(cn'\) by Lemma 7.4. On the other hand if \(o\) is not the sending object the case is immediately closed by the induction hypothesis, as the “pending” relation transfers from \(cn'\) to \(cn\). We then apply the IH to conclude that \(o\) is on the notification path from \(f\) in \(cn\), and then we use Lemma 7.4 to conclude that this also applies to \(cn'\).

\textsc{call-rcv-2}: Assume first that \(o\) is the object receiving the call, and that \(f\) is the future of the call. Then \(o\) is on the notification path from \(f\) by Definition 7.3.2. Another option is that \(f\) is a future in \(\tau\) (referring to the transition rule in Figure 9). Then \(f\) is active for \(o\) in \(cn\) as well, by Definition 7.2. If \(f\) is some other future the case is completed by the IH as above.

\textsc{fut-send}: Follows from the induction hypothesis and Lemma 7.4, as in the case for \textsc{call-send-2}.

\textsc{fut-rcv}: If \(f\) is active for \(o\) for the configuration \(cn'\) then \(f\) is active for \(o\) also for \(cn\), and \(f\) is not the received future. But then, the result follows by the IH and Lemma 7.4.

\textsc{ret-2}: If \(f\) is active for \(o\) for \(cn'\), then \(f\) is not the return future and we again complete by the induction hypothesis. Finally, property 3 holds by property 1 and repeated application of property 2.

\begin{proof}

\textbf{Appendix 3: Proofs for Section 9}

\textbf{Proposition 9.3.} \textit{Algorithm 1 terminates.}

\end{proof}
Proof. In each iteration of the outermost loop of Algorithm 1, exactly one message is enqueued on each proper link, and at least one message is dequeued from all link queues. MSG-RCV, MSG-DELAY-1, and OBJ-RCV cause messages to leave the link queues, except for external messages, which are moved to the self-loop queues. If the link queues contain only routing table messages, the algorithm terminates in that iteration. If not, there must be object messages or routable call messages in some link queue. Since no new object messages are enqueued, there must some number of iterations \( n_0 \) after which all object messages have been received via OBJ-RCV and the associated object OIDs \( o \) registered on some node \( u \) so that \( t(o) = (u, 0) \).

Let \( m_0 \) be the size of the largest link queue at the point which there are no object messages in transit. After \( n_0 + m_0 + 1 \) iterations, each node \( u \) has received at least one table update from each of its neighbors \( u' \), and the last table update applied to \( u \) has \( t(o) = 0 \). As a result, at point \( n_0 + m_0 + 1 \), each node \( u \) has \( t(o) = (u', 1) \) whenever the \( u' \) is the host of \( o \) and the minimal length path from \( u \) to \( u' \) has length 1. The entry of the routing table of \( u \) for \( o \) will not change from that point onwards. We say that those entries are stable. Proceeding, let \( m_1 \) be the size of the largest link queue 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 can only be applied a finite number of times before a routable message will be delivered. There is no chance of routable messages getting stuck in self-loop queues, since they are continuously shuffled using MSG-DELAY-3.

The only detail remaining to be checked is that a message can always be read from a link. Table and object messages can always be delivered, and call messages can also always be delivered, if nothing else to the self-loop link, in which case the routing table is not up-to-date or the message is external. This is the only case where MSG-DELAY-1 is used. This completes the argument. \( \square \)

Lemma 9.8. \( \equiv_1 \) is reduction closed.

Proof. Suppose \( cn_1 \equiv_1 cn_2 \), where \( cn_1 \) and \( cn_2 \) are WF2. Assume \( cn_1 \rightarrow cn'_1 \); we need to find \( cn'_2 \) such that \( cn_2 \rightarrow^* cn'_2 \) and \( cn'_1 \equiv_1 cn'_2 \). We proceed by case analysis on the transition \( cn_1 \rightarrow cn'_1 \), eliding uses of CTXT-1.

For the cases T-SEND, T-RCV, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, OBJ-RCV, MSG-DELAY-3, and OBJ-REG, we take \( cn'_2 = cn_2 \), since then, the stable form is unaffected, i.e., \( cn_1 \equiv_1 cn'_1 \), by Proposition 9.5.

The remaining cases include the rules for sequential control, MSG-SEND, MSG-DELAY-2, CALL-SEND-2, CALL-RCV-2, FUT-SEND, FUT-RCV, RET-2, GET-2, NEW-2, and OBJ-SEND. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition of the form

\[
\begin{align*}
    cn \circ (o, a, u, q_{in}, q_{out}) t(o, l, x = e; s) \rightarrow cn \circ (o, a[v/x], u, q_{in}, q_{out}) t(o, l, s)
\end{align*}
\]

where \( [v]_{(a, l)} = v \) and \( x \in \text{dom}(a) \). Consider \( cn''_2 \) such that \( \mathcal{A}_1(cn_2) \rightarrow cn''_2 \). By
Proposition 9.5, there is a task \( t(o, l, x = c; s) \) and an object \( o(a, u, q_{in}, q_{out}) \) in \( cn''_2 \). Hence, it is possible to perform a transition

\[
 cn' o(a, u, q_{in}', q_{out}) t(o, l, x = c; s) \rightarrow cn' o(a[v/x], u, q_{in}', q_{out}) t(o, l, s)
\]

and we have

\[
 cn o(a, u[v/x], u, q_{in}, q_{out}) t(o, l, s) \equiv_1 cn' o(a, u[v/x], u, q_{in}', q_{out}) t(o, l, s)
\]

as needed, setting \( cn''_2 \) to the right-hand side.

**call-send-2:** Consider a transition of the form

\[
 cn o(a, a, u, q_{in}, q_{out}) t(o, l, x = e!m(t); s) \rightarrow cn o(a, a', u, q_{in}, \text{enq}(msg, q_{out})) t(o, l[f/x], s)
\]

where \( \overline{v} = [v/a], f = \text{new}(u), msg = \text{call}(o', f, m, \overline{v}), o' = \{e\} \) (a, l), and \( a' = \text{init}(f, a) \). Consider \( cn''_2 \) such that \( A_1(cn_2) \Rightarrow cn''_2 \). By Proposition 9.5, there is a task \( t(o, l, x = e!m(t); s) \) and an object \( o(a, u, q_{in}, q_{out}) \) in \( cn''_2 \). Hence, it is possible to perform a transition

\[
 cn' o(a, a, u, q_{in}', q_{out}) t(o, l, x = e!m(t); s) \rightarrow cn' o(a, a', u, q_{in}', \text{enq}(msg, q_{out})) t(o, l[f/x], s)
\]

and we have

\[
 cn o(a, a', u, q_{in}, \text{enq}(msg, q_{out})) t(o, l[f/x], s) \equiv_1 cn' o(a, a', u, q_{in}', \text{enq}(msg, q_{out})) t(o, l[f/x], s)
\]

setting \( cn''_2 \) to the right-hand side. The remaining cases are proved in a similar straightforward way, by mimicking the transition after applying Algorithm 1.

\[\square\]

**Lemma 9.9.** \( \equiv_1 \) is context closed.

**Proof.** Assume \( cn_1 \equiv_1 cn_2 \) and \( cn_1 cn \) is WF2. We first show that \( cn_2 cn \) is WF2 as well.

**OID Uniqueness:** If \( obj_1, obj_2 \leq cn_2 cn \) either \( obj_1, obj_2 \leq cn_2, obj_1, obj_2 \leq cn \), or (wlog) \( obj_1 \leq cn_2 \) and \( obj_2 \leq cn \). In either case, since \( OID(cn_1) = OID(cn_2) \) by the definition of \( \equiv_1 \), the result follows.

**Task-Object Existence:** If \( tsk \leq cn_2 cn \) either \( tsk \leq cn_2 \) or \( tsk \leq cn \). In the former case, if \( tsk = Tsk(o, l, s) \) then, since \( cn_1 \equiv_1 cn_2 \) and so \( cn_2 \) is WF2, we find \( obj \leq cn_2 \) with \( OID(obj) = \{o\} \). Otherwise, since \( cn_1 cn \equiv_1 cn_2 cn \) we find \( o(a, u, q_{in}, q_{out}) \leq cn_1 cn \) and hence by definition of \( \equiv_1 \), \( o(a, u, q_{in}, q_{out}) \leq cn_2 cn \) as well.

**Object-Node Existence:** If \( o(a, u, q_{in}, q_{out}) \leq cn_2 cn \), then either the container is in \( cn_2 \), which is WF2 and thus has a node \( u \), or it is in \( cn \), which means node \( u \) is in \( cn_1 \), which has the same network as \( cn_2 \).

**Buffer Cleanliness:** If \( obj \leq cn_2 cn \) either \( obj \leq cn_2 \) or \( obj \leq cn \). In the former case we are done since \( cn_2 \) is WF2. In the latter case, we get \( obj \leq cn_1 cn \) and \( cn_1 cn \) is WF2, which is sufficient.
Local Routing Consistency: This is immediate since $cn$ contains only object and task containers.

Call Uniqueness: If $call \not\leq cn_2$ and $call' \not\leq cn$, and there is a future clash, there must be a clash between $call'$ and some message in $cn_1$. But this is ruled out since $cn_1 cn$ is WF2.

Future Uniqueness: Assume $cn_2$ assigns $v_1$ to $f$ and $cn$ assigns $v_2$ to $f$ and $v_1 \neq v_2$. Then, since $cn_1 \equiv_2 cn_2$, $cn_1$ assigns $v_1$ to $f$ as well, violating WF2.

Single Writer: Assume $tsk_1, tsk_2 \not\leq cn_2 cn$, with associated future identifiers $f_1$ and $f_2$. Clearly, $f_1 \neq f_2$, and both are unassigned, or else WF2 would be violated for $cn_2$ or $cn_1 cn$. Assume $call \not\leq cn_2 cn$ with $f$; then $f$ is distinct from $f_1$ and $f_2$, or there would have been a clash in $cn_2$ or $cn_1 cn$, ruled out by WF2.

External OID: $cn_2$ is WF2, so $ext \notin OID(cn_2)$. Also, $cn_1 cn$ is WF2, so $ext \notin OID(cn)$. Hence $ext \notin OID(cn_2 cn)$. Also, if $t$ is a routing table in $cn_2 cn$ it is a routing table in $cn_2$ and $cn_2$ is WF2. Then $ext \notin dom(t)$, as required.

Future Liveness: Assume $f$ is active for $o$ in $cn_2 cn$; then $f$ is active for $o$ in $cn_1 cn$, and thus on the notification path of $f$ there. Hence, it is also on the notification path of $o$ in $cn_2 cn$.

