Towards a Primitive Higher Order Calculus of Broadcasting Systems

Karol Ostrovský*
Department of Computing Science
Chalmers University of Technology
Gothenburg, Sweden
karol@cs.chalmers.se

K. V. S. Prasad
Department of Computing Science
Chalmers University of Technology
Gothenburg, Sweden
prasad@cs.chalmers.se

Walid Taha†
Department of Computer Science
Rice University
Houston, USA
taha@cs.rice.edu

ABSTRACT

Ethernet-style broadcast is a pervasive style of computer communication. In this style, the medium is a single nameless channel. Previous work on modelling such systems proposed CBS. In this paper, we propose a fundamentally different calculus called HOBS. Compared to CBS, HOBS 1) is higher order rather than first order, 2) supports dynamic subsystem encapsulation rather than static, and 3) does not require an "underlying language" to be Turing-complete. Moving to a higher order calculus is key to increasing the expressivity of the primitive calculus and alleviating the need for an underlying language. The move, however, raises the need for significantly more machinery to establish the basic properties of the new calculus. This paper develops the basic theory for HOBS and presents two example programs that illustrate programming in this language. The key technical underpinning is an adaptation of Howe's method to HOBS to prove that bisimulation is a congruence. From this result, HOBS is shown to embed the lazy λ-calculus.

Categories and Subject Descriptors

F.4 [**Mathematical logic and formal languages**]
; F.3.2 [**Operational semantics**]
; D.1 [**Programming techniques**]
; D.3 [**Programming languages**]

General Terms

Design, Languages, Theory, Verification

Keywords

Broadcasting, Ethernet, Calculi, Programming languages, Semantics, Concurrency

1. INTRODUCTION

Ethernet-style broadcast is a pervasive style of computer communication. The bare medium provided by the Ethernet is a single nameless channel. Typically, more sophisticated programming idioms such as point-to-point communication or named channels are built on top of the Ethernet. But using the Ethernet as is can allow the programmer to make better use of bandwidth, and exploit broadcast as a powerful and natural programming primitive. This paper proposes a primitive higher order calculus of broadcasting systems (HOBS) that models many of the important features of the bare Ethernet, and develops some of its basic operational properties.

1.1 Basic Characteristics of the Ethernet

The basic abstractions of HOBS are inspired by the Ethernet protocol:

- The medium is a single nameless channel.
- Any node can broadcast a message, and it is instantaneously delivered to all the other nodes.
- Messages need not specify either transmitter or receiver.
- The transmitter of a message decides what is transmitted and when.
- Any receiver has to consume whatever is on the net, at any time.
- Only one message can be transmitted at any time.
- Collision detection and resolution are provided by the protocol (for HOBS, the operational semantics), so the abstract view is that if two nodes are trying to transmit simultaneously, one is chosen arbitrarily to do so.

*Partly funded by a grant from Telefonaktiebolaget LM Ericsson, and NSF ITR-0113569.

†This work was started while the author was a post-doctoral fellow at Chalmers University. Funded by a Postdoctoral Fellowship from the Swedish Research Council for Engineering Sciences (TFR). The work was completed while the author was at Yale University.

PPDP'02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.

- All nodes are treated equally; their position on the net does not matter.

HOBS refines and extends a previously proposed system called the calculus of broadcasting systems (CBS) [16]. Although both HOBS and CBS are models of the Ethernet, the two systems take fundamentally different approaches to subsystem encapsulation. To illustrate these differences, we take a closer look at how the Ethernet addresses these issues.

1.2 Modelling Ethernet-Style Encapsulation

Whenever the basic mode of communication is broadcast, encapsulating subsystems is an absolute necessity. In the Ethernet, bridges are used to regulate communication between Ethernet subsystems. A bridge can stop or translate messages crossing it, but local transmission on either side is unaffected by the bridge. Either side of a bridge can be seen as a subsystem of the other.

CBS models bridges by pairs of functions that filter and translate messages going in each direction across the bridge. While this is an intuitively appealing model suitable for many applications, it has limitations:

1. CBS relies on a completely separate "underlying language". In particular, CBS is a *first order* process calculus, meaning that messages are distinct from processes. A separate, computationally rich language is needed to express the function pairs.

