# Static Deadlock Prevention in Dynamically Configured Communication Networks

**Manuel Fähndrich**[1], **Sriram K. Rajamani**[2] **and Jakob Rehof**[3]

[1]*Microsoft Research Redmond*
`maf@microsoft.com`

[2]*Microsoft Research India*
`sriram@microsoft.com`

[3]*Technische Universität Dortmund and Fraunhofer-ISST*
`rehof@do.isst.fhg.de`

## Abstract

We propose a technique to avoid deadlocks in a system of communicating processes. Our network model is very general. It supports dynamic process and channel creation and the ability to send channel endpoints over channels, thereby allowing arbitrary dynamically configured networks.
Deadlocks happen in such networks if there is a cycle created by a set of channels, and processes along the cycle circularly wait for messages from each other. Our approach allows cycles of channels to be created, but avoids circular waiting by ensuring that for every cycle $C$, some process $P$ breaks circular waits by selecting to communicate on both endpoints involved in the cycle $C$ at $P$. We formalize this strategy as a calculus with a type system. Our type system keeps track of markers called *obstructions* where wait cycles are intended to be broken. Programmers annotate message types with design decisions on how obstructions are managed. Using these annotations, our type checker works modularly and independently on each process, without suffering any state space explosion.
We prove the soundness of the analysis (namely deadlock freedom) on a simple but realistic language that captures the essence of such communication networks. We also describe how the technique can be applied to a substantial example.

**Keywords:** Deadlock prevention, communicating systems, reconfigurable systems

## 1 Introduction

In this paper we study a problem in modular specification and analysis of concurrent, communicating processes with mobile channels, i.e., communication channels that can be passed in messages as formalized in, e.g., the pi-calculus [11]. Such systems are difficult to specify and analyze, because channel mobility allows programs to communicate in a dynamically evolving network topology.

Communicating systems based on message passing have been gaining practical importance over the past few years due to the emergence of web services, peer-to-peer algorithms, and in general by the desire to construct more loosely coupled and isolated systems. Our motivation for looking at message passing systems comes from working on the Singularity project [8, 7]. The Singularity project investigates ways to build reliable systems based on modern programming languages and sophisticated program analyses. The operating system is built almost entirely in an extension of C$^{\#}$. Processes in this system are strongly isolated (no shared memory) and communicate solely via message passing.

We wish to avoid deadlocks in such systems. One approach is to use model checking [3] to systematically explore the state space of the entire system and ensure absence of deadlocks, but the state spaces of such programs are infinite due to dynamic channel and process creation capabilities, preventing direct use of model checking. Even if we restrict attention to finite number of channels and processes, the state space is exponential, limiting scalability. Thus, we need a *modular* design and analysis technique to avoid deadlocks for such programs.

If we restrict our attention to systems with static communication topologies, and assume that we do not have cycles created by channels, then we can specify communication contracts at module boundaries, and check modularly if processes conform to their contracts (see for example, [5]).

However both the above assumptions (static topologies, and absence of cycles) are too restrictive. As we illustrate below, it is impossible to write operating system components such as name servers without allowing dynamic topologies, and without allowing cycles of channels. We present an approach that allows cycles of channels to be created in dynamic topologies, but avoids circular waiting by ensuring that for every cycle $C$, some process $P$ breaks circular waits by selecting to communicate on on both endpoints involved in the cycle $C$ at $P$. We formalize this strategy as a calculus with a type system. Our type system keeps track of markers called *obstructions* where wait cycles are intended to be broken. Programmers annotate message types with design decisions on how obstructions are managed. Using these

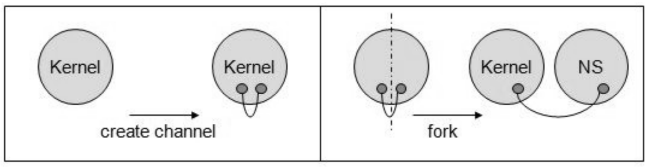

**Figure 1:** *Graphical illustration of channel creation and fork operations*

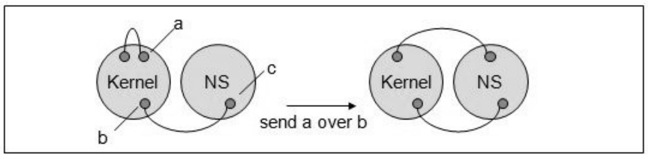

**Figure 2:** *Graphical illustration of endpoint send*

annotations, our type checker works modularly and independently on each process, without suffering any state space explosion.

We first provide an informal explanation of our communication model and how to prevent deadlocks using obstructions. In sections 2 and 3 we formalize a small programming language and its instrumented operational semantics. Section 4 presents our type system for static deadlock prevention, followed by a soundness theorem in Section 5. Section 6 describes an example to illustrate how obstructions and our type system can guide the design of a name service. Section 7 contains further discussion of the technical motivation behind our system and of future work. Sections 8 and 9 contain a discussion of related work and conclusions. Proofs of the main result are in the appendix.

## 1.1 Communication model

The communication model we are investigating in this paper is based on channels, where each channel consists of exactly two endpoints, and each endpoint is owned and operated on by at most one process at any time. This is in contrast to traditional $\pi$-calculus [11] style systems, where a shared name is all that is needed to communicate. Our endpoint-based approach is motivated by the fact that it is simpler for modular static verification and also corresponds more closely to actual implementations of message passing systems.

The communication model can be easiest understood using a graphical notation. A configuration consists of processes denoted by large circles and endpoints denoted by small circles. Channels are simply edges between endpoints. Endpoints appear nested within the boundary of the unique process that owns it.

There are three primary operations acting on a configuration: 1) channel creation, 2) process forking, and 3) sending of an endpoint over an existing channel. Other operations do not affect network topology and thus cycles and deadlocks, so we ignore them here. In Figure 1 on the left, we see a configuration consisting of a single process named Kernel. The Kernel process then creates a new channel, resulting in the second configuration, where the process owns both endpoints of the newly created channel. Observe that channel creation is a purely local operation.

On the right of Figure 1 we see how a process can fork (along the dotted line) resulting in a configuration with two distinct processes (here named Kernel and NS for name service). Note that a fork operation partitions the endpoints of the forking process so that each endpoint ends up in exactly one of the two resulting processes. Finally, Figure 2 shows how in the previous configuration, an endpoint $a$ moves over the channel formed by endpoints $(b, c)$. We consider moves to be atomic, corresponding to a synchronous communication model, where a send and corresponding receive operation synchronize. In Section 7, we briefly discuss how our approach extends to asynchronous communication models.

Through repeated applications of channel creation, forking, and sending of endpoints over channels, we can achieve arbitrary configurations of processes and endpoints. Figures 3 and 4 continue the progression of configurations from Figure 2, showing how the kernel can create processes with channels to the name server and how process P1 can obtain a direct channel to P2 by forwarding an endpoint of a locally created channel through the name service via two send operations.

## 1.2 Deadlock

The example configurations of the previous section make it clear that cycles in the process-channel graph arise both temporarily and sometimes as permanent parts of the configuration (e.g., the cycle P1, NS, P2). Thus, our strategy for avoiding deadlock is not to restrict networks to trees, or to restrict communication to a subset of tree edges of the graph. Instead, we allow arbitrary graphs, but track

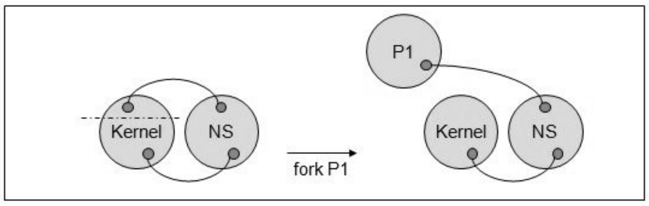

**Figure 3:** *Process creation with name service channel*

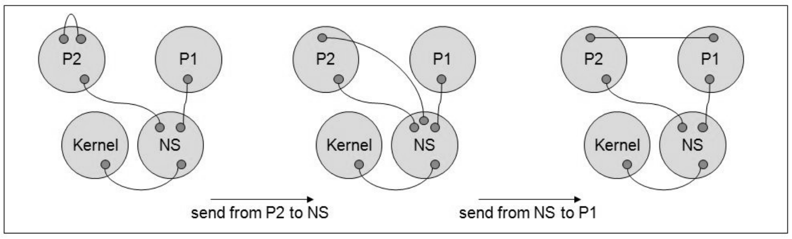

**Figure 4:** *Forwarding a channel through name server*

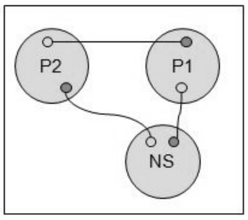

**Figure 5:** *A deadlock*

potential cycles and force processes to communicate in a way that prevents deadlock.