We next need to show that $cn_1 cn \equiv_1 cn_2 cn$. The WF2 property is immediate. Suppose $A_1(cn_1 cn) \leadsto cn'_1$ and $A_1(cn_2 cn) \leadsto cn'_2$. It suffices to prove $cn'_1 R cn'_2$. We check the requirements:

- $\text{graph}(cn'_1) = \text{graph}(cn_1 cn) = \text{graph}(cn_1) = \text{graph}(cn_2 cn) = \text{graph}(cn'_2)$.
- $t(cn'_1) = t(cn_1 cn) = t(cn_1) \cup t(cn) = t(cn_2) \cup t(cn) = t(cn_2 cn) = t(cn'_2)$.
- $o(cn'_1) \supseteq o(cn_1 cn) = o(cn_1) \cup o(cn) \equiv_1 o(cn_2) \cup o(cn) = o(cn_1 cn) \equiv_1 o(cn'_2)$.
- $m(cn'_1) = m(cn_1 cn) = m(cn_1) \cup m(cn) = m(cn_2) \cup m(cn) = m(cn_2 cn) = m(cn'_2)$.

**Proposition 9.10.** $\equiv_1$ is a type 2 witness relation.

**Proof.** Reduction closure and its converse follows from Lemma 9.8, and context closure and its converse follows from Lemma 9.9. Hence, it suffices to show barb preservation in both directions. Suppose $cn_1 \equiv_1 cn_2$ and $cn_1 \downarrow obs$. Then, there is a call message $msg$ with destination $ext$ at the head of some self-loop queue in $cn_1$. After running Algorithm 1 on $cn_2$, we will have external link messages, with $msg$ in some link queue. Using MSG-DELAY-1 and MSG-DELAY-3, $msg$ can then be brought to the head of some self-loop queue. Hence, $cn_2 \downarrow obs$. The proof of converse barb preservation is symmetric.

**Proposition 9.12.** Algorithm 2 terminates.

**Proof.** Routing is stable after each run of Algorithm 1, and none of the rules applied in the outermost loop in the first outermost loop affect routing.
Thus, one of msg-send or msg-delay-2 will be enabled whenever the output outqueue is nonempty, causing output queue size to decrease by one. By Buffer Cleanliness, one of call-rcv-2 or fut-rcv will be applicable if the object in-queue is nonempty, decreasing in-queue size by one. Thus, when the inner while loop is reached, each nonempty inqueue has decreased in size by one, and each out-queue may have increased in size by one if the in-queue head position contains a delayed message.

Sending future messages may cause out-queues to increase in size. Each application of fut-send causes a forwarding list to decrease in length by one. Thus, termination of the inner while loop is clear. We need to argue that the outer while loop also terminates.

We first show that, eventually, no forwarding list is incremented. Only two rules can cause forwarding lists to increase in size, namely call-send-2 and call-rcv-2. Of these, call-send-2 is never used in either Algorithm 1 or Algorithm 2. Each application of call-rcv-2 consumes one call message, and none of the rules cause new call messages to be created. Thus, eventually, call-rcv-2 is never applied, and from that point onward forwarding lists are either emptied completely by the inner loop, or they remain untouched, since their corresponding future is undefined. Futures can become instantiated by fut-rcv, but again, this can only happen a bounded number of times. Moreover, the only rule causing futures to be created is msg-send, so the supply of futures to consider is fixed. Consequently, eventually, each future either remains uninstantiated forever, or else the corresponding forwarding list is empty. From that point onward, no fut-send is enabled, and the innermost loop terminates trivially in all future iterations. In this situation, since msg-send, call-rcv-2, and fut-rcv all consume messages from a bounded resource (the set of messages in transit), if the outermost loop fails to terminate the only option is that, from some point onwards, only msg-delay-2 is applied. From this point onward, since routing is stable, all messages will eventually be delivered.

Termination of the final loop is trivial. Observe that Algorithm 2 does not rely on routing to move the object towards u. For the algorithm, it is sufficient to establish that some good direction exists, and this is clearly the case as the network is stable and connected.

**Proposition 9.14.** Suppose cn is WF2. If $A_2(cn) \rightarrow cn'$, then the following holds:

1. $cn'$ is reachable from $cn$.
2. $cn'$ is in normal form.
3. graph($cn$) = graph($cn'$).
4. $t_2(cn) = t(cn')$.
5. $o_2(cn) = o(cn')$.
6. $m_2(cn) = m(cn')$. 

50
Case 5: again contradicts the assumption of having only external messages.

Proof. Property 1 is immediate. Property 3 follows from property 2 of Proposition 9.5.

For property 4, observe first that the function \( t_2 \) is invariant under transitions used in Algorithm 2. On termination of Algorithm 2, only external messages are in transit, and since no rule cause a task to be modified, property 4 follows.

For property 5, let \( o(a_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq o(cn') \). We show that it is then the case that \( o(a_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq o_2(cn) \). From the definition of Algorithm 2, \( q_{in,2} = q_{out,2} = \varepsilon \). Also, \( u_2 = u \). We know that there is an object container \( o(a, a', u'', q_{in}, q_{out}) \preceq cn \), since there is a one-to-one correspondence between object containers in pre- and poststate for each transition used in Algorithm 2. We also know that \( a'(x) = a_2(x) \) for all \( x \). Suppose finally that an object container \( obj = o(a_1, a_1, u_1, q_{in,1}, q_{out,1}) \) exists in \( cn \) with \( a_1(f) = (v, \eta) \). Let \( o(a_1, a_1', u_1', q_{in,1}', q_{out,1}') \) be the derivative of \( obj \) in \( cn' \). Then \( \pi_1(a_1'(f)) = v \) as well, by Proposition 7.7. We know by Future Uniqueness that \( a_2(f) = (v', \overline{\eta}) \) implies \( v' = v \). It remains to show that \( \pi_2(a_2(f)) \neq \perp \). Assume not. We have that \( a_2 \) is on the notification path of \( f \) in \( n \) steps for some nonnegative integer \( n \). We proceed by induction on the notification path relation:

Case 1: \( n = 0 \), and \( \pi_1(a_2(f)) = v \neq \perp \). This yields a contradiction.

Case 2: \( n = 1 \), and there is future message \( future(a_2, f, v') \preceq cn \). This yields a contradiction, since the only queued messages in \( cn' \) are external.

Case 3: \( n = 1 \), and there is a task \( t(o_2, l_2, s_2) \preceq cn' \) with \( l_2(ret) = f \). Then, \( \pi_1(a_1'(f)) = \perp \) by well-formedness, which is a contradiction.

Case 4: \( n = 2 \), and there is a call message \( call(o_2, o''_2, f, m, \overline{\eta}) \preceq cn' \). This again contradicts the assumption of having only external messages.

Case 5: \( n = 4 \), and there is a call message \( call(o', o_2, f, m, \overline{\eta}) \preceq cn' \) such that \( o' \in OID(cn') \). This again contradicts the assumption of having only external messages.

Case 6: \( n = n' + 2 \) and there is an object container \( o(a, a, u, q_{in}, q_{out}) \preceq cn' \) such that \( o_2 \in \pi_2(a(f)) \), and \( o \) is on the notification path from \( f \) in \( n' \) steps. Then, if we have \( \pi_1(a(f)) = v \neq \perp \), \( cn' \) is not in normal form, a contradiction; alternatively, we conclude by the IH.

Case 7: \( n = 2n' + n'' \) and there is an object container \( o(a''_2, a, u, q_{in}, q_{out}) \preceq cn' \) such that \( a \) is on the notification path of \( f' \) by \( a''_2 \) in \( n'' \) steps, \( f' \) is assigned to \( f \), and there is an object container \( o(a', a', u', q_{in}, q_{out}) \preceq cn' \) such that \( a' \) is on the notification path of \( f \) in \( n'' \) steps. By the IH, we have \( \pi_2(a'(f)) = v \neq \perp \) and thus \( \pi_2(a(f)) = v \). Since \( f \) will have been forwarded down to \( o_2, v \) will have been forwarded down the same chain.

We can thus conclude that \( o(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq o_2(cn) \). Conversely, assume that \( o(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq o_2(cn) \). Object \( o_2 \) has exactly one derivative in \( cn' \), by well-formedness. That object has empty queues, the same NID as in \( cn \), preserves assignments to variables, and has \( \pi_1(a_2(f)) \) assigned to a non-\( \perp \) value if and only if some object in \( cn' \) has so, by the above argument.
For 9.14.6, the property holds as it does so already for Algorithm 1.

We finally need to prove property 2. Property 9.13.1 is trivial, as each run of Algorithm 2 ends with a run of Algorithm 1, and Algorithm 1 ensures that $cn'$ has stable routing. Property 9.13.2 holds since Algorithm 1 ensures almost empty link queues, and since on termination, Algorithm 2 ensures empty object queues. For 9.13.3, if $obj = o(a, a', u', \varepsilon, \varepsilon)$ satisfies the properties defining $q_2$ above then, referring to those conditions, $u' = u'' = u$, $a' = a$, $q_{in} = \varepsilon = q_{out}$, and $obj \preceq cn'$ as needed to be shown. For 9.13.4, the result follows since only external messages are in transit in $cn'$.

\begin{lemma}
\label{lem:reduce-closed}
\end{lemma}

\begin{proof}
Suppose $cn_1 \equiv_2 cn_2$, where $cn_1$ and $cn_2$ are WF2. Assume $cn_1 \rightarrow cn'_1$; we need to find $cn'_2$ such that $cn_2 \rightarrow^* cn'_2$ and $cn'_1 \equiv_2 cn'_2$. We proceed by case analysis on the transition $cn_1 \rightarrow cn'_1$, eliding uses of CTXT-1.

For the cases T-SEND, T-RCV, MSG-SEND, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, MSG-DELAY-2, MSG-DELAY-3, CALL-RCV-2, FUT-SEND, FUT-RCV, OBJ-REG, OBJ-SEND, and OBJ-RCV, we take $cn'_2 = cn_2$, since then, the normal form is unaffected, i.e., $cn_1 \equiv_2 cn'_1$, by Proposition 9.14.