2. CBS only provides a *static* model of Ethernet architectures. But, for example real bridges can change their routing behaviour. CBS provides neither for such a change nor for mobile systems that might cross bridges.

3. Any broadcast that has to cross a bridge in CBS does so instantly. This is unrealistic; real bridges usually buffer messages.

HOBS addresses these limitations by:

- Supporting first-class, transmittable processes, and
- Providing novel encapsulation primitives.

Combined, these features of HOBS yield a Turing-complete language sufficient for expressing translators (limitation 1 above), and allow us to model dynamic architectures (limitation 2). The new encapsulation primitives allow us to model the buffering of messages that cannot be consumed immediately (limitation 3).

1.3 Problem

Working with a higher order calculus comes at a cost to developing the theory of HOBS. In particular, whereas the definition of behavioural equivalence in a first order language can require an exact match between the transmitted messages, such a definition is too discriminating when messages involve processes (non-first order values). Thus, behavioural equivalence must only require the transmission of equivalent messages.

Unfortunately, this syntactically small change to the notion of equivalence introduces significant complexity to the proofs of the basic properties of the calculus. In particular, the standard technique for proving that bisimulation is a congruence [11] does not go through. A key difficulty seems to be that the standard technique cannot be used directly to show that the substitution of equivalent processes for variables preserves equivalence (This problem is detailed in Section 4.1.)

1.4 Contributions and Organisation

The main contributions of this paper are the design of HOBS and formal verification of its properties.

After presenting the syntax and semantics of HOBS (Section 2), we give the formal definition of applicative equivalence for HOBS (Section 3).

A key step in the technical development is the use of Howe's method [10] to establish that applicative equivalence for HOBS is a congruence (Section 4). As is typical in the treatment of concurrent calculi, we also introduce a notion of *weak* equivalence. Essentially the same development is used to develop this notion of equivalence (Section 5).

As an application of the these results, in section 6 we use them to show that HOBS embeds the lazy λ-calculus [1], which in turn allows us to define various basic datatypes. This encoding relies on the fact that HOBS is higher order. Possible encodings of CBS and the π-calculus [12] are discussed briefly (More details can be found in the extended version of the paper [14]). Two examples, one dealing with associative broadcast [2] and the other dealing with database consistency [2] are presented in Section 7.

These results lay the groundwork for further applied and theoretical investigation of HOBS. Future work includes developing implementations that compile into Ethernet or Internet protocols, and comparing the expressive power of different primitive calculi of concurrency. We conclude the paper by discussing the related calculi and future works (Section 8).

1.5 Remarks

No knowledge is assumed of CBS or any other process calculus. The formal development in this paper is self-contained.

For readers familiar with CCS [11], CHOCS [18] and CBS: Roughly, HOBS is to CBS what CHOCS is to CCS.

The extended version of the paper (available online [14]) gives details of all definitions, proofs and some additional results.

2. SYNTAX AND SEMANTICS

HOBS has eleven process constructs, formally defined in the next subsection. Their informal meaning is as follows:

- **0** is a process that says nothing and ignores everything it hears.

- x is a variable name. Scoping of variables and substitution is essentially as in the λ-calculus.

- $x?p_1$ receives any message q and becomes $p_1[q/x]$.

- $p_1!p_2$ can say p_1 and become p_2. It ignores everything it hears.

- $\langle x?p_1 + p_2!p_3 \rangle$ says p_2 and becomes p_3 except if it hears something, say q, whereupon it becomes $p_1[q/x]$.

- $p_1|p_2$ is the parallel composition of p_1 and p_2. They can interact with each other and with the environment, $p_1|p_2$ interacts as if it were one process.