**Definition 1** *[Path] A path is a sequence of segments $s_1..s_n$, where each segment $s_i$ is of the form $(a_i, p_i, b_i)$, and $a_i$, $b_i$ are endpoints owned by process $p_i$, and each $(b_i, a_{i+1})$ forms a channel (for $i = 1..n-1$).*

We restrict our attention to primitive paths where no process (or endpoint) appears more than once.

**Definition 2** *[Cycle] A cycle $\gamma$ is a path $(a_1, p_1, b_1)...(a_n, p_n, b_n)$ where additionally, $(b_n, a_1)$ forms a channel.*

A channel is enabled for communication if both its endpoints are selected for communication by their respective owning processes. A set of processes is deadlocked if each process $p_i$ wants to communicate on a nonempty set $S_i$ of endpoints it owns, but no channel is enabled for communication. In other words, a deadlock arises if there exists no channel $(a, b)$, such that both $a$ and $b$ are selected by their respective owning processes, even though each process selects a nonempty set of endpoints to communicate.

Figure 5 shows a configuration with processes P1, P2, and the name service, where filled circles denote selected endpoints, and unfilled circles denote non-selected endpoints. No channel is enabled, since there is no channel with both endpoints being filled, even though each process has selected one endpoint. Therefore this configuration is deadlocked.

## 1.3 Obstructions

The main idea and contribution of the paper is to statically track potential cycles in the configuration and to prevent deadlock by forcing some process on each cycle to select enough endpoints for communication so as to guarantee the existence of an enabled channel.

We track potential cycles via *obstructions*. An obstruction is a marker in the form of a pair of endpoints $(a, b)$ owned by the same process $p$. An obstruction summarizes the potential existence of a cycle containing segment $(a, p, b)$. We say that an endpoint is obstructed if it participates in an obstruction.

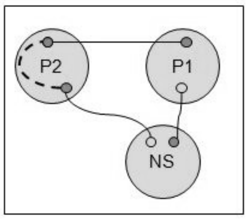

**Figure 6:** *P2 with obstruction and valid selection*

Figure 6 shows the same configuration as Figure 5, but where the cycle is covered by an obstruction maintained by process P2 between its two endpoints (depicted via a dashed edge). We can now prevent deadlocks by the following strategy for selecting endpoints:

**Definition 3** *[Valid selection] A process $p$ wanting to communicate has to select at least one unobstructed endpoint or at least one obstructed endpoint $a$ and all obstruction peers $\{b \mid (a, b)$ is an obstruction in $p\}$.*

Process P2 in Figure 6 is thus forced by the obstruction on its endpoints to select both endpoints for communication, even if it wants to communicate only with NS. As a result, the channel between P1 and P2 is now enabled and the configuration can evolve a step.

We prove two main parts in this paper: 1) the operational rules for maintaining local obstructions (and the corresponding type rules) are sound, meaning that in every reachable configuration, all cycles are covered by an obstruction. 2) given that all potential cycles are covered by some obstruction, the obstruction observing selection strategy is sufficient to guarantee that processes don't deadlock.

## 1.4 Limitations

Though our type system is modular, it makes heavy use of programmer supplied annotations. We require programmers to annotate message protocols, we also require programmers to provide obstruction annotations (see Section 3) whenever endpoints are sent and received over other endpoints. We are aware that supplying these annotations requires a deep understanding of our strategy for propagating obstructions in the operational semantics. We leave it to future work to come up with automatically inferring these annotations, and decrease the annotation burden on the programmer.

## 1.5 Outline

The remainder of the paper makes these ideas more precise and provides the following contributions: 1) an operational semantics instrumented with obstructions, 2) a static type system for tracking obstructions and restricting processes to obstruction observing communications, and 3) proofs of soundness of the obstruction tracking and selection strategy.

## 2 Language

Figure 7 defines a small language. A program consists of a set of protocol state declarations, message sequence declarations, process declarations, and a starting process term $P$.

A protocol state specifies a set of possible message sequences. A state with an empty set is a terminating state signaling the end of the protocol. A message declaration $m : \tau^t \rightarrow \sigma$ specifies that message $m$ carries an argument of type $\tau$ with obstruction tag $t$. Furthermore,

| | | | |
|---|---|---|---|
| *Program* | ::= | $\mathit{StateDecl}^{*}\ \mathit{SeqenceDecl}^{*}\ \mathit{ProcDecl}^{*}\ P$ | *program* |
| *StateDecl* | ::= | $\mathit{State} = \{m_1, m_2, \ldots, m_n\}$ | *protocol state* |
| *SeqenceDecl* | ::= | $m : \tau^{t} \rightarrow \sigma$ | *sequence declaration* |
| $\tau$ | ::= | $\mathbf{int} \mid \sigma$ | *parameter type* |
| $\sigma$ | ::= | $!\mathit{State} \mid ?\mathit{State}$ | *protocol* |
| $t$ | ::= | $r \mid s$ | *obstruction tag* |
| *State* | ::= | *Name* | *state name* |
| $m$ | ::= | *Name* | *message name* |
| *ProcDecl* | ::= | $PN(x_1 : \tau_1, \ldots, x_n : \tau_n) : O = P$ | *process declaration* |
| $P$ | ::= | $(\texttt{new}\ x : \sigma, y); P \mid \texttt{fork}\, P_1, P_2$ $\mid PN(E_1, E_2, \ldots, E_n)$ $\mid \texttt{select} \sum_{i \in I} G_i \mid (\texttt{free}\, E); P$ $\mid \texttt{halt}$ | *process* |
| $O$ | ::= | $\{(x_1, x_2), (x_3, x_4), \ldots\}$ | *obstructions* |
| $G$ | ::= | $E?m[x].P$ | *input guard* |
| | $\mid$ | $E_1!m[E_2].P$ | *output guard* |
| $E$ | ::= | $x \mid i \mid v$ | *expressions* |
| *PN* | ::= | *Name* | *process name* |
| $x$ | ::= | *Name* | *variable* |
| $v$ | ::= | *Name* | *value* |
| $i$ | ::= | integer | *integer* |

**Figure 7:** *Process terms and types*

it specifies the continuation $\sigma$ of the protocol for the sender of $m$. Thus, a message and a message sequence are isomorphic in our formalization. We use the notation next$(m)$ to refer to the continuation protocol of $m$.

The parameter type $\tau$ specifies the message argument to be either a value of type **int** or a channel endpoint with the given protocol $\sigma$. The protocol $\sigma$ of an endpoint describes the remaining interactions on the endpoint. A protocol $\sigma$ is either $!\mathit{State}$, specifying a send of any message in set *State*, or $?\mathit{State}$, specifying a receive of any message in set *State*.

By convention, the protocol continuation for any message is written from the perspective of the sender. The message sequences for the matching receiver is obtained by dualizing a protocol $\sim\sigma$ (turning sends into receives and vice-versa):

$$\begin{aligned} \sim !M &= ?M \\ \sim ?M &= !M \end{aligned}$$

Processes $P$ can create new channels via `new` or `fork` into two separate processes. Processes recurse by invoking process definitions. Select offers a set of communication alternatives, each with its own continuation. Endpoints are explicitly freed via `free`, and `halt` allows a process to terminate. We sometimes write $m![b]$ as a shorthand for `select` $m![b]$, and similarly $m?[x]$ as a shorthand for `select` $m?[x]$.

Obstructions $O$ are unordered pairs of endpoints. The obstruction set $O$ on process definitions is used solely by the static type system and specifies under what obstruction assumptions to type $P$ and applications of $P$.

The obstruction tag $t$ on message parameter types specifies whether some obstructions known to the sender on the endpoint argument are maintained by the sender $t = s$, or passed to the receiver $t = r$ after the exchange. The need for these tags is explained in the operational semantics.