The remaining cases include the rules for sequential control, CALL-SEND-2, RET-2, GET-2, and NEW-2. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition of the form

$cn \circ (o(a, a, u, q_{in}, q_{out}) \ t(o, l, x = \varepsilon; s) \rightarrow cn \circ (o(a, a[v/x], u, q_{in}, q_{out}) \ t(o, l, s)$

where $[v]_I(a, l) = v$ and $x \in \text{dom}(a)$. Consider $cn''_2$ such that $A_2(cn_2) \rightarrow cn''_2$. By Proposition 9.14, there is a task $t(o, l, x = \varepsilon; s)$ and an object $o(a', u', \varepsilon, \varepsilon)$ in $cn''_2$. Hence, it is possible to perform a transition

$cn' \circ (o(a', u', \varepsilon, \varepsilon) \ t(o, l, x = \varepsilon; s) \rightarrow cn' \circ (o(a', [v/x], u', \varepsilon, \varepsilon) \ t(o, l, s)$

and we have

$cn \circ (o(a, a[v/x], u, q_{in}, q_{out}) \ t(o, l, s) \equiv_2 cn' \circ (o(a, a'[v/x], u', \varepsilon, \varepsilon) \ t(o, l, s)$

as needed, setting $cn'_2$ to the right-hand side.

\begin{proof}

call-send-2: Consider a transition of the form

$cn \circ (o(a, a, u, q_{in}, q_{out}) \ t(o, l, x = \varepsilon_1 m(\tau_2); s) \rightarrow cn \circ (o(fw(\eta, o', \text{init}(f, a)), u, q_{in}, \text{enq}(msg, q_{out})) t(o, l[f/x], s)$

where $\eta = [\tau_2]_{(a,j)}, f = \text{nwe}_f(a), msg = \text{call}(o', a, f, m, \eta)$, and $o' = [\varepsilon_1]_{(a,l)}$. Consider $cn''_2$ such that $A_2(cn_2) \rightarrow cn''_2$. By Proposition 9.14, there is a task $t(o, l, x = \varepsilon_1 m(\tau_2); s)$ and an object $o(a', u', \varepsilon, \varepsilon)$ in $cn''_2$ with $[\tau_2]_{(a,l)} = \eta$ and $[\varepsilon_1]_{(a,l)} = o'$. Hence, it is possible to perform a transition

$cn' \circ (o(a', u', \varepsilon, \varepsilon) \ t(o, l, x = \varepsilon_1 m(\tau_2); s) \rightarrow cn' \circ (o(fw(\eta, o', \text{init}(f, a')), u', \varepsilon, \text{enq}(msg, \varepsilon)) t(o, l[f/x], s)$

52
and we have
\[
\text{cn} \ o(a, fu(\pi, o', \text{init}(f, a)), u, q_{\text{in}}, \text{enq}(\text{msg}, q_{\text{out}})) \ t(o, l[0|x], s) \\
\equiv_2 \ \text{cn'} \ o(a, fu(\pi, o', \text{init}(f, a')), u', \varepsilon, \text{enq}(\text{msg}, \varepsilon)) \ t(o, l[0|x], s)
\]
as needed, setting \( cn_2' \) to the right-hand side.

**ret-2:** Consider a transition of the form
\[
\text{cn} \ o(a, a, u, q_{\text{in}}, q_{\text{out}}) \ t(o, l, \text{return} \ e; s) \rightarrow \text{cn} \ o(a, a[u/f], u, q_{\text{in}}, q_{\text{out}})
\]
where \( v = [e]_{(a,l)} \) and \( f = l(\text{ret}) \). Consider \( cn_2'' \) such that \( A_2(cn_2) \leadsto cn_2'' \). By Proposition 9.14, there is a task \( t(o, l, \text{return} \ e; s) \) and an object \( o(a, a', u', \varepsilon, \varepsilon) \) in \( cn_2'' \) where \( [e]_{(a',l)} = v \). Hence, it is possible to perform a transition
\[
\text{cn'} \ o(a, a', u', \varepsilon, \varepsilon) \ t(o, l, \text{return} \ e; s) \rightarrow \text{cn'} \ o(a, a'[v/f], u', \varepsilon, \varepsilon)
\]
and we have
\[
\text{cn} \ o(a, a[v/f], u, q_{\text{in}}, q_{\text{out}}) \equiv_2 \text{cn'} \ o(a, a'[v/f], u', \varepsilon, \varepsilon)
\]
as needed, setting \( cn_2' \) to the right-hand side.

**get-2:** Consider a transition of the form
\[
\text{cn} \ o(a, a, u, q_{\text{in}}, q_{\text{out}}) \ t(o, l, x = e.\text{get}; s) \rightarrow \text{cn} \ o(a, a, u, q_{\text{in}}, q_{\text{out}}) \ t(o, l[v/x], s)
\]
where \( [e]_{(a,l)} = f \) and \( \pi_1(a(f)) = v \). Consider \( cn_2'' \) such that \( A_2(cn_2) \leadsto cn_2'' \). By Proposition 9.14, there is a task \( t(o, l, x = e.\text{get}; s) \) and an object \( o(a, a', u', \varepsilon, \varepsilon) \) in \( cn_2'' \) where \( [e]_{(a',l)} = f \) and \( \pi_1(a'(f)) = v \). Hence, it is possible to perform a transition
\[
\text{cn'} \ o(a, a', u', \varepsilon, \varepsilon) \ t(o, l, x = e.\text{get}; s) \rightarrow \text{cn'} \ o(a, a', u', \varepsilon, \varepsilon) \ t(o, l[v/x], s)
\]
and we have
\[
\text{cn} \ o(a, a, u, q_{\text{in}}, q_{\text{out}}) \ t(o, l[v/x], s) \equiv_2 \text{cn'} \ o(a, a', u', \varepsilon, \varepsilon) \ t(o, l[v/x], s)
\]
as needed, setting \( cn_2' \) to the right-hand side.

**new-2:** Consider a transition of the form
\[
\text{cn} \ o(a, a, u, q_{\text{in}}, q_{\text{out}}) \ t(o, l, x = \text{new} \ C(\pi); s) \\
\rightarrow \ \text{cn} \ o(a, a', u, q_{\text{in}}, q_{\text{out}}) \ t(o, l[0'/x], s) \ o(a', \alpha', u, \varepsilon, \varepsilon)
\]
where \( \alpha' = \text{new}(u), \pi = [\pi]_{(a,l)}, \alpha' = fu(\pi, \alpha', a) \), and \( \alpha'' = \text{init}(\pi, \text{init}(C, \pi, \alpha')) \). Consider \( cn_2'' \) such that \( A_2(cn_2) \leadsto cn_2'' \). By Proposition 9.14, there is a task \( t(o, l, x = \text{new} \ C(\pi); s) \) and an object \( o(a, a_1, u', \varepsilon, \varepsilon) \) in \( cn_2'' \) where \( [e]_{(a_1,l)} = \pi \) and \( \pi_1(a_1(f)) = v \). Hence, it is possible to perform a transition
\[
\text{cn'} \ o(a, a_1, u', \varepsilon, \varepsilon) \ t(o, l, x = \text{new} \ C(\pi); s) \\
\rightarrow \ \text{cn'} \ o(a, a_1', u', \varepsilon, \varepsilon) \ t(o, l[0'/x], s) \ o(a', \alpha', u', \varepsilon, \varepsilon)
\]
and we have
\[ \text{cn} \circ (o, a', u, q_{in}, q_{out}) \circ t(o, l[\ell/x], s) \circ (o', a'', u, \varepsilon, \varepsilon) \equiv_2 \text{cn}' \circ (o, a', u', \varepsilon, \varepsilon) \circ t(o, l[\ell'/x], s) \circ (o', a'', u', \varepsilon, \varepsilon) \]
as needed, setting \( \text{cn}'_2 \) to the right-hand side. \( \square \)

**Lemma 9.19.** \( \equiv_2 \) is context closed.

**Proof.** The proof follows the proof of Lemma 9.9. Assume \( \text{cn}_1 \equiv_2 \text{cn}_2 \) and \( \text{cn}_1 \text{cn} \) is WF2. We first show that \( \text{cn}_2 \text{cn} \) is WF2 as well.

**OID Uniqueness:** If \( \text{obj}_1, \text{obj}_2 \preceq \text{cn}_2 \text{cn} \) either \( \text{obj}_1, \text{obj}_2 \preceq \text{cn}_2 \), \( \text{obj}_1, \text{obj}_2 \preceq \text{cn}_2 \), or (wlog) \( \text{obj}_1 \preceq \text{cn}_2 \) and \( \text{obj}_2 \preceq \text{cn}_2 \). In either case, since \( \text{OID}(\text{cn}_1) = \text{OID}(\text{cn}_2) \) by the definition of \( \equiv_2 \), the result follows.\[ \]

**Task-Object Existence:** If \( \text{tsk} \preceq \text{cn}_2 \text{cn} \) either \( \text{tsk} \preceq \text{cn}_2 \) or \( \text{tsk} \preceq \text{cn}_2 \). In the former case, if \( \text{tsk} \equiv t(o, l, s) \) then, since \( \text{cn}_1 \equiv_2 \text{cn}_2 \) and so \( \text{cn}_2 \) is WF2, we find \( \text{obj} \preceq \text{cn}_2 \) with OID \( a \). Otherwise, since \( \text{cn}_1 \text{cn} \equiv_2 \text{cn}_2 \) we find \( o(a, a, u, q_{in}, q_{out}) \leq \text{cn}_1 \text{cn} \) and hence by definition of \( \equiv_2 \), \( o(o', a', u', q_{in}', q_{out}') \leq \text{cn}_2 \text{cn} \) as well.

**Object-Node Existence:** If \( o(o, a, u, q_{in}, q_{out}) \leq \text{cn}_2 \text{cn} \), then either the container is in \( \text{cn}_2 \), which is WF2 and thus has a node \( u \), or it is in \( \text{cn} \), which means node \( u \) is in \( \text{cn}_1 \), which has the same network as \( \text{cn}_2 \).

**Buffer Cleanliness:** If \( \text{obj} \preceq \text{cn}_2 \text{cn} \) either \( \text{obj} \preceq \text{cn}_2 \) or \( \text{obj} \preceq \text{cn}_2 \). In the former case we are done since \( \text{cn}_2 \) is WF2. In the latter case, we get \( \text{obj} \preceq \text{cn}_1 \text{cn} \) and \( \text{cn}_1 \text{cn} \) is WF2, which is sufficient.

**Local Routing Consistency:** This is immediate since \( \text{cn} \) contains only object and task containers.

**Call Uniqueness:** If \( \text{call} \preceq \text{cn}_2 \) and \( \text{call}' \preceq \text{cn}_2 \), and there is a future clash, there must be a clash between \( \text{call}' \) and some message in \( \text{cn}_1 \) or a task in \( \text{cn}_1 \). But this is ruled out since \( \text{cn}_1 \text{cn} \) is WF2.

**Future Uniqueness:** Assume \( \text{cn}_2 \) assigns \( v_1 \) to \( f \) and \( \text{cn} \) assigns \( v_2 \) to \( f \) and \( v_1 \neq v_2 \). Then, since \( \text{cn}_1 \equiv_2 \text{cn}_2 \), \( \text{cn}_2 \) assigns \( v_1 \) to \( f \) as well, violating WF2.

**Single Writer:** Assume \( \text{tsk}_1, \text{tsk}_2 \preceq \text{cn}_2 \text{cn} \), with associated future identifiers \( f_1 \) and \( f_2 \). Clearly, \( f_1 \neq f_2 \), and both are unassigned, or else WF2 would be violated for \( \text{cn}_2 \) or \( \text{cn}_1 \text{cn} \). Assume \( \text{call} \preceq \text{cn}_2 \text{cn} \) with \( f \); then \( f \) is distinct from \( f_1 \) and \( f_2 \), or there would have been a clash in \( \text{cn}_2 \) or \( \text{cn}_1 \text{cn} \), ruled out by WF2.

**External OID:** If \( \text{ext} \in \text{OID}(\text{cn}_2 \text{cn}) \), then \( \text{ext} \in \text{OID}(\text{cn}) \), which is ruled out by WF2; the second requirement is immediate since \( \text{cn}_2 \text{cn} \) is WF2 and \( \text{cn} \) contains no nodes.

**Future Liveness:** Assume \( f \) is active for \( o \) in \( \text{cn}_2 \text{cn} \); then \( f \) is active for \( o \) in \( \text{cn}_1 \text{cn} \), and thus on the notification path of \( f \) there. Hence, it is also on the notification path of \( o \) in \( \text{cn}_2 \text{cn} \).