Syntax

Variables	$x \in X$	$=$	*A countable set of names*	
Processes	$p, q, r \in P$	$::=$	$g \mid f$	
Ground-terms	$g \in G$	$::=$	$\mathbf{0}$	– Nil
		$\mid$	x	– Variable
		$\mid$	$x?p$	– Input
		$\mid$	$p!p$	– Output
		$\mid$	$\langle x?p + p!p \rangle$	– Guarded Choice
		$\mid$	$p \mid p$	– Parallel Composition
		$\mid$	$p \leftarrow\!\circ\, p$	– In-Filter
		$\mid$	$p \circ\!\!\rightarrow p$	– Out-Filter
		$\mid$	$p \bullet\!\!\leftarrow [p]\, p$	– Out-Filter Internal
Buffers	$f \in F$	$::=$	$p \triangleleft p$	– Feed Buffer
		$\mid$	$p \leftarrow\!\bullet\, p$	– In-Filter Buffer
Messages	$l, m, n \in M$	$::=$	$\tau \mid p$	
Actions	$a \in A$	$::=$	$m! \mid m?$	
Contexts	$c \in C$	$::=$	$[] \mid \mathbf{0} \mid x \mid x?c \mid c!c \mid \langle x?c + c!c \rangle \mid c \mid c \mid c \leftarrow\!\circ\, c \mid c \circ\!\!\rightarrow c \mid c \leftarrow\!\bullet\, c \mid c \bullet\!\!\leftarrow [c]\, c \mid c \triangleleft c$	

Syntax indexed by a set of free-variables L

$$P_L = \{p \mid p \in P \wedge FV(p) \subseteq L\}$$
$$A_L = \{\tau?, \tau!, p?, p! \mid p \in P_L\}$$

Figure 1: Syntax

- $p_1 \leftarrow\!\circ\, p_2$ is the in-filter construct, and behaves as p_1 except that all incoming messages are filtered through p_2. This in-filter is asymmetric; p_2 hears the environment and p_1 speaks to it. A nameless private channel connects p_2 to p_1. This construct represents an in-filter that is waiting for an incoming message. Process p_1 can progress independently.

- $p_1 \leftarrow\!\bullet\, p_2$ represents in-filter in a busy state. Process p_1 is suspended while p_2 processes an input message. Later p_2 sends the processed message to p_1 and p_1 is resumed.

- $p_1 \circ\!\!\rightarrow p_2$ is the out-filter construct, and behaves as p_2 except all outgoing messages from p_2 are filtered through p_1. This out-filter is also asymmetric; p_2 hears the environment and p_1 speaks to it. A nameless private channel connects p_2 to p_1. This filtering construct represents out-filter in passive state waiting for p_2 to produce an output message. Process p_2 can progress independently.

- $p_1 \bullet\!\!\leftarrow [p_2]\, p_3$ represents out-filter in busy state. Process p_3 is suspended while p_1 processes an output message. After processing, p_1 sends the message to the environment, and p_3 is resumed. In case process p_1 fails to process the message before the environment sends some other message, out-filter will "roll-back" to its previous state represented by process p_2.

- $p_1 \triangleleft p_2$ is the feed construct, and consists of p_1 being "fed" p_2 for later consumption as an incoming message. It cannot speak to the environment until p_1 has consumed p_2.

Thus, HOBS is built around the same syntax and semantics as CBS [16], but without a need for an underlying language. Instead, the required expressivity is achieved by adding the ability to communicate processes (higher-orderness), as well as the feed and filtering constructs.

2.1 Formal Syntax and Semantics of HOBS

The syntax of HOBS is presented in Figure 1. Terms are treated as equivalence classes of α-convertible terms. The set P_L is a set of process terms with free variables in the set L, for example $P_\varnothing$ is the set of all closed terms.

REMARK 1 (NOTATION). *The names ranging over each syntactic category are given in Figure 1. Thus process terms are always p, q or r. Natural number subscripts indicate subterms. So p_1 is a subterm of p. Primes mark the result of a transition (or transitions), as in $p \xrightarrow{q?} p'$.*

Figure 2 defines the semantics in terms of a transition relation $\cdot \xrightarrow{\cdot} \cdot \subseteq P_\varnothing \times A_\varnothing \times P_\varnothing$. *Free-variables*, *substitution* and *context filling* are defined as usual (see [14] for details). The semantics is given by labeled transitions, the labels being of the form $p!$ or $p?$ where p is a process. The former are broadcast, speech, or transmission, and the latter are hearing or reception. Transmission can consistently be interpreted as an autonomous action, and reception as controlled by the environment. This is because processes are always ready to hear anything, transmission absorbs reception in parallel composition, and encapsulation (filtering)