Our language has two binders: (1) `new` and (2) message receive. We say that names $x$ and $y$ are bound in the process (`new` $x : \sigma, y); P$. We say that name $x$ is bound in the process $E?m[x].P$. A name that occurs in a process $P$, but is not bound in $P$ is said to be *free* in $P$. For a process $P$, we use $\text{fn}(P)$ to denote the set of all free names in $P$.

Given a process $P$, we use the notation $P[a/x]$ to denote the process obtained by substituting all free occurrences of $x$ in $P$ with $a$.

## 3 Operational Semantics

This section describes an operational semantics for processes that is instrumented to track the cycle obstructions our static type system reasons about. For each possible reduction, we show how the obstructions in the pre-state are related to the obstructions in the post-state. The semantics is presented as small step rewrite rules.

We use $x, y, z \in \mathcal{X}$ to range over bound variables, $u, v, w \in \mathcal{V}$ to range over values (or endpoints of channels), and $a, b, c \in \mathcal{X} \cup \mathcal{V}$ to range over either variables or values. The domain of values is needed to generate fresh names during application of the **new** operator while describing the operational semantics.

| | |
|---|---|
| $\langle \text{peer}, O, (\texttt{new}\ a : \sigma, b); P \mid \Pi \rangle$<br>$\longrightarrow \langle \text{peer}', O', P \mid P_{\sim\sigma}(c,d) \mid \Pi \rangle$ | $\text{peer}' = \text{peer} \cup \{(a,c),(c,a),(b,d),(d,b)\}$<br>$a,b,c,d \notin \text{dom}(\text{peer})$<br>$O' = O \cup \{(a,b)\}$ |
| $\langle \text{peer}, O, \texttt{fork}\, P_1, P_2 \mid \Pi \rangle$<br>$\longrightarrow \langle \text{peer}, O', P_1 \mid P_2 \mid \Pi \rangle$<br>**requires** $\text{fn}(P_1) \cap \text{fn}(P_2) = \{\}$ | $O' = (O \setminus O_{cut}) \cup O_{\text{cl}}$<br>$O_{cut} = \{(a,b) \mid a \in \text{fn}(P_1) \wedge b \in \text{fn}(P_2)$<br>$\wedge\ (a,b) \in O\}$<br>$O_{\text{cl}} = \{(b,d) \mid (a,b) \in O_{cut} \wedge (c,d) \in O_{cut}$<br>$\wedge\ b \neq d\}$ |
| $\langle \text{peer}, O, \texttt{select}\, b!m[a].P_1 + S_1$<br>$\mid \texttt{select}\, c?m[x].P_2 + S_2 \mid \Pi \rangle$<br>$\longrightarrow \langle \text{peer}, O', P_1 \mid P_2[a/x] \mid \Pi \rangle$<br>**requires** $(a,b) \notin O$ | $\text{peer}(b) = c \qquad m : \tau^t \to \sigma$<br>$O' = O \setminus a \cup O_{\text{sender}} \cup O_{\text{recvr}} \cup O_m$<br>$O_{\text{sender}} = \{(y,z) \mid (a,y) \in O \wedge (b,z) \in O\}$<br>$O_{\text{recvr}} = \{(a,z) \mid (c,z) \in O\}$<br>$O_m = \begin{cases} \{(c,a)\} & t = r \\ \{(b,y) \mid (a,y) \in O\} & t = s \end{cases}$ |
| $\langle \text{peer}, O, p(a_1 \ldots a_n) \mid \Pi \rangle$<br>$\longrightarrow \langle \text{peer}, O, P[a_1/x_1 \ldots a_n/x_n] \mid \Pi \rangle$<br>**requires** $a_i$ disjoint and $a_i \notin \text{fn}(P)$ | $p(x_1 \ldots x_n) \stackrel{\Delta}{=} P$ |
| $\langle \text{peer}, O, (\texttt{free}\, a); P \mid \Pi \rangle$<br>$\longrightarrow \langle \text{peer}', O', P \mid \Pi \rangle$<br>**requires** $a \in \text{dom}(\text{peer})$ | $\text{peer}' = \text{peer} \setminus a \qquad O' = O \setminus a$ |
| $\langle \text{peer}, O, \texttt{halt} \mid \Pi \rangle \longrightarrow \langle \text{peer}, O, \Pi \rangle$ | |

**Figure 8:** *Small-step operational semantics*

## 3.1 Configurations

Machine configurations are tuples of the form $\langle \text{peer}, O, \Pi \rangle$. The function peer: $\mathcal{V} \to \mathcal{V}$ is a bijection that associates endpoints with their channel peer, i.e., if $\text{peer}(a) = b$, then $(a,b)$ is a channel and $\text{peer}(b) = a$. The set $O \subseteq \mathcal{V} \times \mathcal{V}$ is the set of obstructions of the configuration. Finally, $\Pi$ is a parallel composition of processes, $\Pi = P_1 \mid \ldots \mid P_n$, considered modulo reordering of its components.

We only consider configurations that satisfy the invariant that any free name occurring in $\Pi = P_1 \mid \ldots \mid P_n$ occurs in exactly one process $P_i$. We write $\text{proc}(a)$ to denote the process in which the free name $a$ occurs. Furthermore, we are only interested in configurations where obstructions $O$ are local to a process, i.e., an obstruction is between endpoints $a$, and $b$ belonging to the same process.

$$\forall a,b.(a,b) \in O \implies \text{proc}(a) = \text{proc}(b) \tag{1}$$

The motivation for this restriction is due to the fact that the type system performs a per process modular analysis and thus cannot keep track of obstructions if they are not local to a process. Naturally, the type system in the next section enforces these invariants.

## 3.2 Reduction rules

Figure 8 contains small-step reduction rules for our language. We write dom$(R)$ to denote the domain of a binary relation $R$. If $O \subseteq \mathcal{V} \times \mathcal{V}$ is a set of pairs and $a \in \mathcal{V}$, then we write $O \setminus a$ for $O \setminus \{(a,b) \mid (a,b) \in O\}$.

If we ignore the treatment of obstructions, the reduction rules are essentially standard process reduction rules. The only complication arises in the creation of new channels. Instead of creating a direct channel between the two endpoints $a$ and $b$ of the `new` term, the operational semantics introduces a size one buffer process $P_{\sim\sigma}(c,d)$ (definition below) connected via channels $(a,c)$ and $(b,d)$. The buffer process executes message forwarding on $c$ according to protocol $\sim\sigma$ and on $d$ according to protocol $\sigma$. We chose to introduce a buffer process to prevent a process from trying to synchronize with itself. With buffer processes, we have the invariant that each process of the original program only communicates with buffer processes and vice versa. Thus, processes can never try to synchronize with themselves. In Figures, we continue to omit the buffer processes.

$$
\begin{aligned}
P_{?\{\}}(c,d) &\triangleq \texttt{free}\, c; \texttt{free}\, d; \texttt{halt}\\
P_{!\{\}}(c,d) &\triangleq \texttt{free}\, c; \texttt{free}\, d; \texttt{halt}\\
P_{?M}(c,d) &\triangleq \texttt{select} \textstyle\sum_{m\in M} c?m[x].d!m[x].P_{\sim \mathrm{next}(m)}(c,d)\\
P_{!M}(c,d) &\triangleq \texttt{select} \textstyle\sum_{m\in M} d?m[x].c!m[x].P_{\mathrm{next}(m)}(c,d)
\end{aligned}
$$

The next section provides insight into how the treatment of obstructions by the reduction rules provides strong enough invariants to prevent deadlock.

## 3.3 Properties

We first relate configurations $C$ to the graphs shown in the introduction.

**Definition 4** *[Induced graph] The induced graph $G(C)$ of a configuration*

$$C = \langle \mathrm{peer}; O; P_1 \mid \ldots \mid P_n \rangle$$

*is the graph $(N,E)$, where the set of vertices $N$ consists of a vertex per process and a vertex per endpoint, and the set of edges consists of an edge per channel $(a, \mathrm{peer}(a))$, and an edge linking each endpoint $a$ to its owning process* proc$(a)$.

Recall that the purpose of obstructions is to characterize all cycles in a configuration. The next definition makes this precise.