We next need to show that \( \text{cn}_1 \text{cn} \equiv_2 \text{cn}_2 \text{cn} \). The WF2 property is immediate. Suppose \( A_2 (\text{cn}_1 \text{cn}) \rightsquigarrow \text{cn}_1' \text{cn} \) and \( A_2 (\text{cn}_2 \text{cn}) \rightsquigarrow \text{cn}_2' \). It suffices to prove \( \text{cn}_1' \text{cn} \equiv_2 \text{cn}_2' \text{cn} \). We check the requirements:
\[
\text{graph}(cn_1') = \text{graph}(cn_1) = \text{graph}(cn_1) = \text{graph}(cn_2) = \text{graph}(cn_2) = \text{graph}(cn_2').
\]

Since \(t_2(cn_1) = t_2(cn_2)\), we have \(t_2(cn_1) = t_2(cn_2)\). Hence, \(t(cn_1') = t_2(cn_1) = t_2(cn_2) = t(cn_2')\).

Since \(o_2(cn_1) = o_2(cn_2)\), we have \(o_2(cn_1) = o_2(cn_2)\). Hence, \(o(cn_1') = o_2(cn_1) = o_2(cn_2) = o(cn_2')\).

\[
m(cn_1') = m_2(cn_1) = m_2(cn_1) = m_2(cn_2) = m_2(cn_2) = m(cn_2').
\]

\[\square\]

**Appendix 4: Proofs for Section 10**

**Proposition 10.3.** If \(\text{bind } \tau.cn\) is a WF1 configuration in standard form, then \(\text{net}(cn)\) is WF2.

**Proof.** We consider the WF2 conditions in turn. OID Uniqueness and Task-Object Existence follows from the respective WF1 conditions and from how the name representation map is defined. Object-Node Existence holds since all objects are placed on the node \(n_0\), which exists by the definition of \(cn_{\text{graph}}\). Buffer Cleanliness holds since all object containers in \(\text{net}(cn)\) have empty queues. Local Routing Consistency follows from how routing tables in \(cn_{\text{graph}}\) are defined. Call Uniqueness follows from the corresponding WF1 condition and from how the name representation map is defined. Future Uniqueness follows the definition of \(\text{oenvmap}\), how object environments are represented, and from Proposition 10.1. Single Writer follows the corresponding WF1 condition and from how the name mapping representation map is defined. External OID follows from the corresponding WF1 condition and from how routing tables are defined.

Consider finally Future Liveness. Assume \(o(a, a, u, q_{in}, q_{out}) \preceq \text{net}(cn)\). Note that by the way we define future mappings in object environments, we have that, whenever \(f\) is active for \(o\) in \(\text{net}(cn)\), \(o(f) \perp\) holds. Suppose therefore that \(o(f) \perp\). Then, there must be some future container for the FID corresponding to \(f\) in \(\text{bind } \tau.cn\) with lifted value \(v\). If \(v_\perp \neq \perp\), then \(o(f) = (v, \varepsilon)\) for some value \(v\), which means that \(o\) is trivially on the notification path of \(f\) in \(\text{net}(cn)\). If \(v_\perp = \perp\), there is, by Future Existence, some task or some call message associated with \(f\) for some object \(o'\) in \(\text{net}(cn)\), and, additionally, \(o'\) will have \(o\) in its forwarding list for \(f\); hence \(o\) is on the notification path of \(f\) in this case also.

Suppose finally that \(\text{net}(cn)\) assigns \(f\) to \(f'\), and \(o\) is on the notification path of \(f'\). There must then be a corresponding assignment in \(\text{bind } \tau.cn\), which by Future Existence yields that there is future container for the future corresponding to \(f\) with lifted value \(v\). If \(v_\perp \neq \perp\), then \(o(f) = (v, \varepsilon)\) for some value \(v\), which means that \(o\) is trivially on the notification path of \(f\). If \(v_\perp = \perp\), there is again, by Future Existence, some task or some call message associated with \(f\) for some object \(o'\) in \(\text{net}(cn)\), and, additionally, \(o'\) will have \(o\) in its forwarding list for \(f\); hence \(o\) is on the notification path of \(f\) in this case also.

\[\square\]
Proposition 10.8. Let \( cn_1 \) and \( cn_2 \) be WF2 configurations such that \( cn_1 \succeq cn_2 \). Then, if \( o \) is on the notification path of \( f \) by \( o_2 \) in \( n \) steps in \( cn_2 \), \( o \) is on the notification path of \( f \) by some \( o_1 \) in at most \( n \) steps in \( cn_1 \), such that the notification path from \( o \) to \( o_2 \) in \( cn_2 \) traverses \( o_1 \).

Proof. Suppose \( o(a, o_1, u, q_{in}, q_{out}) \preceq cn_1 \) and \( o(a, o_2, u, q_{in}, q_{out}) \preceq cn_2 \). The proof is by induction on the notification path relation.

Case 1: \( n = 0, o_2 = o \), and \( \pi_1(a_2(f)) = v \). Then, we have \( \pi_1(a_1(f)) = v \), and we set \( n = 0 \) and \( o_1 = o \).

Case 2: \( n = 1, o_2 = o \), and there is a message \( \text{future}(o, f, v) \preceq cn_2 \). Since messages are unaffected by extension, the same message is in \( cn_1 \), and we set \( n = 1 \) and \( o_1 = o \).

Case 3: \( n = 1, o_2 = o \), and there is a task \( t(o, l, s) \preceq cn_2 \), such that \( t(\text{ret}) = f \). Since task containers are unaffected by extension, the same container is in \( cn_1 \), and we set \( n = 1 \) and \( o_1 = o \).

Case 4: \( n = 2, o_2 = o \), and there is a call message \( \text{call}(o, o', f, m, \pi) \preceq cn_2 \).

Then, the same message is in \( cn_1 \), and we set \( n = 2 \) and \( o_1 = o \).

Case 5: \( n = 4 \) and there is a message \( \text{call}(o_2, o, f, m, \pi) \preceq cn_2 \). Then, the same message is in \( cn_1 \), and we set \( n = 4 \) and \( o_1 = o_2 \).

Case 6: \( n = n_2' + 2, \) and there is an object \( o(a', o_2', u', q_{in}', q_{out}') \preceq cn_2 \) such that \( o \in \pi_2(a_2'(f)) \), and \( o' \) is on the notification path of \( f \) by \( o_2' \) in \( n_2' \) steps in \( cn_2 \). As induction hypothesis, we have that \( o' \) is on the notification path of \( f \) by some \( o_1' \) in some number \( n_1' \) steps (\( n_1' \leq n_2' \)) in \( cn_1 \). Consider the object \( o(a', o_1', u', q_{in}', q_{out}') \preceq cn_1 \); by the definition of extended object container, we have that \( o \) is forwarding resolved for \( f \) in \( cn_1 \) at \( o' \). This means that either \( o \in \pi_2(a_1'(f)) \) or \( \pi_1(a_1(f)) = v \), where \( cn_1 \) assigns \( v \) to \( f \). In the first case, we set \( n = n_1' + 2 \) and \( o_1 = o_1' \). In the second case, we set \( n = 0 \) and \( o_1 = o \).

Case 7: \( n = 2n_2' + n_2'' \), and \( o \) is on the notification path of \( f' \) by \( o_2' \) in \( n_2' \) steps in \( cn_2 \), \( cn_2 \) assigns \( f \) to \( f' \), and \( o_2' \) is on the notification path of \( f \) by \( o_2 \) in \( n_2'' \) steps in \( cn_2 \). Since no configuration future assignments are changed by extension, we have that \( cn_1 \) assigns \( f \) to \( f' \). As induction hypotheses, we have first that \( o \) is on the notification path of \( f' \) by some \( o_1' \) in some \( n_1' \) steps (\( n_1' \leq n_2'' \)) in \( cn_1 \), such that the notification path from \( o \) to \( o_1' \) in \( cn_2 \) traverses \( o_2' \). Second, we have that \( o_2' \) is on the notification path of \( f \) by some \( o_1'' \) in some \( n_1'' \) steps (\( n_1'' \leq n_2'' \)) in \( cn_1 \). Since the path from \( o \) to \( o_2' \) in \( cn_2 \) traverses \( o_1' \), and because of forward resolution in \( cn_1 \), \( o_1' \) is on the notification path of \( f \) by \( o_1'' \) in at most \( n = 2n_2' + n_2'' \) steps, setting \( o_1 = o_1'' \).

\( \Box \)

Proposition 10.9. If \( cn_1 \) and \( cn_2 \) are WF2 configurations such that \( cn_1 \succeq cn_2 \) and there are \( cn_1' \) and \( cn_2' \) such that \( A_2(cn_1) \succeq cn_1' \), and \( A_2(cn_2) \succeq cn_2' \), then \( cn_1' \succeq cn_2' \).

Proof. Let \( \text{obj} = o(a, a_1', u, \varepsilon, \varepsilon) \) be an object container in \( cn_1' \) and let \( \text{obj}' = o(a, a_2', u, \varepsilon, \varepsilon) \) be the corresponding container in \( cn_2' \). There must then be a container \( o(a, a_1, u', q_{in}, q_{out}) \preceq cn_1 \) and a container \( o(a, a_2, u', q_{in}, q_{out}) \preceq cn_2 \).

Case 1: We have \( a_2(x) = a_1(x) \) for all \( x \), and \( a_2(\text{self}) = a_1(\text{self}) \).

Since normalization does not affect object-environment variable-value mappings, we get that \( a_2'(x) = a_1'(x) \), for all \( x \), and \( a_2'(\text{self}) = a_1'(\text{self}) \).
Case 2: Suppose $\pi_1(o'_2(f)) = v$. Then, $cn_2$ must assign $v$ to $f$, and $o$ must be on the notification path of $f$ in $cn_2$. Consequently, by Proposition 10.8, $o$ is then on the notification path of $f$ in $cn_1$, while also assigning $v$ to $f$. Hence, it is the case that $\pi_1(o'_1(f)) = v$.

Case 3: Suppose $o'' \in \pi_2(o'_2(f))$. Then, $f$ is unresolved at $o$ in $cn'_2$, i.e. $\pi_1(o'_2(f)) = \bot$, and thus $f$ is unassigned in $cn'_2$. Consequently, $f$ is unassigned at $o$ in $cn_2$, and hence also in $cn_1$. We now have two cases. Assume first $o'' \in \pi_2(a_2(f))$; then, $o'' \in \pi_2(a_1(f))$ and consequently $o'' \in \pi_2(o'_1(f))$, as needed. Assume next there is some sequence of futures $f_1, f_2, \ldots, f_n$ such that $cn_2$ assigns $f_{i+1}$ to $f_i$ for $1 \leq i < n$, $cn_2$ assigns $f$ to $f_n$, and $o'' \in \pi_2(o_2(f_1))$. By well-formedness, $o$ is on the notification path of $f_1, f_2, \ldots, f_n$ in $cn_2$, and hence also in $cn_1$. By assumption, we have that $o''$ is forwarding resolved for $f_1$ in $cn_1$ at $o$, which means that either $o'' \in \pi_2(a_1(f_1))$ or that $o''$ is forwarding resolved for $f_2$ in $cn_1$ at $o$, etc. Hence, it will be the case that $o'' \in \pi_2(o'_1(f))$, as needed.