Semantics

	Receive	Transmit
Silence	$p \xrightarrow{\tau?} p$	
Nil	$\mathbf{0} \xrightarrow{q?} \mathbf{0}$	
Input	$x?p_1 \xrightarrow{q?} p_1[q/x]$	
Output	$p_1!p_2 \xrightarrow{q?} p_1!p_2$	$p_1!p_2 \xrightarrow{p_1!} p_2$
Choice	$\langle x?p_1 + p_2!p_3\rangle \xrightarrow{q?} p_1[q/x]$	$\langle x?p_1 + p_2!p_3\rangle \xrightarrow{p_2!} p_3$
Compose	$\dfrac{p_1 \xrightarrow{q?} p_1' \quad p_2 \xrightarrow{q?} p_2'}{p_1 \vert p_2 \xrightarrow{q?} p_1' \vert p_2'}$	$\dfrac{p_1 \xrightarrow{m!} p_1' \quad p_2 \xrightarrow{m?} p_2'}{p_1 \vert p_2 \xrightarrow{m!} p_1' \vert p_2'}$ $\dfrac{p_1 \xrightarrow{m?} p_1' \quad p_2 \xrightarrow{m!} p_2'}{p_1 \vert p_2 \xrightarrow{m!} p_1' \vert p_2'}$
Buffers	$f \xrightarrow{q?} f \triangleleft q$	$\dfrac{g_1 \xrightarrow{p_2?} p_1'}{g_1 \triangleleft p_2 \xrightarrow{\tau!} p_1'}$ $\dfrac{f_1 \xrightarrow{\tau!} p_1'}{f_1 \triangleleft p_2 \xrightarrow{\tau!} p_1' \triangleleft p_2}$ $\dfrac{p_2 \xrightarrow{\tau!} p_2'}{p_1 \longleftarrow\!\bullet\, p_2 \xrightarrow{\tau!} p_1 \longleftarrow\!\bullet\, p_2'}$ $\dfrac{p_1 \xrightarrow{m?} p_1' \quad p_2 \xrightarrow{p!} p_2' \quad p \xrightarrow{m!} p'}{p_1 \longleftarrow\!\bullet\, p_2 \xrightarrow{\tau!} p_1' \longleftarrow\!\circ\, p_2'}$
In-Filter	$\dfrac{p_2 \xrightarrow{q?} p_2'}{p_1 \longleftarrow\!\circ\, p_2 \xrightarrow{q?} p_1 \longleftarrow\!\bullet\, p_2'}$	$\dfrac{p_1 \xrightarrow{m!} p_1'}{p_1 \longleftarrow\!\circ\, p_2 \xrightarrow{m!} p_1' \longleftarrow\!\circ\, p_2}$
Out-Filter	$\dfrac{p_2 \xrightarrow{q?} p_2'}{p_1 \propto\!\!- p_2 \xrightarrow{q?} p_1 \propto\!\!- p_2'}$	$\dfrac{p_2 \xrightarrow{\tau!} p_2'}{p_1 \propto\!\!- p_2 \xrightarrow{\tau!} p_1 \propto\!\!- p_2'}$ $\dfrac{p_1 \xrightarrow{p?} p_1' \quad p_2 \xrightarrow{p!} p_2'}{p_1 \propto\!\!- p_2 \xrightarrow{\tau!} p_1' \bullet\!\!\leftarrow\!\![p_1 \propto\!\!- p_2]\, p_2'}$
Out-Filter Internal	$\dfrac{p_2 \xrightarrow{q?} p_2'}{p_1 \bullet\!\!\leftarrow\!\![p_2]\, p_3 \xrightarrow{q?} p_2'}$	$\dfrac{p_1 \xrightarrow{\tau!} p_1'}{p_1 \bullet\!\!\leftarrow\!\![p_2]\, p_3 \xrightarrow{\tau!} p_1' \bullet\!\!\leftarrow\!\![p_2]\, p_3}$ $\dfrac{p_1 \xrightarrow{p!} p_1' \quad p \xrightarrow{m!} p'}{p_1 \bullet\!\!\leftarrow\!\![p_2]\, p_3 \xrightarrow{m!} p_1' \propto\!\!- p_3}$

Figure 2: Semantics

can stop reception but only hide transmission. For more discussion, see [16].