**Definition 5** *[Cycle coverage] Given an induced graph G of configuration* $\langle \text{peer}; O; P_1 \mid \ldots \mid P_n \rangle$*, we say that the* configuration is cycle covered, *if for each cycle* $\gamma$ *in G, there exists a segment* $(a,i,b) \in \gamma$*, such that* $(a,b) \in O$.

We now inspect some of the reduction rules of Figure 8 in more detail to see how they transform a configuration that is cycle covered into a new configuration that is also cycle covered, while maintaining only local obstructions.

Note that every initial configuration is of the form $\langle \emptyset; \emptyset; P \rangle$, where $P$ contains no free names. Initial configurations are thus cycle free and consequently cycle covered.

Rule `new` is the base case for creating a cycle. The new channels and the buffer process form a cycle containing two segments $(a,P,b)(d,P_{\sim\sigma},c)$. In order to cover this cycle in the resulting configuration, obstruction $(a,b)$ is added. Every other cycle in the resulting configuration exists in the prior configuration and is thus covered.

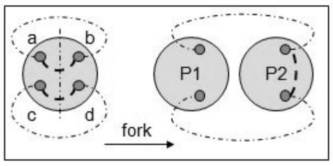

**Figure 9:** *Obstructions prior and after fork*

Rule `fork` has to consider what happens to the obstructions that are no longer local to a process after the split, namely those obstructions $(a,b)$, where $P_1$ retains $a$ and $P_2$ retains $b$. Figure 9 shows the situation for two such pairs of endpoints, $(a,b)$ and $(c,d)$. The dashed lines inside processes represent obstructions, the dot-dashed lines outside processes represent the potential path completing the cycle the obstruction covers. We see that after the fork, the two potential cycles have been turned into a larger single cycle. To cover it, one of the two processes needs to have an obstruction between its endpoints. The reduction rule for `fork` chooses $(b,d)$ in $P_2$ as shown in the picture. In general, cutting $n$ obstructions via `fork`, results in $\frac{n\cdot(n-1)}{2}$ new cycles and thus obstructions, one between every pair of paths connecting the two processes.

The most complicated case is the reduction for `select`. As can be seen from the rule in Figure 8, the obstructions $O'$ after the move of endpoint $a$ can be characterized by four groups. First, $O \setminus a$ are the obstructions not changed by the move. All obstructions in which $a$ is involved are removed because they are no longer local to the sending process when $a$ changes owner. The groups $O_{\text{sender}}$ and $O_{\text{recvr}}$ are closure rules that add obstructions in the sender and receiver necessary in the worst case to cover new potential cycles. To gain insight into these, we invite the reader to inspect the proof in the appendix. Here, we will focus on the last component, $O_m$.

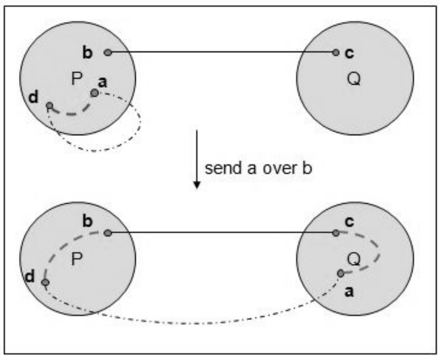

**Figure 10:** *Obstructions $O_m$ for send*

Consider the top configuration in Figure 10. We are about to send $a$ over $b$. It so happens that $(a,d)$ is an obstruction, and that there might thus be a path from $a$ to $d$ in the graph. Consider that same path after the move (bottom of figure). We observe that we have added the channel edge $(b,c)$ to the cycle. Since $(a,d)$ is no longer a valid obstruction, we must find a new obstruction that covers this cycle. We now have two options to cover this cycle: either by obstruction $(d,b)$ maintained by the sender, or $(c,a)$ by the receiver. Both are highlighted in the figure, but only one is needed. The choice between these to possibilities is under programmer control. Message signatures contain an obstruction tag $t \in r,s$ on the message argument type. This tag provides the contract between the sender and receiver as to who witnesses the new cycle via an obstruction. If the tag is $s$, only obstruction $(b,d)$ is added; if the tag is $r$, only obstruction $(c,a)$ is added. In both cases, the cycle in question is covered.

## 3.4 Stuck configurations

A configuration $C$ is stuck, if there does not exist $C'$, such that $C \longrightarrow C'$ by one of the reduction rules. We group stuck configurations into two groups: *locally stuck* processes and *globally stuck* configurations. Locally stuck processes are stuck independently of other processes, whereas globally stuck configurations are stuck due to the interaction of processes. By inspection of the rules, we find that processes can get locally stuck due to the side-conditions in the reduction rules that check non-sharing of endpoints between processes and rule out sending an endpoint over a channel with which it is obstructed.

A configuration is globally stuck, if no process is locally stuck, and the configuration isn't terminal (no more processes). In that case, all processes are of the form `select`. Configurations are globally stuck on `select` for the following reasons:

1. Deadlock: no two processes want to communicate on the same channel
2. Wrong message: there exists a channel with both sides ready to communicate, but the sides do not agree on a common message.

The deadlock case naturally includes the situation where a channel is orphaned, namely one side deletes its endpoint or terminates, but the other still tries to communicate. Note that due to our channel buffer processes, it is not possible for a process to get stuck with itself.

While the type system presented in the next section guarantees stuck-freedom from all of the above, the formal treatment in this paper shows that our novel use of obstructions avoids deadlock. The remaining cases are treated using established techniques, such as linear typing of endpoints.

## 4 Obstruction Type System

The type system validates judgments of the form

$$E;O \vdash P$$

meaning that, in endpoint environment $E$, and obstructions $O$, the process term $P$ is well typed.

We use $M,N$ to denote state names and use them interchangeably with the message sets they denote. We use $\delta$ as a meta-variable for either ! or ?. Environments

$$E ::= x{:}\tau \mid E,E \mid \cdot$$

map names to types $\tau$, which can be either base type (here just **int**), or an endpoint in a particular protocol state $\sigma$. In the latter case, $\sigma$

describes the remaining conversation on the endpoint. In order to track endpoint ownership by processes in the type system, we treat environments using a linear typing discipline. Every endpoint in the environment is owned. In principle, non-endpoint names do not need to be treated linearly. To avoid making the formalisms more complex however, we focus mostly on endpoints and thus all values are treated linearly. In practice, the environment would be split into a linear part for endpoints, and a non-linear part for other values. Obstructions

$$O ::= (x,y) \mid O,O \mid \cdot$$

correspond to the obstructions in the operational semantics, with a subtle difference that in the case of judgments, obstructions are kept in terms of variables in the process instead of endpoint values as in the operational semantics.

Figure 11 shows the type rule for each process term. It is interesting to note that the type system models the operational semantics precisely, except that it abstracts away the channel edges and actual values generated at runtime. In other words, the type system does not contain any information about which endpoints are peers, and the type system does not keep track of fresh values generated by the **new** operator.

The notation $ep(G)$ represents the endpoint and message on which $G$ communicates, i.e., $ep(a?m[x].P) = (a,m)$ and $ep(a!m[b].P) = (a,m)$. We use $dom(R)$ for the domain of a binary relation $R$ and $R(a)$ the range of $R$ w.r.t. $a$. This notation is used in the T-SELECT rule.

We briefly explain each rule. The `halt` process is well-typed only in the empty environment. This guarantees that processes do not halt while holding some endpoint on which the peer expects more communication. Freeing an endpoint requires that the endpoint is owned and has no conversation left (type $\delta\{\}$). Note that by rules T-HALT and T-FREE the type system thus enforces that all endpoints are freed explicitly.