Case 4: Suppose $\pi_1(o'_1(f)) = v$. Then, $cn'_1$ assigns $v$ to $f$, and consequently, so does $cn'_2$, and hence, since normalization does not affect future assignments, so does $cn_1$ and $cn_2$. If $v \in PVal$, we can then conclude. Assume instead $v = f' \in FID$. Assume $\pi_1(a_1(f)) = f'$; this means that $o$ is on the notification path of $f'$ in $cn_1$ and thus in $cn'_1$. Otherwise, $o$ must be on the notification path of $f$ in $cn_1$ (in some number of steps greater than 0), which means that $o$ is also on the notification path of $f$ in $cn'_1$, as needed.

Case 5: Suppose $o'' \in \pi_2(o'_2(f))$. Then, $f$ is unresolved at $o$ in $cn'_1$, i.e. $\pi_1(o'_2(f)) = \bot$, and thus $f$ is unassigned in $cn'_1$. Consequently, $f$ is unassigned at $o$ in $cn_1$, and hence also in $cn_2$. Suppose $o'' \in \pi_2(a_2(f))$. Then either $o'' \in \pi_2(a_2(f))$ or both $o'' \in OID(cn_2)$ and $o$ is on the notification of $f$ in $cn_1$. Assume the former; we then have $o'' \in \pi_2(o'_2(f))$ as required. Assume the latter; then we have $o'' \in OID(cn'_2)$ and $o$ is still on the notification path of $f$ in $cn'_1$, as needed.

**Lemma 10.11.** $\equiv_2$ is reduction closed.

**Proof.** Suppose that $cn_1 \equiv_2 cn_2$ and that $A_2(cn_1) \rightarrow cn''_1 \geq cn''_2 \geq A_2(cn_2)$. Then $cn_1$ and $cn_2$ are WF2. We proceed by case analysis on the transition $cn_1 \rightarrow cn'_1$, eliding uses of CXTX-1. For the cases T-SEND, T-RCV, MSG-SEND, MSG-RCV, MSG-RUNE, MSG-DELAY-1, MSG-DELAY-2, MSG-DELAY-3, CALL-RCV-2, FUT-SEND, FUT-RCV, OBJ-REG, OBJ-SEND, and OBJ-RCV, the configuration $cn_1'$ is fundamentally unchanged, so we set $cn'_2 = cn_2$ and have $A_2(cn'_1) \rightarrow cn''_1 \geq cn''_2 \geq A_2(cn'_2)$.

The remaining cases include the rules for assignment and sequential control in Figure 8, CALL-SEND-2, RET-2, GET-2, and NEW-2. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition from $cn_1$ to $cn'_1$ of the form

\[ cn \circ (o, a, u, q_{in}, q_{out}) \mathbin{t} (o, l, x = e; s) \rightarrow cn \circ (o, a[v/x], u, q_{in}, q_{out}) \mathbin{t} (o, l, s) \]

where $x \in \text{dom}(a)$ and $v = [e]_{\text{dom}(a)}$. By the definition of $\geq$, there is a task container $t(o, l, x = e; s)$ and an object container $o(o, a', u', v, e)$, where $a'$
has the same variable-value mappings as \( o \), in \( cn'_{o} \). Hence, it is possible to perform a transition
\[
 cn' \left( o, o', u', e, \varepsilon \right) t \left( o, l, x = e; s \right) \rightarrow cn' \left( o, o'[v/x], u', e, \varepsilon \right) t \left( o, l, s \right)
\]
and we have that
\[
 cn \left( o, a[v/x], u, q_{in}, q_{out} \right) t \left( o, l, s \right) \cong cn' \left( o, o'[v/x], u', e, \varepsilon \right) t \left( o, l, s \right) ,
\]
setting \( cn'_{o} \) to the right-hand side. This is the case since the transition does not change future-value mappings in any object container.

**call-send-2**: Consider a transition of the form
\[
 cn \left( o, a_{1}, u_{1}, q_{in}, q_{out} \right) t \left( o, l, x = e_{1} ! m(\tau_{2}); s \right) \rightarrow cn \left( o, a'_{1}, u_{1}, q_{in}, \text{enq} (\text{call} \left( o', o, f, m, \tau \right), q_{out}) \right) t \left( o, l[f/x], s \right)
\]
where \( o' = [e_{1}]_{(a_{1}, t)} \), \( \tau = [\tau_{2}]_{(a_{1}, l)} \), \( f = \text{newf} (u_{1}) \), and \( a'_{1} = f u (\tau, o', \text{init} (f, a_{1})) \). By the definition of extension, we have that \( t \left( o, l, x = e_{1} ! m(\tau_{2}); s \right) \cong cn'_{o} \) and that there is some container \( o \left( o, a_{2}, u, \varepsilon, \varepsilon \right) \cong cn'_{o} \), with the same variable-value mappings as \( o \) in \( cn_{o} \). Hence, we can perform a transition
\[
 cn' \left( o, a_{2}, u, e, \varepsilon \right) t \left( o, l, x = e_{1} ! m(\tau_{2}); s \right) \rightarrow cn' \left( o, a'_{2}, u, \varepsilon, \text{enq} (\text{call} \left( o', o, f, m, \tau \right), \varepsilon) \right) t \left( o, l[f/x], s \right)
\]
where \( a'_{2} = f u (\tau, o', \text{init} (f, a_{2})) \). It now suffices to show that
\[
 cn \left( o, a'_{1}, u_{1}, q_{in}, \text{enq} (\text{call} \left( o', o, f, m, \tau \right), q_{out}) \right) t \left( o, l[f/x], s \right) \cong cn' \left( o, a'_{2}, u, \varepsilon, \text{enq} (\text{call} \left( o', o, f, m, \tau \right), \varepsilon) \right) t \left( o, l[f/x], s \right)
\]
referring to the left-hand side as \( cn'_{o} \) and the right-hand side as \( cn'_{o} \). Compared to the corresponding configurations before the transition, the only changes are that a task has progressed, a call message has been generated, and forwardings added to \( o' \) for the futures in \( \tau \). Since the configurations are WF2, \( o \) is on the notification path of the futures in \( \tau \). Hence, the changes effected by the transitions are the same for both configurations. Consequently, the equivalence holds.

**ret-2**: Consider a transition of the form
\[
 cn \left( o, a_{1}, u_{1}, q_{in}, q_{out} \right) t \left( o, l, \text{return} \ e; s \right) \rightarrow cn \left( o, a_{1}[v/f], u_{1}, q_{in}, q_{out} \right)
\]
where \( v = [e]_{(a_{2}, l)} \) and \( f = l(\text{ret}) \). By the definition of extension, we have that \( t \left( o, l, \text{return} \ e; s \right) \cong cn'_{o} \), and that there is some container \( o \left( o, a_{2}, u, \varepsilon, \varepsilon \right) \cong cn'_{o} \) such that \( v = [e]_{(a_{2}, l)} \). Hence, we can perform a transition
\[
 cn' \left( o, a_{2}, u, \varepsilon, \varepsilon \right) t \left( o, l, \text{return} \ e; s \right) \rightarrow cn' \left( o, a_{2}[v/f], u, \varepsilon, \varepsilon \right)
\]
and we wish to prove that
\[
 cn \left( o, a_{1}[v/f], u_{1}, q_{in}, q_{out} \right) \cong cn' \left( o, a_{2}[v/f], u, \varepsilon, \varepsilon \right)
\]
we can perform a transition

\[ \text{cn} \ o(o, a_1, u_1, q_{in}, q_{out}) \ t(o, l, x = \text{e.get}; \ s) \]

\[ \rightarrow \ \text{cn} \ o(o, a_1, u_1, q_{in}, q_{out}) \ t(o, l[v/x], s) \]

where \([v]_{(a_1, l)} = f\) and \(\pi_1(a_1(f)) = v\). By the definition of extension and well-formedness, we have that \(t(o, l, x = \text{e.get}; \ s) \preceq cn_2^n\) and that there is some \(o(a_2, u, \varepsilon, \varepsilon) \preceq cn_2^n\) such that \([v]_{(a_2, l)} = f\) and \(\pi(a_2(f)) = v\). Hence, we can perform a transition

\[ \text{cn}' \ o(o, a_2, u, \varepsilon, \varepsilon) \ t(o, l, x = \text{e.get}; \ s) \rightarrow \text{cn}' \ o(o, a_2, u, \varepsilon, \varepsilon) \ t(o, l[v/x], s) \]

and we have that

\[ \text{cn} \ o(o, a_1, u_1, q_{in}, q_{out}) \ t(o, l[v/x], s) \overset{2}{=} \text{cn}' \ o(o, a_2, u, \varepsilon, \varepsilon) \ t(o, l[v/x], s) \]

setting \(cn_2'^n\) to the right-hand side. In the same way as for \(\text{wfield-2}\), this is the case since the transition does not change the future-value mappings in any object container.

\textbf{new-2}: Consider a transition of the form

\[ \text{cn} \ o(o, a_1, u_1, q_{in}, q_{out}) \ t(o, l, x = \text{new} \ C(\overline{\tau}); \ s) \]

\[ \rightarrow \ \text{cn} \ o(o, a'_1, u_1, q_{in}, q_{out}) \ t(o, l[o'/x], s) \ o(o', a', u_1, \varepsilon, \varepsilon) \]

where we have \(o' = \text{new}(u_1)\), \(\overline{\tau} = [\tau]_{(a_1, l)}\), \(a'_1 = \text{fw}(\overline{\tau}, a', a_1)\), and \(a' = \text{init}(\overline{\tau}, \text{init}(C(\overline{\tau}); s)) \preceq cn_2^n\) and that there is some \(o(a_2, u, \varepsilon, \varepsilon) \preceq cn_2^n\) such that \([\overline{\tau}]_{(a_2, l)} = \overline{\tau}\). We can then perform a transition

\[ \text{cn}' \ o(o, a_2, u, \varepsilon, \varepsilon) \ t(o, l, x = \text{new} \ C(\overline{\tau}); s) \]

\[ \rightarrow \ \text{cn}' \ o(o, a'_2, u, \varepsilon, \varepsilon) \ t(o, l[o'/x], s) \ o(o', a', u, \varepsilon, \varepsilon) \]

where \(a'_2 = \text{fw}(\overline{\tau}, a', a_2)\). We wish to prove that

\[ \text{cn} \ o(o, a'_1, u_1, q_{in}, q_{out}) \ t(o, l[o'/x], s) \ o(o', a', u_1, \varepsilon, \varepsilon) \]

\[ \overset{2}{=} \text{cn}' \ o(o, a'_2, u, \varepsilon, \varepsilon) \ t(o, l[o'/x], s) \ o(o', a', u, \varepsilon, \varepsilon) \]

referring to the left-hand side as \(cn_1'\) and the right-hand side as \(cn_2'^n\). Note that the only added forwardings through the transition are from \(o\) to \(o'\), and that they will be for the same futures in both \(cn_1'\) and \(cn_2'^n\), by well-formedness. Hence, after running Algorithm 2, there will be an object container \(o(o', a', u, \varepsilon, \varepsilon)\) in both resulting configurations, where all futures that
are assigned in the original configurations are resolved. In addition both resulting configurations will contain the task \( t(o, l[p'/x], s) \). It remains to argue that the object container with OID \( o \) in the configuration resulting from \( cn'_1 \) extends the corresponding object container resulting from \( cn'_2 \). The only difference between the containers compared to their counterparts in \( cn''_1 \) and \( cn''_2 \) is the possible addition of \( o' \) in some forwarding lists; however, these additions will be the same for both containers, so extension still holds.