The filter constructs required careful design. One difficulty is that filters should be able to hide messages. Technically, this means that filters should be able to produce silent messages as a result. But "silence" message τ is not a process construct. Therefore, each filter produces a "messenger" process as a result, and this messenger process then sends the actual result of filtration and is then discarded. This way the messenger process can produce any message (that is any process or silent τ).

The transition relation $\cdot \xrightarrow{\cdot} \cdot \subseteq P_\varnothing \times A_\varnothing \times P_\varnothing$ fails to be a function (in the first two arguments) because the composition rule allows arbitrary exchange of messages between sub-processes. The choice construct does not introduce nondeterminism by itself, since any broadcast collision can be resolved by allowing the left sub-process of a parallel composition to broadcast.

However, the calculus is deterministic on input, and is input enabled. That is,

$$\forall p, m.\, \exists! p'.\, p \xrightarrow{m?} p' \tag{1}$$

This is easily shown by induction on the derivation $p \xrightarrow{m?} p'$.

For further discussion of these and other design decisions we refer the reader to the extended version of the paper [14].

3. APPLICATIVE BISIMULATION

There are no surprises in the notions of simulation and bisimulation for HOBS, and the development uses the same techniques as for channel-based calculi [11].

Because the transition relation carries processes in labels, and because notions of higher order simulation and bisimulation have to account for the structure of these processes, we use the following notion of *message extension* for convenience.

Definition 1 (Message Extension). *Let $\mathcal{R} \subseteq P \times P$ be a relation on process terms. Its* message extension $\mathcal{R}_\tau \subseteq M \times M$ *is defined by the following rules*

$$\frac{}{\tau\, \mathcal{R}_\tau\, \tau}\ \textsc{TauExt} \qquad \frac{p\, \mathcal{R}\, q}{p\, \mathcal{R}_\tau\, q}\ \textsc{MsgExt}$$

Thomsen's notion of applicative higher order simulation [18] is suitable for strong simulation in HOBS, because we have to take the non-grounded (higher order) nature of the messages into account.

Definition 2 (Applicative Simulation). *A relation $\mathcal{R} \subseteq P_\varnothing \times P_\varnothing$ on closed process terms is a* (strong, higher order) applicative simulation*, written $S(\mathcal{R})$, when*

$\forall (p, q) \in \mathcal{R}, \forall m, p'.$

1. $p \xrightarrow{m?} p' \ \Rightarrow\ (\exists q'.\, q \xrightarrow{m?} q' \ \wedge\ p'\, \mathcal{R}\, q')$
2. $p \xrightarrow{m!} p' \ \Rightarrow\ (\exists q', n.\, q \xrightarrow{n!} q' \ \wedge\ p'\, \mathcal{R}\, q' \ \wedge\ m\, \mathcal{R}_\tau\, n)$

We use a standard notion of bisimulation:

Definition 3 (Applicative Bisimulation). *A relation $\mathcal{R} \subseteq P_\varnothing \times P_\varnothing$ on closed process terms is an* applicative bisimulation*, written $B(\mathcal{R})$, when both $S(\mathcal{R})$ and $S(\mathcal{R}^{-1})$ hold. That is,*

$$\frac{S(\mathcal{R}) \quad S(\mathcal{R}^{-1})}{B(\mathcal{R})}$$

Using standard techniques, we can show that the identity relation on closed processes is a simulation and a bisimulation. The property of being a simulation, a bisimulation respectively, is preserved by relational composition and union. Also, the bisimulation property is preserved by converse.

Two closed processes p and q are equivalent, written $p \sim q$, if there exist a bisimulation relation $\mathcal{R}$ such that $(p, q) \in \mathcal{R}$. In other words, applicative equivalence is a union of all bisimulation relations:

Definition 4 (Applicative Equivalence). *The* applicative equivalence *is a relation $\sim \subseteq P_\varnothing \times P_\varnothing$ defined as a union of all bisimulation relations. That is,*

$$\sim\ = \bigcup_{\mathcal{R} \subseteq P_\varnothing \times P_\varnothing} \{\mathcal{R} \mid B(\mathcal{R})\}$$

Proposition 1.