The modularity of the analysis is most explicit in the rule T-FORK. After the fork, the analysis of the two sub-processes is completely independent and governed only by the concise endpoint types and message declarations that programmers can write. Rule T-FORK splits the environment into $E_1$ and $E_2$, thereby guaranteeing a partitioning of the endpoints owned by the two resulting processes. Similarly, we partition the obstructions into $O_1$, $O_2$, and $O_{\text{cut}}$, where $O_i$ are all obstructions on endpoints in $E_i$ $(i = 1,2)$. After the fork, obstructions $O_{\text{cut}}$ would no longer correspond to local obstructions, since they relate endpoints of distinct processes. They are thus removed from the

$$\cdot;\cdot \vdash \texttt{halt} \qquad [\text{T-HALT}]$$

$$\frac{E, x{:}\sigma, y{:}{\sim}\sigma; O, (x,y) \vdash P}{E; O \vdash (\texttt{new}\, x{:}\sigma, y).P} \qquad [\text{T-NEW}]$$

$$\frac{\begin{array}{c} m \in M \quad m : \tau^t \to \sigma \\ O_{\text{recvr}} = \{(b,x) \mid (b,a) \in O\} \\ O_m = \begin{cases} \emptyset & \text{if } t = s \\ \{(a,x)\} & \text{if } t = r \end{cases} \\ E, x{:}\tau, a{:}{\sim}\sigma; O \cup O_{\text{recvr}} \cup O_m \vdash P \end{array}}{E, a{:}?M; O \vdash a?m[x].P} \qquad [\text{T-INP}]$$

$$\frac{\begin{array}{c} m \in M \quad m : \tau^t \to \sigma \quad (a,b) \notin O \\ O_{\text{sender}} = \{(y,z) \mid (a,y) \in O \land (b,z) \in O\} \\ O_m = \begin{cases} \{(b,y) \mid (a,y) \in O\} & \text{if } t = s \\ \emptyset & \text{if } t = r \end{cases} \\ E, b{:}\sigma; O \setminus a \cup O_{\text{sender}} \cup O_m \vdash P \end{array}}{E, a{:}\tau, b{:}!M; O \vdash b!m[a].P} \qquad [\text{T-OUT}]$$

$$\frac{\begin{array}{c} \delta \in \{!,?\} \\ E; O \setminus a \vdash P \end{array}}{E, a{:}\delta\{\}; O \vdash \texttt{free}\, a.P} \qquad [\text{T-FREE}]$$

$$\frac{\begin{array}{c} O_{\text{cut}} \subseteq \text{fn}(E1) \times \text{fn}(E2) \\ O_{\text{cl}} = \{(b,d) \mid (a,b) \in O_{\text{cut}} \land (c,d) \in O_{\text{cut}} \land b \neq d\} \\ E_1; O_1 \vdash P_1 \quad \text{fn}(O_1) \subseteq \text{fn}(E_1) \\ E_2; O_2, O_{\text{cl}} \vdash P_2 \quad \text{fn}(O_2) \subseteq \text{fn}(E_2) \end{array}}{E_1, E_2; O_1, O_2, O_{\text{cut}} \vdash \texttt{fork}\, P_1, P_2} \qquad [\text{T-FORK}]$$

$$\frac{\begin{array}{c} R = \bigcup_i \mathit{ep}(G_i) \quad S \subseteq \mathit{dom}(R) \\ \mathit{ValidSelection}(S, O) \\ \forall a \in S.\, \mathit{Exhaustive}(R(a), E(a)) \\ E; O \vdash G_i \quad (i = 1 \ldots n) \end{array}}{E; O \vdash \texttt{select} \sum_i G_i} \qquad [\text{T-SELECT}]$$

$$\begin{array}{l} \mathit{ValidSelection}(S,O) = \exists a \in S.\; a \notin O \lor \forall b.(a,b) \in O \implies b \in S \\ \mathit{Exhaustive}(N, !M) = \mathit{true} \;\; \text{if } N \subseteq M \\ \mathit{Exhaustive}(N, ?M) = \mathit{true} \;\; \text{if } \mathrm{N} = \mathrm{M} \end{array}$$

$$\frac{\begin{array}{c} p(x_1{:}\tau_1 .. x_n{:}\tau_n) : O \triangleq P \\ x_1{:}\tau_1, .., x_n{:}\tau_n; O \vdash P \\ O' \subseteq O[a_i/x_i] \end{array}}{a_1{:}\tau_1, .., a_n{:}\tau_n; O' \vdash p(a_1 .. a_n)} \qquad [\text{T-INVOKE}]$$

**Figure 11:** *Obstruction type rules*

obstruction set. Instead, we compute a new set of obstructions $O_{\text{cl}}$ for process $P_2$ covering all resulting cycles (see also Figure 9). The choice of obstructions $O_{\text{cl}}$ in the type rule is just one of many possible designs. Alternative type rules could equally well compute a set of obstructions for $P_1$, or distribute obstructions among $P_1$ and $P_2$. We have not yet investigated the need for such alternatives.

Rule T-SELECT checks that the process communicates on a valid selection of endpoints $S$ that is obstruction observing. The *ValidSelection* predicate formalizes Definition 3. For each endpoint $a$ in $S$, the rule then enforces that receives are exhaustive, namely that the process is ready to receive all possible messages that the type of the endpoint (that is, the corresponding message state) specifies. For sends, it suffices to choose any subset of possible messages that the type of the endpoint specifies. Sends in message sequence types correspond thus to internal choice, receives to external choice.

Rule T-INP checks that $m$ is a valid message to receive given the type $?M$ of the receiving endpoint $a$. This is checked by ensuring that there is a message $m : \tau^t \rightarrow \sigma$ in the message state $M$. Since the message sequence for state $M$ is written (by convention) from the perspective of the sender, and endpoint $a$ is a receiver with type $?M$, we have to complement the polarity of $\sigma$ to obtain $\sim\sigma$ and type the remaining process $P$ where endpoint $a$ has type $\sim\sigma$. We also check that the actual argument $x$ has type $\tau$ as specified by the message signature. Obstructions consist of prior obstructions $O$ plus obstructions $O_{\text{recvr}}$ required on the receiver side between the received endpoint $x$ and obstruction peers of the receiving endpoint $a$. In addition, if the message obstruction tag $t = r$, then the receiver keeps track of the obstruction $(a, x)$ between the received and receiving endpoints.

Rule T-OUT checks that $m$ is part of $M$, the messages that can be sent on $b$. The remaining process $P$ is typed in the environment where $b$ has type $\sigma$ corresponding to the remaining conversation after sending $m$. Endpoint $a$ is no longer in the environment, since ownership of $a$ has passed to the owner of the peer of $b$. Obstructions consist of prior obstructions $O$, albeit without any obstructions mentioning $a$, followed by necessary obstructions $O_{\text{sender}}$ between obstruction peers of $a$ and $b$. If the obstruction tag $t = s$, then the sender additionally turns obstructions on $a$ into obstructions on $b$ (see also Figure 10).

The condition $(a, b) \notin O$ on T-OUT enforces that we can never send $a$ over $b$, if $a$ is obstructed with $b$. Were we to permit such an operation, it would have the effect of shortening a potential cycle in the graph by the one segment that contains the obstruction, but without being able to add the obstruction to any remaining segments. The receiving process would have to assume that the received endpoint is obstructed with all other endpoints owned by the process. We chose to simply disallow such sends rather than adding this conservative assumption.

## 5 Soundness

We present a soundness theorem establishing that if a configuration is typable by the obstruction typing rules then it cannot deadlock.

Recall that all cycles considered in this paper are primitive. We do not need to consider non-primitive cycles, since every dead-lock configuration exhibits a primitive wait cycle.

**Definition 6** *[Valid configuration] A configuration* $\langle \text{peer}; O_1..O_n; P_1 \mid \ldots \mid P_n \rangle$ *is valid if*

1. *there exist environments* $E_1..E_n$*, such that* $E_i; O_i \vdash P_i$
2. *for any channel* $(a, b)$*, such that* $\text{peer}(a) = b$*,* $a \in E_i$ *and* $b \in E_j$*, the endpoint states agree, i.e., if* $E_i(a) = !M$*, then* $E_j(b) = ?M$*, and if* $E_i(a) = ?M$*, then* $E_j(b) = !M$*.*
3. *the configuration is cycle covered (Definition 5)*

**Lemma 7** *[Preservation] Given a valid configuration* $C_1$ *and a reduction step* $C_1 \rightarrow C_2$*, configuration* $C_2$ *is valid.*
***Proof*** By case analysis over the possible redexes of a configuration. Preservation of points 1, and 2 of definition 6 is straight-forward. Appendix A.1 contains the graph theoretic argument for preservation of cycle coverage.

The following lemma states that given a valid configuration where all processes want to communicate, there exist two processes connected by a channel and both processes are ready to exchange a message over that channel.