The same arguments are valid with minor changes for the case when \( cn''_2 \geq cn''_1 \).

\[ \text{Lemma 10.12.} \ \mathcal{C}_2 \text{ is context closed.} \]

**Proof.** Suppose \( cn_1 \mathcal{C}_2 cn_2 \), and let \( cn \) be a context configuration. We then have that \( cn_1 \) and \( cn_2 \) are WF2.

Assume \( cn_1 \geq cn_2 \) and that \( cn_1 \) is WF2. We first show that \( cn_2 \) is WF2. For OID Uniqueness, note that \( OID(cn_1) = OID(cn_2) \) by the definition of extension; hence, any OID clash violates the well-formedness of \( cn_1 \). For Task-Object Existence, it suffices to consider the case \( t(o, l, s, q_{in}, q_{out}) \leq cn \) and \( o(a, u, q_{in}, q_{out}) \leq cn_1 \); by the definition of \( \mathcal{C}_2 \), there is then also an object container with OID \( o \). Object-Node Existence follows from \( \text{graph}(cn_1) = \text{graph}(cn_2) \), which holds by the definition of \( \mathcal{C}_2 \). Buffer Cleanliness distributes over composition and thus holds for \( cn_1 \), \( cn_2 \), and \( cn_2 \) separately.

Local Routing Consistency only concerns nodes, which are unaffected by adding the context \( cn \). Call Uniqueness holds since messages in \( cn_2 \) are either from \( cn \), correspond to some message or task (which is a single writer) in \( cn_1 \); Single Writer holds for similar reasons. External OID follows again by noting that nodes are unchanged when applying \( cn \).

For Future Liveness, assume that \( o(a, u, q_{in}, q_{out}) \leq cn_2 \) and either \( f \) is active for \( o \) in \( cn_2 \) \( cn \), \( f \) \( \downarrow \), or \( cn_2 \) assigns \( f \) to \( f' \) and \( o \) is on the notification path of \( f' \). Assume without loss of generalization that \( o(a, u, q_{in}, q_{out}) \leq cn \), since the property already holds in \( cn_2 \) separately.

We consider the cases in turn. Suppose \( f \) is active for \( o \) in \( cn_2 \) \( cn \); then \( f \) is also active for \( o \) in \( cn_1 \) \( cn \), so \( o \) is on the notification path of \( f \) in \( cn_1 \) \( cn \). If the path in \( cn_1 \) \( cn \) from which \( o \) can receive \( f \) traverses some \( o' \in OID(cn_1) \), there must be an equivalent path in \( cn_2 \) \( cn \), since extension does not change forwardings to configuration-external OIDs. The case where the path does not go outside of \( cn \) is even simpler. The other cases hold for similar reasons.

The converse of this well-formedness argument, that \( cn_1 \) \( cn \) is WF2 whenever \( cn_2 \) \( cn \) is WF2, holds with only minor changes to the above reasoning.

Assume that \( \mathcal{A}_2(cn_1) \leadsto cn'_1 \geq cn'_2 \leadsto \mathcal{A}_2(cn_2) \). Let \( \mathcal{A}_2(cn_1) \leadsto cn''_1 \) and \( \mathcal{A}_2(cn_2 cn) \leadsto cn''_2 \). Well-formedness holds from the above reasoning. We wish to prove that \( cn''_1 \geq cn''_2 \). We consider the conditions in turn.

\[ \text{graph}(cn''_1) = \text{graph}(cn_1 cn) = \text{graph}(cn_1) = \text{graph}(cn_2) = \text{graph}(cn_2 cn) = \text{graph}(cn''_2). \]

\[ m(cn''_1) = m_2(cn_1 cn) = m_2(cn_1) = m_2(cn_2) = m_2(cn_2 cn) = m(cn''_2). \]
Since \( t_2(cn_1) = t_2(cn_2) \), we have \( t_2(cn_1) = t_2(cn_2) \). Hence, \( t(cn''_1) = t(cn''_2) \).

Consider \( \text{obj} = o(a, a_1, u, \varepsilon, \varepsilon) \preceq cn_1'' \) and \( \text{obj}' = o(a, a_2, u, \varepsilon, \varepsilon) \preceq cn_2'' \). We want to argue that \( \text{obj} \) extends \( \text{obj}' \). Suppose the containers are derived from an object container \( o(a, u, q_{in}, q_{out}) \preceq cn \).

Case 1: Since Algorithm 2 does not affect \text{self} or stored values, we have \( a_2(\text{self}) = a_1(\text{self}) \) and \( a_2(x) = a_1(x) \) for all \( x \).

Case 2: Assume \( \pi_1(a_2(f)) = v \). Then, if \( \pi_1(a(f)) = v \), we also have \( \pi_1(a_1(f)) = v \). Suppose \( \pi_1(a(f)) = \bot \); then, \( o \) is on the notification path in \( cn_2 \) and \( cn_2 \) assigns \( v \) to \( f \). If the notification path traverses \( o' \in \text{OID}(cn) \), a corresponding notification path will traverse \( o' \) in \( cn_1 \); since the forwardings and external future messages of \( cn_1 \) and \( cn_2 \) to external objects coincide after running Algorithm 2. Consequently, \( o \) is also on the notification path of \( f \) in \( cn_1 \); and we have \( \pi(a_1(f)) = v \) by well-formedness and the definition of Algorithm 2.

Case 3: Assume \( o'' \in \pi_2(a_2(f)) \). Then, \( f \) is unresolved in \( cn_1 \) and \( cn_2 \). We need to argue that \( o'' \) is forwarding resolved for \( o \) in \( cn''_1 \). Suppose that \( o'' \in \pi_2(a(f)) \); then clearly \( o'' \in \pi_2(a_1(f)) \). Suppose \( o'' \not\in \pi_2(a(f)) \); then there must some sequence of futures \( f_1, \ldots, f_n \), such that \( cn_1 \) and \( cn_2 \) assign \( f_i \) to \( f_{i+1} \) for \( 1 \leq i < n \), assign \( f \) to \( f_n \), and additionally \( o'' \in \pi_2(a(f_1)) \) and \( o \) is on the notification path of \( f_1 \). After running Algorithm 2, we will clearly have \( o'' \in \pi_2(a_1(f)) \).

Case 4: Assume \( \pi_1(a_1(f)) = v \). Then, either \( cn_1 \) or \( cn \) assigns \( v \) to \( f \), which implies that either \( cn_2 \) or \( cn \) assigns \( v \) to \( f \). Suppose \( v = f' \in \text{FID} \). If \( \pi_1(a(f)) = f' \), \( o \) is on the notification path of \( f' \) in \( cn''_1 \) by well-formedness.

Case 5: Assume \( o'' \in \pi_2(a_1(f)) \). Then \( f \) is unassigned and on the notification path of \( o \) in \( cn_1 \) and consequently also in \( cn_2 \). Since external forwardings and messages are the same in \( cn_1 \) and \( cn_2 \), the forwarding for \( o'' \) was added either by some internal message in \( cn \) or by a message or forwarding present in both \( cn_1 \) and \( cn_2 \). Consequently, \( o'' \in \pi_2(a_2(f)) \).

**Proposition 10.13.** \( \cong_2 \) is a type 2 witness relation.

**Proof.** By Lemma 10.11 and Lemma 10.12, it suffices to prove barb preservation in both directions. Assume without loss of generalization that \( A(cn_1) \rightarrow cn_1' \cong cn_2' \rightarrow A(cn_2) \). Suppose \( cn_1 \Downarrow \text{obs} \), where \( \text{obs} = \text{ext}!m(\overline{v}) \); then \( \text{msg} \preceq cn_1 \) with \( \text{msg} = \text{call}(\text{ext}, o, f, m, \overline{v}) \), whence \( \text{msg} \preceq cn_1' \). By the definition of extension, we then have \( \text{msg} \preceq cn_2' \), such that \( \text{msg} \) is in some link queue. It is then possible to repeatedly use the rule \text{MSG-DELAY-3} to reach a configuration \( cn_2'' \) where \( \text{msg} \) is at the top of that link queue. We thus have \( cn_2 \rightarrow^* cn_2' \rightarrow^* cn_2'' \Downarrow \text{obs} \), and conclude that \( cn_2 \Downarrow \text{obs} \). The proof of the converse property is symmetric.

**Lemma 10.14.** Suppose that \( cn' \) is WF2, and \( cn \cong cn' \). Then, \( cn \) is WF2 as well, and \( cn \cong_2 cn' \).

**Proof.** It suffices to prove that \( cn \) is WF2, since we then have that \( cn \cong_2 cn' \) by Proposition 10.9, which by Proposition 10.13 implies that \( cn \cong_2 cn' \).
Note that every WF2 clause except Single Writer and Future Liveness holds straightforwardly in \( cn \) from the assumption that \( cn' \) is WF2, since only future maps in object containers are different. For Single Writer, note that \( cn \) does not introduce mappings for unused futures, and does not introduce new messages. For Future Liveness, assume some \( f \) is active for \( o \) in \( cn \); then \( f \) is active for \( o \) in \( cn' \), and therefore, \( o \) is on the notification path of \( f \) in \( cn' \). Then, since \( cn \) does not remove any OIDs from forwarding lists or any instantiations of futures, \( o \) must still be on the notification path of in \( cn \) (possibly with fewer steps). \( \Box \)

**Lemma 10.15.** Let \( \varphi.cn \) be a WF1 configuration in standard form.

1. If bind \( \varphi.cn \rightarrow \varphi.cn' \), then for some \( cn'' \), net\((cn) \rightarrow.cn'' \equiv_2 \text{net}(cn') \).

2. If net\((cn) \rightarrow.cn'' \), then for some \( \varphi' \) and \( cn' \), bind \( \varphi.cn \rightarrow^* \varphi.cn' \) and \( cn'' \equiv_2 \text{net}(cn') \).

**Proof.** 1. We proceed by cases on the nature of the given type 1 transition. Let

\[
\text{bind } \varphi.cn \rightarrow \varphi.cn'. \tag{3}
\]

Fix \( cn_{\text{graph}} \) and name representation map \( \text{rep} \). As above we elide uses of \( \text{ctxt}-1 \) in both semantics by applying the rules to arbitrary configuration subsets, and we elide uses of \( \text{ctxt}-2 \) in the type 1 semantics, by considering transitions in arbitrary binding contexts. Each of the remaining transitions in Figure 4 immediately translates into a corresponding transition at type 2 level, and moreover, the resulting type 2 configuration is in normal form.