1. *$B(\sim)$, that is, $\sim$ is a bisimulation.*
2. *$\sim$ is an equivalence, that is, $\sim$ is reflexive, symmetric and transitive.*

The calculus enjoys the following basic properties:

Proposition 2. *Let $p, p_i \in P_\varnothing$. Then, we have*

- *Input*
  1. $x?\mathbf{0} \sim \mathbf{0}$
- *Parallel Composition*
  1. $p|\mathbf{0} \sim p$
  2. $p_1|p_2 \sim p_2|p_1$
  3. $p_1|(p_2|p_3) \sim (p_1|p_2)|p_3$
- *Filters*
  1. $\mathbf{0} \propto\!\!- p \sim \mathbf{0}$
  2. $(p_1 \propto\!\!- p_2) \leftarrow\!\!\circ\, p_3 \sim p_1 \propto\!\!- (p_2 \leftarrow\!\!\circ\, p_3)$
  3. $x?p \leftarrow\!\!\circ\, \mathbf{0} \sim \mathbf{0}$
  4. $\langle x?p_1 + p_2!p_3 \rangle \leftarrow\!\!\circ\, \mathbf{0} \sim p_2!p_3 \leftarrow\!\!\circ\, \mathbf{0}$
- *Choice*
  1. $\langle x?p_1!p_2 + p_1!p_2 \rangle \sim p_1!p_2,\ x \notin FV(p_1!p_2)$
  2. $\langle x?p_1 + p_2!p_3 \rangle | x?p_4 \sim \langle x?(p_1|p_4) + p_2!(p_3|p_4[p_2/x]) \rangle$

4. EQUIVALENCE AS A CONGRUENCE

In this section we use Howe's method [10] to show that the applicative equivalence relation $\sim$ is a congruence. To motivate the need for Howe's proof method, we start by showing the difficulties with the standard proof technique [11]. Then, we present an adaptation of Howe's basic development for HOBS. We conclude the section by applying this adaptation to applicative equivalence.

An equivalence is a congruence when two equivalent terms are not distinguishable in any context:

Definition 5 (Congruence). *Let $\mathcal{R} \subseteq P \times P$ be an equivalence relation on process terms. $\mathcal{R}$ is a* congruence *when*

$$\forall p, q \in P, c \in C.\, p\, \mathcal{R}\, q \ \Rightarrow\ c[p]\, \mathcal{R}\, c[q]$$

$$\frac{}{\mathbf{0}\,\widehat{\mathcal{R}}\,\mathbf{0}}\ \textsc{Comp Nil} \qquad \frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2}{p_1!p_2\,\widehat{\mathcal{R}}\,q_1!q_2}\ \textsc{Comp Out} \qquad \frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2}{p_1 \leftarrow\!\circ\, p_2\,\widehat{\mathcal{R}}\,q_1 \leftarrow\!\circ\, q_2}\ \textsc{Comp InFilter}$$

$$\frac{}{x\,\widehat{\mathcal{R}}\,x}\ \textsc{Comp Var} \qquad \frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2}{p_1|p_2\,\widehat{\mathcal{R}}\,q_1|q_2}\ \textsc{Comp Comp} \qquad \frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2}{p_1 \leftarrow\!\bullet\, p_2\,\widehat{\mathcal{R}}\,q_1 \leftarrow\!\bullet\, q_2}\ \textsc{Comp InFilterB}$$

$$\frac{p_1\,\mathcal{R}\,q_1}{x?p_1\,\widehat{\mathcal{R}}\,x?q_1}\ \textsc{Comp In} \qquad \frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2}{p_1 \triangleleft p_2\,\widehat{\mathcal{R}}\,q_1 \triangleleft q_2}\ \textsc{Comp Feed} \qquad \frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2}{p_1 \propto\!\!-\, p_2\,\widehat{\mathcal{R}}\,q_1 \propto\!\!-\, q_2}\ \textsc{Comp OutFilter}$$

$$\frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2 \quad p_3\,\mathcal{R}\,q_3}{\langle x?p_1 + p_2!p_3\rangle\,\widehat{\mathcal{R}}\,\langle x?q_1 + q_2!q_3\rangle}\ \textsc{Comp Choice} \qquad \frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2 \quad p_3\,\mathcal{R}\,q_3}{p_1 \bullet\!\!-\![p_2]\,p_3\,\widehat{\mathcal{R}}\,q_1 \bullet\!\!-\![q_2]\,q_3}\ \textsc{Comp OutFilterI}$$