**Lemma 8** *[Progress] Given a valid configuration* $C_1$*, then either* $C_1$ *is final and of the form* $\langle \emptyset; \emptyset; \emptyset \rangle$*, or there exists a configuration* $C_2$*, such that* $C_1 \rightarrow C_2$*.*
***Proof*** By case analysis over stuck configurations. Appendix A.2 contains the proof.

**Theorem 9** *[Soundness] Every closed program* $P$ *that is well typed* $\cdot; \cdot \vdash P$ *either reduces to the final configuration, or it reduces ad-infinitum.*
***Proof*** By induction over the reduction steps and Lemmas 7 and 8.

## 6 Name Server Example

A driving motivator for the present work is the design of a realistic name service for our research operating system. The name service process is a natural convergence point of many channels. It is also common to have cycles involving the name server as shown in the final configuration of Figure 4.

```
CLIENT = {NewClient, Bind}
ACKCLIENT = {AckClient}
NewClient :  (?CLIENT)^r →?ACKCLIENT
Bind      :  (?SERVICE)^r →?ACKCLIENT
AckClient :  void → ?CLIENT

REGISTRAR = {NewReg, Register}
NewReg    :  (?REGISTRAR)^s →?{AckReg}
Register  :  (?SERVICEPROVIDER)^s →?{AckReg}
AckReg    :  void → ?REGISTRAR

SERVICEPROVIDER = {Connect}
Connect   :  (?SERVICE)^r →?{AckConnect}
AckConnect: void → ?SERVICEPROVIDER
```

**Figure 12:** *Contracts for the name server endpoints*

In designing the name service, one could attempt to have the name service handle all obstructions to cover such cycles and be responsive enough so that other clients need not deal with obstructions. It turns out, though, that such a strategy is only partially feasible. Keeping all obstructions on the name server leads to a situation where all its client endpoints are obstructed with all service provider endpoints. This is not a desirable situation, since it prevents the name server from forwarding connection endpoints it receives over a client channel to a service, due to the fact that the connection endpoint will be obstructed with all services upon receipt.

Instead, we use the design shown graphically in Figure 14. Figure 12 shows the contract definitions for the end points of the name server and Figure 13 contains the server code itself.

The name server maintains four sets of end points, called clients, ackclients, registrants, and serviceproviders. Sets are not part of our core language but are straight-forward to implement. The sets collect end points that share the same type (message state). Endpoints can be added to sets and selections can involve sets of endpoints. For example, the guard **case** e.NewClient?( newclient ) **in** clients is triggered by a NewClient message from any member of the set clients. The particular endpoint e on which the message is received is bound in the **case** block and removed from the set. We use the type **void** for message arguments when the message does not carry any value although this is not part of the formalism.

The main function of the name server is to bind service endpoints to service providers. The name server supports this on endpoints in the client set. These endpoints have type ?CLIENT, specifying the

```
set<?CLIENT> clients ;
set<!ACKCLIENT> ackclients;
set<?REGISTRAR> registrants ;
set<?SERVICEPROVIDER> serviceproviders;
 ...
/*  initialize  clients , ackclients ,  registrants  and  service  providers */
 ...
while(true) {
  select {
    case e.NewClient?( newclient ) in  clients :
      // e, newclient , clients , and ackclients  are mutually  obstructed
       clients .Add( newclient );
       ackclients .Add(e );
      break;

    case e. AckClient !() in  ackclients :
      // e, clients , and ackclients are mutually  obstructed
       clients .Add(e );
      break;

    case e. Bind ?( service ) in  clients :
      // e, service , clients , and ackclients  are mutually  obstructed
       ackclients .Add(e );
      // service , clients , and ackclients  are mutually  obstructed
       serviceprovider  = GetServiceProvider ( serviceproviders ,  ...);
      //  serviceprovider  is not obstructed with any endpoints
       serviceprovider .Connect!( service );
      // clients , and ackclients  are mutually  obstructed
       serviceprovider .AckConnect ?();
       serviceproviders .Add( serviceprovider );
      break;

    case e.NewReg?(newreg) in  registrants :
      // neither e nor newreg is obstructed with any other endpoints
      e.AckReg !(); // can send AckReg on e alone
       registrants .Add(newreg);
       registrants .Add(e );
      break;

    case e. Register ?( serviceprovider ) in  registrants :
      // neither e nor  serviceprovider  is obstructed
      e.AckReg !(); // can send AckReg on e alone
       serviceproviders .Add( serviceprovider );
       registrants .Add(e );
      break;
  }
}
```

**Figure 13:** *Name server code*

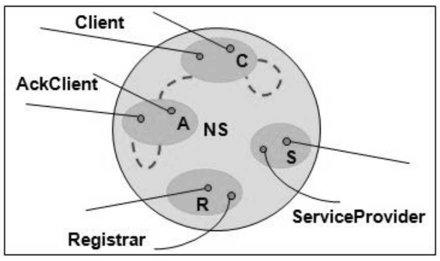

**Figure 14:** *Stable configuration of nameserver*

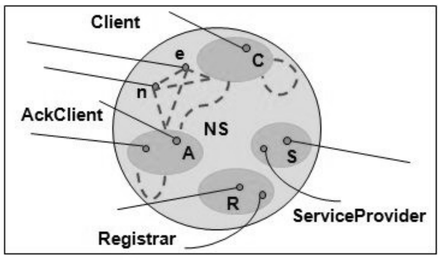

**Figure 15:** *Nameserver after receiving client request*

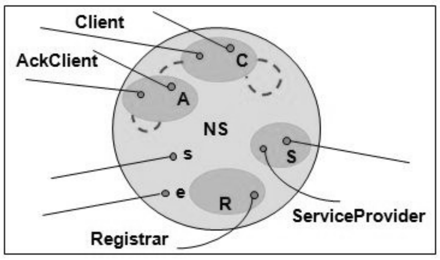

**Figure 16:** *Nameserver after receiving registrar request*

arrival of either message NewClient to add a new client, or message Bind to bind a service endpoint to a service provider. Both messages specify that obstructions on the message argument are handled by the receiver (the $r$ superscript). Thus, processes can obtain duplicate name service channels or bind to service providers without being encumbered by obstructions on their side.

Registrations of service providers happen on a different set of endpoints stored in the registrars set. These endpoints have type

?REGISTRAR and the name server receives either message NewReg to add a new registrar client, or message Register to register a new service provider. In contrast to the messages on the clients set, the messages received from registrants are tagged with superscript $s$, making it part of the channel contract that obstructions on these message arguments are kept on the sender.

Service providers are kept in set serviceproviders and have type ?SERVICEPROVIDER. They expect a single message Connect which passes a service endpoint to the service provider. The final set ackclients will be explained below.

Figure 14 shows the four sets as shaded areas labeled C, R, S, and A. The figure shows that our design allows endpoints in clients and ackclients to be mutually obstructed, but registrar or service provider endpoints are not obstructed with any other endpoint.

Consider the first case of the **select** statement on line 10 in the name server implementation. It handles NewClient messages carrying a new endpoint newClient arriving on some endpoint e in set clients. Because the message declaration is tagged with $r$, endpoints e and newClient are obstructed after the receive. Additionally, due to the receiver closure rule, newClient is also obstructed with all obstruction peers of e, namely all endpoints in sets clients and ackclients. This is graphically shown in Figure 15 where newClient is abbreviated with n. Next, the newClient endpoint is added to the clients set (it has the same type and obstructions as all other endpoints in that set). Now the nameserver needs to send an acknowledgment message on e as prescribed by the message declaration. However, offering to send only on e at this point is not allowed by the obstruction rules (not a valid selection), since e is obstructed with all other endpoints in sets clients and ackclients. In order to offer the acknowledgment on e, we also have to simultaneously accept messages on all client endpoints. We achieve this by storing e in set ackclients and re-executing the select statement. The set ackclients thus contains all client endpoints whose peers may be expecting the AckClient message. This is handled by the case on line 16.

After receipt of Bind on line 21, the obstruction situation is analogous to the previous case shown in Figure 15 but where n represents the service endpoint. The service endpoint is forwarded to some service provider from the service provider set. Since the service provider is not obstructed, we can send the Connect message and receive the subsequent AckConnect without having to consider offering to handle messages on other endpoints. No obstructions from the service endpoint are retained by the name server, since the Connect message specifies that the receiver (the service provider) handles obstructions.