Consider for instance rule \( \text{wfield} \). We obtain a type 1 transition of the form

\[
\text{bind } \varphi.cn \circ(o,a) t(o,l,x = e; s) \rightarrow \text{bind } \varphi.cn \circ(o,a[v/x]) t(o,l,s)
\]

where \( x \in \text{dom}(a) \) and \( v = [e]_{(o,l)} \). Let \( v' = [e]_{(\text{rep}(o),\text{rep}(l))} \); then \( \text{rep}(v) = v' \) by (2). We obtain:

\[
\text{net}(cn \circ(o,a) t(o,l,x = e; s)) = \text{net}(cn,\text{rep}) \circ \text{net}(o(a,a),\text{rep}) \circ \text{net}(t(o,l,x = e; s),\text{rep}) \circ \text{net}(cn_{\text{graph}})
\]

\[
\text{= net}(cn,\text{rep}) \circ \text{net}(o(a,a),\text{rep}) \circ \text{net}(t(o,l,x = e; s),\text{rep}) \circ \text{net}(cn_{\text{graph}}))
\]

\[
\text{= net}(cn,\text{rep}) \circ \text{net}(o(a,a),\text{rep}) \circ \text{net}(t(\text{rep}(o),\text{rep}(l),x = e; s) \circ cn_{\text{graph}})
\]

\[
\text{= net}(cn,\text{rep}) \circ \text{net}(o(\text{rep}(o),\text{rep}(a),u_0,\varepsilon,\varepsilon) t(\text{rep}(o),\text{rep}(l),x = e; s) \circ cn_{\text{graph}})
\]

\[
\text{= net}(cn,\text{rep}) \circ \text{net}(o(\text{rep}(o),\text{rep}(a)[v'/x],u_0,\varepsilon,\varepsilon) t(\text{rep}(o),\text{rep}(l),s) \circ cn_{\text{graph}})
\]

\[
\text{= net}(cn,\text{rep}) \circ \text{net}(o(\text{rep}(o),\text{rep}(a[v/x]),u_0,\varepsilon,\varepsilon) t(\text{rep}(o),\text{rep}(l),s) \circ cn_{\text{graph}})
\]

\[
\text{= net}(cn \circ(o,a[v/x]) t(o,l,s))
\]

using the rule \( \text{wfield}-2 \) to derive the transition.

The remaining rules in Figure 4 are proved in the same manner, so we proceed to the rules in Figure 5.
CALL-SEND: Consider the following type 1 transition:

\[
\text{bind } \exists.cn \ o(a, a) \ t(o, l, x = e_1 m(\overline{v_2}) \ s) \\
\rightarrow \ \text{bind } f \exists.cn \ o(a, a) \ t(o, l[f/x], s) \ f(f, \bot) \ c(o', f, m, \overline{v})
\]

where \( o' = [e_1]_{[a,l]} \) and \( \overline{v} = [\overline{v_2}]_{[a,l]} \). We calculate:

\[
\text{net}(cn \ o(a, a) \ t(o, l, x = e_1 m(\overline{v_2}) \ s)) = (\text{net}(cn, \text{rep}) \circ \text{net}(o(a, a), \text{rep}) \circ \text{net}(t(o, l, x = e_1 m(\overline{v_2}); s), \text{rep}))(cn_{\text{graph}})
\]

\[
= \text{net}(cn, \text{rep})(\text{net}(o(a, a), \text{rep}))
\]

\[
= \text{net}(cn, \text{rep})(\text{net}(o(a, a), \text{rep}) \circ \text{net}(t(o, l, x = e_1 m(\overline{v_2}); s), \text{rep}))(cn_{\text{graph}}))
\]

\[
= \text{net}(cn, \text{rep})(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon))
\]

\[
\rightarrow o \cong_2 \text{net}(cn, \text{rep}')(o(\text{rep}'(o), \text{rep}'(a), u_0, \varepsilon, \varepsilon))
\]

\[
\text{call}(\text{rep}'(o), \text{rep}'(a), f', m, \text{rep}(\overline{v}))
\]

\[
\text{using CALL-SEND-2, and where } f' = \text{newf}(u_0) \text{ and } \text{rep}' = \text{rep}[f'/f].
\]

CALL-RCV: Consider the following type 1 transition:

\[
\text{bind } \exists.cn \ o(a, a) \ c(o, f, m, \overline{v}) \rightarrow \text{bind } \exists.cn \ o(a, a) \ t(o, l, s)
\]

where \( l = \text{locals}(o, f, m, \overline{v}) \) and \( s = \text{body}(o, m) \). We calculate:

\[
\text{net}(cn \ o(a, a) \ c(o, f, m, \overline{v}))
\]

\[
= (\text{net}(cn, \text{rep}) \circ \text{net}(o(a, a), \text{rep}) \circ \text{net}(c(o, f, m, \overline{v}), \text{rep}))(cn_{\text{graph}})
\]

\[
= \text{net}(cn, \text{rep})(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon))
\]

\[
\text{send}(\text{rep}(o), \text{rep}(a), f, m, \text{rep}(\overline{v}), \text{cn}_{\text{graph}}))
\]

\[
\equiv_2 \text{net}(cn, \text{rep})(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon))
\]

\[
\text{t}(\text{rep}(o), \text{rep}(l), s) \text{ cn}_{\text{graph}})
\]

\[
\text{RET: Consider the following type 1 transition:}
\]

\[
\text{bind } \exists.cn \ o(a, a) \ t(o, l, \text{return } \epsilon; s) \ f(f, \bot) \rightarrow \text{bind } \exists.cn \ o(a, a) \ f(f, v)
\]

63
where \( l(\text{ret}) = f \) and \( v = \llbracket e \rrbracket_{(a,l)} \). We calculate:

\[
\begin{align*}
\text{net}(cn \circ (o,a) t(o,l,\text{return } e; s) f(f, \perp)) &= \text{net}(cn, \text{rep})(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \\
& \quad t(\text{rep}(o), \text{rep}(l), \text{return } e; s) cn_{\text{graph}}) \\
& \rightarrow o \cong_2 \text{net}(cn, \text{rep})(o(\text{rep}(o), \text{rep}(a), \text{rep}(v)/\text{rep}(f)), u_0, \varepsilon, \varepsilon) cn_{\text{graph}} \\
& = \text{net}(cn \circ (o,a) f(f, v))
\end{align*}
\]

using \text{ret}-2.

**GET:** Consider the following type 1 transition:

\[
\begin{align*}
\text{bind } \exists.cn \circ (o, a) f(f, v) t(o, l, x = e.\text{get}; s) \\
& \rightarrow \text{bind } \exists.cn \circ (o, a) f(f, v) t(o, l[v/x], s)
\end{align*}
\]

where \( f = \llbracket e \rrbracket_{(a,l)} \). We calculate:

\[
\begin{align*}
\text{net}(cn \circ (o,a) f(f, v) t(o,l,x = e.\text{get}; s)) &= \text{net}(cn, \text{rep})(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \\
& \quad t(\text{rep}(o), \text{rep}(l), x = e.\text{get}; s) cn_{\text{graph}}) \\
& \rightarrow \text{net}(cn, \text{rep})(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \\
& \quad t(\text{rep}(o), \text{rep}(l)[\text{rep}(v)/x], s) cn_{\text{graph}}) \\
& = \text{net}(cn \circ (o,a) f(f, v) t(o,l[v/x], s))
\end{align*}
\]

using \text{get}-2.

**NEW:** Consider the following type 1 transition:

\[
\begin{align*}
\text{bind } \exists.cn \circ (o, a) t(o, l, x = \text{new } C(\overline{r}); s) \\
& \rightarrow \text{bind } a'. \exists.cn \circ (o, a) t(o, l[a'/x], s) o(a', a')
\end{align*}
\]

where \( \overline{r} = \llbracket r \rrbracket_{(a,l)} \) and \( a' = \text{init}(C, \overline{r}, a') \). We calculate:

\[
\begin{align*}
\text{net}(cn \circ (o,a) t(o,l,x = \text{new } C(\overline{r}); s)) &= \text{net}(cn, \text{rep})(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \\
& \quad t(\text{rep}(o), \text{rep}(l), x = \text{new } C(\overline{r}); s) cn_{\text{graph}}) \\
& \rightarrow o \cong_2 \text{net}(cn, \text{rep}')(o(\text{rep}'(o), \text{rep}'(a), u_0, \varepsilon, \varepsilon) \\
& \quad t(\text{rep}'(o), \text{rep}'(l)[\text{rep}'(v)/x], s) \\
& \quad o(a'', \text{rep}'(a'), u_0, \varepsilon, \varepsilon) cn_{\text{graph}}) \\
& = \text{net}(cn, \text{rep}')(o(\text{rep}'(o), \text{rep}'(a), u_0, \varepsilon, \varepsilon) t(\text{rep}'(o), \text{rep}'(l)[\text{rep}'(v)/x], s) \\
& \quad o(a'', \text{rep}'(a'), u_0, \varepsilon, \varepsilon) cn_{\text{graph}}) \\
& = \text{net}(cn \circ (o,a) t(o,l[a'/x], s) o(a', a'))
\end{align*}
\]

where \( \text{rep}' = \text{rep}[a''/a'] \) and we use (2) as usual. This completes the proof of property 1.

2. We proceed now by cases on the type 2 transition. Suppose \( \text{net}(cn) \rightarrow cn'' \), and we find \( \overline{r}', cn'' \) to complete the diagram as stated in the lemma. As
above, we apply the rules to configuration subsets, to elide uses of \texttt{CTXT}-1. Rules in the type 2 instance of Figure 4 are straightforward and omitted. For the rules in Figure 9 excepting \texttt{CALL-SEND-2}, \texttt{RET-2}, \texttt{GET-2}, and \texttt{NEW-2}, we can choose \( \tau' = \tau \) and \( cn' = cn \), since by Corollary 9.17, \( \texttt{net}(cn) \equiv_2 cn'' \).

For the four remaining cases, each case is obtained by reversing the arguments, i.e., proving that if the type 2 transition holds, depending on the rule application and shape of configurations, then also the corresponding type 1 transition holds, and the resulting pair of configurations are in \( \equiv_2 \). This completes the argument.

\begin{proof}
We exhibit a bisimulation relation \( \mathcal{R} \) defined as
\[
\mathcal{R} = \{(\text{bind } \tau.cn, cn') \mid \text{net}(cn) \equiv_2 cn'\}.
\]
where \( \text{bind } \tau.cn \) is a WF1 configuration in standard form, and \( cn' \) is a WF2 configuration. Note that \((\text{bind } \tau.cn, \text{net}(cn)) \in \mathcal{R}\), since the identity relation is included in \( \equiv_2 \). We show that \( \mathcal{R} \) is a witness relation. Clearly, \( \equiv_2 \) is reduction closed.

Suppose \( \text{bind } \tau.cn_1 \mathcal{R} cn_2 \) (or the converse for \( \mathcal{R}^{-1} \)); then \( \text{bind } \tau.cn_1 \) is WF1 and in standard form, \( cn_2 \) is WF2, and \( \text{net}(cn_1) \equiv_2 cn_2 \).

For reduction closure, assume \( \text{bind } \tau.cn_1 \rightarrow \text{bind } \tau'.cn'_1 \), where \( \text{bind } \tau'.cn'_1 \) is in standard form. Then, by property 1 of Lemma 10.15, \( \text{net}(cn_1) \rightarrow^* cn'_1 \equiv_2 \text{net}(cn'_1) \). This means that, for some \( cn'_2 \), \( cn_2 \rightarrow^* cn'_2 \) and \( cn'_2 \equiv_2 cn'_1 \). Hence, by transitivity of \( \equiv_2 \), \( \text{bind } \tau'.cn'_1 \mathcal{R} cn'_2 \). For converse reduction closure, assume \( cn_1 \rightarrow cn'_1 \). Then, \( \text{net}(cn_1) \rightarrow^* cn''_1 \) and \( cn'_2 \equiv_2 cn''_1 \). By property 2 of Lemma 10.15, this means that \( \text{bind } \tau.cn_1 \rightarrow^* \text{bind } \tau'.cn'_1 \) and \( cn'_2 \equiv_2 \text{net}(cn'_1) \). Hence, by the transitivity of \( \equiv_2 \), \( cn'_2 \mathcal{R}^{-1} \text{bind } \tau'.cn'_1 \).

For context closure, assume \( \text{bind}_\tau.cn_1.cn \) is WF1, and compose the type 2 interpretation of \( cn \) with \( cn_2 \) to produce \( cn_2.cn' \), using the representation map. We need to show that this configuration is WF2; to do this we assume that the interpretation and composition is done so that Object-Node Existence holds, i.e., the type 2 versions of objects in \( cn \) get a NID for a node that exists in graph(\( cn_2 \)). We also assume that Buffer Cleanliness is respected for messages added from \( cn \). For OID Uniqueness, it suffices to consider the case where \( o(a_1, a_2, u_1, q_{in,1}, q_{out,1}) \preceq cn_2 \) and \( o(a_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq cn' \); if \( a_1 = a_2 \), then there is a corresponding clash in \( cn_1.cn \), violating WF1. For Task-Object Existence, assume \( t(o, l, s) \preceq cn' \); then, if there is no \( o(a, a, u, q_{in}, q_{out}) \) \( cn_2.cn' \), there is no corresponding object container in \( cn_1.cn \), violating WF1. Local Routing Consistency holds since the composition does not add nodes or links and does not change routing tables. For Call Uniqueness, it suffices to consider the case where \( \text{call}(a_1, a'_1, f_1, m_1, \pi_1) \preceq cn_2 \) and \( \text{call}(a_2, a'_2, f_2, m_2, \pi_2) \preceq cn' \); if \( f_1 = f_2 \), then there must be a clash between the corresponding messages in \( cn_1 \) and \( cn \), which WF1 rules out. For Single Writer, a clash of identifiers in return futures for tasks and calls can again be traced back to a clash in \( cn_1.cn \), contradicting WF1. External OID holds since \( \text{ext} \) cannot be defined in WF1 configurations, and routing tables are.

\end{proof}

\begin{theorem}
For all well-formed type 1 configurations \( \text{bind } \tau.cn \) in standard form, \( \text{bind } \tau.cn \simeq \text{net}(cn) \).
\end{theorem}

\begin{proof}
We exhibit a bisimulation relation \( \mathcal{R} \) defined as
\[
\mathcal{R} = \{(\text{bind } \tau.cn, cn') \mid \text{net}(cn) \equiv_2 cn'\}.
\]
where \( \text{bind } \tau.cn \) is a WF1 configuration in standard form, and \( cn' \) is a WF2 configuration. Note that \((\text{bind } \tau.cn, \text{net}(cn)) \in \mathcal{R}\), since the identity relation is included in \( \equiv_2 \). We show that \( \mathcal{R} \) is a witness relation. Clearly, \( \equiv_2 \) is reduction closed.

Suppose \( \text{bind } \tau.cn_1 \mathcal{R} cn_2 \) (or the converse for \( \mathcal{R}^{-1} \)); then \( \text{bind } \tau.cn_1 \) is WF1 and in standard form, \( cn_2 \) is WF2, and \( \text{net}(cn_1) \equiv_2 cn_2 \).

For reduction closure, assume \( \text{bind } \tau.cn_1 \rightarrow \text{bind } \tau'.cn'_1 \), where \( \text{bind } \tau'.cn'_1 \) is in standard form. Then, by property 1 of Lemma 10.15, \( \text{net}(cn_1) \rightarrow^* cn'_1 \equiv_2 \text{net}(cn'_1) \). This means that, for some \( cn'_2 \), \( cn_2 \rightarrow^* cn'_2 \) and \( cn'_2 \equiv_2 cn'_1 \). Hence, by transitivity of \( \equiv_2 \), \( \text{bind } \tau'.cn'_1 \mathcal{R} cn'_2 \). For converse reduction closure, assume \( cn_1 \rightarrow cn'_1 \). Then, \( \text{net}(cn_1) \rightarrow^* cn''_1 \) and \( cn'_2 \equiv_2 cn''_1 \). By property 2 of Lemma 10.15, this means that \( \text{bind } \tau.cn_1 \rightarrow^* \text{bind } \tau'.cn'_1 \) and \( cn'_2 \equiv_2 \text{net}(cn'_1) \). Hence, by the transitivity of \( \equiv_2 \), \( cn'_2 \mathcal{R}^{-1} \text{bind } \tau'.cn'_1 \).

For context closure, assume \( \text{bind}_\tau.cn_1.cn \) is WF1, and compose the type 2 interpretation of \( cn \) with \( cn_2 \) to produce \( cn_2.cn' \), using the representation map. We need to show that this configuration is WF2; to do this we assume that the interpretation and composition is done so that Object-Node Existence holds, i.e., the type 2 versions of objects in \( cn \) get a NID for a node that exists in graph(\( cn_2 \)). We also assume that Buffer Cleanliness is respected for messages added from \( cn \). For OID Uniqueness, it suffices to consider the case where \( o(a_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn_2 \) and \( o(a_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq cn' \); if \( a_1 = a_2 \), then there is a corresponding clash in \( cn_1.cn \), violating WF1. For Task-Object Existence, assume \( t(o, l, s) \preceq cn' \); then, if there is no \( o(a, a, u, q_{in}, q_{out}) \) \( cn_2.cn' \), there is no corresponding object container in \( cn_1.cn \), violating WF1. Local Routing Consistency holds since the composition does not add nodes or links and does not change routing tables. For Call Uniqueness, it suffices to consider the case where \( \text{call}(a_1, a'_1, f_1, m_1, \pi_1) \preceq cn_2 \) and \( \text{call}(a_2, a'_2, f_2, m_2, \pi_2) \preceq cn' \); if \( f_1 = f_2 \), then there must be a clash between the corresponding messages in \( cn_1 \) and \( cn \), which WF1 rules out. For Single Writer, a clash of identifiers in return futures for tasks and calls can again be traced back to a clash in \( cn_1.cn \), contradicting WF1. External OID holds since \( \text{ext} \) cannot be defined in WF1 configurations, and routing tables are.

\end{proof}
intact from $cn_2$. For Future Liveness, if $f$ is active for $o$ in $cn_2 cn'$, a corresponding future is active in $cn_1 cn$, which either has future container with a value, or a task that is producing it; in the former case, we assume an entry is added to the future map of $o$ during composition, and in the latter case, we assume forwarding lists are extended when necessary, meaning that $o$ is on the notification path of $f$.

It remains to show that bind $\tau.cn_1 cn \mathcal{R} cn_2 cn'$ The network graphs of $\text{net}(cn_1 cn)$ and $cn_2 cn'$ coincide since $cn'$ does not introduce any new nodes or links. Clearly, if and only if a task or a message is in $cn$, a corresponding task or message is in $cn'$. We already have that the remaining tasks and tasks corresponding to messages in $cn_1$ and $cn_2$ coincide. As for external messages, either they are newly introduced via the context, and are thus in both composed configurations, or they come from the original configuration, and thus have a corresponding message. With respect to objects, the future maps of objects in the normal form of $\text{net}(cn_1 cn)$ are possibly extended when compared to future maps for objects in the normal form of $cn_2 cn'$, but in all other ways, objects in the normal forms are equal. Hence, $\text{net}(cn_1 cn) \equiv_2 cn_2 cn'$.

For converse context closure, assume $cn_2 cn$ is WF2, and compose the type 1 interpretation of $cn$, $cn'$, with $cn_1$ to produce bind $\tau.cn_1 cn'$ in standard form. We need to show that this configuration is WF1. For OID Uniqueness, it suffices to consider the case where $o(o_1, a_1) \preceq cn_1$ and $o(o_2, a_2) \preceq cn'$; note that if $o_1 = o_2$, there would have been an OID clash in $cn_2 cn$, which cannot be the case. For Task-Object Existence, assume $t(o, l, s) \preceq cn'$; then, if there is no $o(o, a)cn_1cn'$, there is no corresponding object container in $cn_2 cn$, violating the WF2 assumption. For Call Uniqueness, it suffices to consider the case where $c(o_1, f_1, m_1, t_1) \preceq cn_1$ and $c(o_2, f_2, m_2, t_2) \preceq cn'$; note that if $f_1 = f_2$, there would have been a WF2 Call Uniqueness violation in $cn_2 cn$. For Single Writer, clashes in future identifiers can be traced back similarly to clashes in $cn_2 cn$, violating WF2. For Future Existence, if $f$ is active for $o$ in $cn_1 cn'$, a corresponding future is active in $cn_2 cn$, meaning that by the WF2 property, there is a notification path to an assignment, task or message, which translates to there being a future container; and a call container or task container when appropriate in $cn_1 cn'$.

It remains to show that $cn_2 cn \mathcal{R}^{-1} \text{bind } \tau.cn_1 cn'$. The network graphs of $\text{net}(cn_1 cn')$ and $cn_2 cn$ coincide since $cn$ does not contain any new nodes or links. Clearly, if and only if a task or a message is in $cn$, a corresponding task or message is in $cn'$. We already have that the remaining tasks and tasks corresponding to messages in $cn_1$ and $cn_2$ coincide. As for external messages, either they are newly introduced via the context, and are thus in both composed configurations, or they come from the original configuration, and thus coincide. With respect to objects, the future maps of objects in the normal form of $\text{net}(cn_1 cn')$ are possibly extended when compared to future maps for objects in the normal form of $cn_2 cn$, but in all other ways, objects in the normal forms are equal. Hence, $\text{net}(cn_1 cn') \equiv_2 cn_2 cn$.

For barb preservation, assume bind $\tau.cn_1 \downarrow \text{obs}$. Then $\text{net}(cn_1) \downarrow \text{obs}$,

66
which by normal form equivalence yields \( cn_2 \downarrow obs \), as needed. For converse barb preservation, assume \( cn_2 \downarrow obs \). Then, by normal form equivalence, \( \text{net}(cn_1) \downarrow obs \), and consequently \( cn_1 \downarrow obs \), whereby \( cn_1 \downarrow obs \). \( \square \)