Thus, after adding e to the ackclient set and the serviceprovider back into the serviceproviders set, we have again reached the stable situation in Figure 14 and can re-execute the loop.

Figure 16 shows the obstruction state that holds after receiving a Register message on endpoint e with service endpoint argument s (line 39). In this case, no obstructions are present among these endpoints. Thus, we can send the acknowledgment on e directly without the need for a set like ackclients . The stable state is reached again by adding endpoint e back to the set registrars , and adding s to set serviceproviders .

## 6.1 Implementation

We have implemented a name service according to the design described in Section 6, as part of the Singularity operating system [8, 7]. The name service provides a uniform name space for all services on a system, including device drivers, network connections and file system. Singularity is written in an extension to C# which includes message-passing primitives and contract specifications as discussed in this paper. Our notion of obstructions was motivated by our experiences in programming Singularity and will be implemented as an extension to the current type system.

## 7 Discussion and Future Work

The type system proposed in this paper has two aspects, message signatures and obstructions. Message signatures can be used to encode simple contracts (stateful communication protocols, somewhat in the style of session types [6]) specifying temporal ordering of types of messages sent and received along a pair of endpoints. In this paper, we wish to focus on the use of obstructions and have deliberately chosen a very simple contract format, in which a specification $\tau$ has a natural inverse, $\sim\tau$. In principle, contracts over message types could be much more expressive. For example, channel contracts used in [9, 2] are CCS processes.

Local and modular reasoning about deadlock in the presence of channel passing is a fundamental challenge. The present work arose partly out of an attempt to apply the theory of conformance developed in [5] to programs with dynamic communication topology. Conformance theory [5] allows us to reason compositionally about deadlock and unhandled message types arising from misuse of contracts, but the theory is restricted to CCS (static topology). Type systems in the style of [9, 2] can be used to extract CCS models from $\pi$-calculus expressions, which could in turn be processed by a model checker. Hence, by composing the theory of conformance with such type systems we could hope to achieve a modular system for typing deadlock-free components with very rich contracts and dynamic topology. It remains an important challenge for future work to devise a system in which a theory of contract conformance such as [5] is integrated with the obstruction discipline.

In this paper we have chosen to factor the deadlock problem into two orthogonal subproblems, namely, deadlock problems arising due to cyclic structures created dynamically by channel passing, and deadlock problems arising from misuse of contracts. Our basic rationale has been that we want contracts to be local to a channel (pair of endpoints). In contrast, the systems [9, 2] admit cross-channel contracts that can express constraints on messaging actions that take place on multiple channels. The drawback with such contracts from a pragmatic perspective is that $(i)$ they may need to span arbitrarily large parts of a system in practice, conflicting with the desire for modularity, and $(ii)$ it is difficult, in general, to find a natural place to state them in programs. It is possible that cross-channel contracts will be needed in practice to accommodate more flexible programming and a wider span of communication topologies, in which actions on multiple channels are correlated in ways that are not allowed by our type system. However, even so, we feel that it is valuable to have a basic, simpler system that works well for channel-local contract specifications, and it has been the goal of this paper to provide such a system.

### 7.1 Asynchronous communication

If we extend the buffer processes we introduce on our channels to buffer arbitrary numbers of messages instead of just one, we obtain an encoding for an asynchronous communication model of the programmer specified processes. Since the asynchrony is simulated by a synchronous system, the deadlock prevention result of this paper extends to asynchronous systems as well.

## 8 Related Work

The most novel aspect of our system is the use of obstructions to prevent global deadlock caused by dynamically created communication cycles by enforcing only local conditions (observation of the obstruction rule), while still allowing cycles in the communication topology to occur during computation.

In the context of the $\pi$-calculus [11], type systems have been proposed for analyzing and controlling the communication structure of mobile processes. Some of these systems capture partial orderings between communication actions and can be used to reason about deadlock. However, we have found no system that reduces the problem of global deadlock detection to locally checkable conditions on processes, and we have not been able to find a concept similar to that of obstructions in the literature. Moreover, our system controls the use of individual channel endpoints rather than $\pi$-calculus channels.

Our notion of contracts bears resemblance to session types [6]. Yoshida's graph types [12] are abstractions of the communication structure of a process in which nodes represent communication actions and edges represent synchronization orderings between actions. Kobayashi et.al. [9, 10] further develop process types to reason about deadlock via partial ordering constraints extracted from communication actions. Further comparisons between our system and the $\pi$-calculus type systems [9, 2] have been given in Section 7.

Flanagan and Abadi's type system [4] prevents deadlock due to locking in programs written in Java-like languages. The system is based on the specification of lock orderings that rule out cyclic wait conditions on locks.

Interaction categories [1] have been used to define type systems that enforce sufficient local control of communication to ensure deadlock-freedom. However, the type discipline does not in general handle global communication cycles.

## 9 Conclusion

We have developed a novel way to modularly characterize potential cycles in communication networks through endpoint obstruction pairs along with a strategy for communicating that is informed by such obstructions. Together, these techniques provably guarantee that a system of processes will not deadlock. We have used the notion of obstructions as a design guide for implementing a name service in the Singularity operating system, and we will extend our contract type checker to include obstructions.

## Acknowledgments

We are indebted to Bill McCloskey for his initial ideas on tainted endpoints.

## References

[1] S. Abramsky, S. Gay, and R. Nagarajan. A type-theoretic approach to deadlock-freedom of asynchronous systems. In *Theoretical Aspects of Computer Software, LNCS Vol. 1281*, pages 295–320, 1997.

[2] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *POPL 02: ACM Principles of Programming Languages*. ACM, 2002.

[3] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[4] C. Flanagan and M. Abadi. Types for safe locking. In *ESOP 99: European Symposium on Programming*, LNCS 1576, pages 91–108. Springer-Verlag, 1999.

[5] C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance. In *CAV 04: Computer-Aided Verification*, LNCS. Springer-Verlag, July 2004.

[6] S. Gay and V. T. Vasconcelos. Session types for inter-process communication. Technical Report 2003–133, Department of Computing, University of Glasgow, 2003.

[7] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[8] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad new OS research: challenges and opportunities. In *Proceedings of Tenth Workshop on Hot Topics in Operating Systems*. USENIX, June 2005.

[9] A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. In *POPL 01: Principles of Programming Languages*, pages 128–141. ACM, 2001.

[10] N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177:122–159, 2002.

[11] R. Milner. *Communicating and Mobile Systems: the $\pi$-Calculus*. Cambridge University Press, 1999.

[12] N. Yoshida. Graph types for monadic mobile processes. In *FSTTCS: Software Technology and Theoretical Computer Science*, LNCS 1180, pages 371–387. Springer-Verlag, 1996.

## Appendix

## A Proofs

### A.1 Lemma 7 (Preservation)

We prove that `new`, `fork`, and `select` reductions preserve cycle coverage. The other reductions are trivial. Assume $C_1 = \langle \mathrm{peer}_1; O_1; \Pi_1 \rangle$

and $C_2 = \langle \text{peer}_2; O_2; \Pi_2 \rangle$

$$\boxed{\begin{array}{l}\textbf{Case}\\ \Pi_1 = (\texttt{new}\, a{:}\sigma, b); P \mid \Pi\end{array}}$$

Recall that channel creation creates a buffer process $P_{\sim\sigma}(c,d)$ as defined in Section 3 in order to avoid having any process communicate with itself. Thus $\Pi_2 = P \mid P_{\sim\sigma}(c,d) \mid \Pi$. Let $\gamma$ be a cycle in $C_2$. If $\gamma$ does not contain any of $a, b, c, d$, then $\gamma$ exists also in $C_1$ and is therefore covered. Otherwise, $\gamma = (a,p,b)(d,q,c)$, where $q$ is the buffer process created for the channel. Since $(a,b) \in O_2$, the cycle is covered.

$$\boxed{\begin{array}{l}\textbf{Case}\\ \Pi_1 = \texttt{fork}\, P_1, P_2 \mid \Pi\end{array}}$$

Let $r$ be the process doing the fork, and let $p = P_1$ and $q = P_2$ be the resulting processes. Let $A = \text{fn}(P_1)$ and $B = \text{fn}(P_2)$. Let $\gamma$ be a cycle in $C_2$. We proceed with three cases: 1) $\neg\exists a \in A$ and $a \in \gamma$, 2) $\neg\exists b \in B$ and $b \in \gamma$, and 3) $\gamma$ contains an endpoint of $A$ and one of $B$.
**Case 1:** We have cycle $\gamma' = \gamma[r/q]$ in $C_1$ and an obstruction $(c,d) \in O_1$, where $c,d \notin A$. Thus $(c,d) \in O_2$, since the only obstructions removed by the step contain an endpoint from $A$.
**Case 2:** as case 1.
**Case 3:** $a,b \in \gamma$ **where** $a \in A$ **and** $b \in B$. Assume $\gamma$ is not covered in $C_2$, otherwise we are done. Without loss of generality, $\gamma = (a,p,x)\gamma_1(b,q,y)\gamma_2$. We have cycle $\gamma_1(b,r,x)$ in $C_1$ and by assumption it must be covered by $(b,x) \in O_{\text{cut}} \subseteq O_1$. Similarly, we have cycle $\gamma_2(a,r,y)$ in $C_1$ covered by $(a,y) \in O_{\text{cut}} \subseteq O_1$. Thus, $(b,y) \in O_{\text{cl}} \subseteq O_2$ covering $\gamma$ and contradicting our assumption.

$$\boxed{\begin{array}{l}\textbf{Case}\\ \Pi_1 = \texttt{select}\, b!m[a].P_1 + S_1 \mid \texttt{select}\, c?m[x].P_2 + S_2 \mid \Pi\end{array}}$$

Note that $\text{peer}_1 = \text{peer}_2$. The move involves moving an endpoint $a$ from a process $p$ to a process $q$ over a channel consisting of endpoints $b$ and $c$ such that $b = \text{peer}(c)$, $\text{proc}(b) = p$, and $\text{proc}(c) = q$. Consider any cycle $\gamma$ in $C_2$. We have three cases (1) endpoint $a$ is not in $\gamma$, (2) $a$ is in $\gamma$ and $c$ is in $\gamma$, and (3) $a$ is in $\gamma$ and $c$ is not in $\gamma$. We prove the theorem for these 3 cases.
**Case 1:** $a$ **is not in** $\gamma$. The only edge change between $C_1$ and $C_2$ is that we remove $(a,p)$ and add $(a,q)$. Since $a$ is not on $\gamma$, $\gamma$ exists in $C_1$ and is therefore covered by some obstruction in $O_1$ not involving $a$. The same obstruction is still present in $O_2$
**Case 2:** $a$ **is in** $\gamma$ **and** $c$ **is in** $\gamma$ Let $\gamma$ be of the form $\gamma_1(a,q,c)(b,p,e)$, where $\gamma_1$ starts with $(\text{peer}_1(e),..)$. We have two sub cases. If $\gamma_1$ is

covered by an obstruction in $C_2$ we are done. Otherwise, $\gamma_1$ does not have an obstruction in $C_2$. We know that $p$ cannot occur in $\gamma_1$, due to the fact that the cycle is primitive (recall that each process is traversed at most once in each cycle). Thus $\gamma_1$ is not covered by any obstructions in $C_1$. Thus, $(e,a) \in O_1$, since $\gamma_1,(a,p,e)$ is a cycle in $C_1$. Assume $m : \tau^t \to \sigma$. By the rewrite step we know that if $t = s$ then $(e,b) \in O_m \subseteq O_2$ covering $\gamma$. Otherwise $t = r$ and $(c,a) \in O_m \subseteq O_2$ covering $\gamma$.
**Case 3: $a$ is in $\gamma$ and $c$ is not in $\gamma$.** Let $\gamma$ be of the form $\gamma_1(f,q,a)$ in $C_2$, where $\gamma_1$ starts with $(\text{peer}_1(a),..)$. If sub-path $\gamma_1$ has an obstruction in $C_2$ we are done. Suppose $\gamma_1$ does not have any obstructions in $C_2$. Again we have two further sub cases. If $p$ is not in the path $\gamma_1$, then we know that the sub-path $\gamma_1$ in $C_1$ does not have any obstructions either. However, since $\gamma_1(f,q,c)(b,p,a)$ is a cycle in $C_1$, we have that it should be covered by some obstruction. This obstruction cannot be $(a,b)$ since that would preclude endpoint $a$ from being sent over $b$. Thus, $(c,f) \in O_1$, and consequently, $(a,f) \in O_{\text{recvr}} \subseteq O_2$ and we are done. For the second sub case, we have that $p$ is in the path $\gamma_1$. Then $\gamma_1 = \gamma_2(e,p,g)\gamma_3$, where $\text{proc}(e) = \text{proc}(g) = p$. Note that we therefore have cycles $\gamma_2$ and $(b,p,q)\gamma_3(f,q,c)$ in $C_1$ and therefore $(a,e) \in O_1$, since $\gamma_2$ is obstruction free. Then, we do another (final) case split. Either $(c,f) \in O_1$, which implies that $(a,f) \in O_{\text{recvr}} \subseteq O_2$ and we are done. Or, $(c,f) \notin O_1$, which implies that $(b,g) \in O_1$ (due to cycle $(b,p,g)\gamma_3(f,q,c)$ in $C_1$ and the fact that $(b,g)$ is the only possible obstruction to cover it in $C_1$, since $\gamma_3$ has no obstructions due to the assumptions). Since $(b,g) \in O_1$ and $(a,e) \in O_1$ we have that $(g,e) \in O_{\text{sender}} \subseteq O_2$ in $C_2$, which contradicts our earlier assumption that $\gamma_1$ does not have any obstructions in $C_2$, and we are done. ⊣

## A.2 Lemma 8 (Progress)

By inspection of the reduction rules and our classification from Section 3, we see that all processes must be of the form `select` ... in order for the machine configuration to be globally stuck in a deadlock. Let $C = \langle \text{peer}; O; \Pi \rangle$ be such a configuration.

From the typing of `select`, we know that each process $p_i$ exhibits a set $S_i$ of selected endpoints that satisfy

$$\exists a \in S_i . a \notin O \vee \forall (a,b) \in O . b \in S_i$$

Let $w_i$ be the existential witness $a$ showing that $S_i$ is a valid selection.

We prove the lemma in 2 steps. First we prove that there exist distinct processes $p_i$ and $p_j$ and an enabled channel $(a, b)$, such that $\text{proc}(a) = p_i$ and $\text{proc}(b) = p_j$, and $\text{peer}(a) = b$. Second, we prove that $p_i$ and $p_j$ agree on a message to be exchanged.

We prove the first part by contradiction. Assume there is no enabled channel. Pick an arbitrary process $p_0$ and the witness $w_0$ of its valid selection $S_0$. Let $b_0 = \text{peer}(w_0)$ and $p_1 = \text{proc}(b_0)$. By channel construction (buffer processes), we know that $p_1 \neq p_0$ and by our assumption, we know that $b_0 \notin S_1$, otherwise we are done. Pick $w_1$, the witness of $S_1$ and continue this strategy. Since there are finitely many processes, we must at some point exhibit a cycle $\gamma$ of processes and endpoints, where each segment has the form $(b_i, p_i, w_i)$ and $w_i \in S_i$ but $b_i \notin S_i$. Since the configuration is cycle covered, we know that there exists a process $p_j$ on $\gamma$ and an obstruction $(b_j, w_j)$, where $b_j, w_j$ in $\gamma$. Since, $w_j$ is the witness of the valid selection $S_j$, it must be the case that $b_j \in S_i$ which contradicts our assumption and exhibits an enabled channel $(b_j, \text{peer}(b_j))$, since $\text{peer}(b_j)$ was a witness and thus selected.

Let $(a, b)$ be the enabled channel and $p = \text{proc}(a)$. The second part is trivial, since we know that $a$ has type $\sigma$ and $b$ has type $\sim\!\sigma$ and both are part of their processes selection, it must be the case that one of them has type $?M$. Without loss of generality, assume $\sigma = ?M$. In that case, process $p$ could only pass type checking if it is exhaustive w.r.t. $M$. Since the peer process also type checked, it must offer at least one message of $m$ (rule T-OUT). Thus the reduction can take place. ⊣