Figure 3: Compatible Refinement

4.1 Difficulty with the Standard Proof Method

The notion of compatible refinement $\widehat{\mathcal{R}}$ allows us to concisely express case analysis on the outermost syntactic constructor:

Definition 6 (Compatible Refinement). *Let $\mathcal{R} \subseteq P \times P$ be a relation on process terms. Its* compatible refinement $\widehat{\mathcal{R}}$ *is defined by the rules in Figure 3.*

The standard congruence proof method is to show, by induction on the syntax, that equivalence $\sim$ contains its compatible refinement $\widehat{\sim}$. The standard method for proving congruence centers around proving the following lemma:

Lemma 1. *Let $\mathcal{R} \subseteq P \times P$ be an equivalence relation on process terms. Then $\mathcal{R}$ is a congruence iff $\widehat{\mathcal{R}} \subseteq \mathcal{R}$.*

The standard proof (to show $\widehat{\sim} \subseteq\!\sim$) still proceeds by case analysis. Several cases are simple (nil $\mathbf{0}$, variable and output $x?\cdot$). The case of feed $\cdot \triangleleft \cdot$ is slightly more complicated. All the other cases are problematic (especially composition $\cdot|\cdot$), since they require substitutivity of equivalence $\sim$ where substitutivity is defined (in a usual way) as:

Definition 7 (Substitutivity). *Let $\mathcal{R} \subseteq P \times P$ be a relation on process terms. $\mathcal{R}$ is called* substitutive *when the following rule holds*

$$\frac{p_1\,\mathcal{R}\,q_1 \quad p_2\,\mathcal{R}\,q_2}{p_1[p_2/x]\,\mathcal{R}\,q_1[q_2/x]}\ \textsc{Rel Subst}$$

In HOBS, the standard inductive proof of substitutivity of equivalence $\sim$ requires equivalence to be a congruence. And, we are stuck. Attempt to prove substitutivity and $\widehat{\sim} \subseteq\!\sim$ simultaneously does not work either, since a term's size can increase and this makes use of induction on syntax impossible. Similar problems seem to be common for higher order calculi (see for example [18, 7, 1]).

4.2 Howe's Basic Development

Howe [10] proposed a general method for proving that certain equivalences based on bisimulation are congruences. Following similar adaptations of Howe's method [9, 7] we present the adaptation to HOBS along with the necessary technical lemmas. We use the standard definition of the restriction $\mathcal{R}_\varnothing$ of a relation $\mathcal{R}$ to closed processes (cf. [14]). Extension of a relation to open terms is also the standard one:

Definition 8 (Open Extension). *Let $\mathcal{R} \subseteq P \times P$ be a relation on process terms. Its* open extension *is defined by the following rule*

$$\frac{\forall\sigma.\,(p)\sigma, (q)\sigma \in P_\varnothing \ \Rightarrow\ (p)\sigma\,\mathcal{R}\,(q)\sigma}{p\,\mathcal{R}^\circ\,q}$$

The key part of Howe's method is the definition of the candidate relation $\mathcal{R}^\bullet$:

Definition 9 (Candidate Relation). *Let $\mathcal{R} \subseteq P \times P$ be a relation on process terms. Then a* candidate relation *is defined as the least relation that satisfies the rule*

$$\frac{p\,\widehat{\mathcal{R}^\bullet}\,r \quad r\,\mathcal{R}^\circ\,q}{p\,\mathcal{R}^\bullet\,q}\ \textsc{Cand Def}$$

The definition of the candidate relation $\mathcal{R}^\bullet$ facilitates simultaneous inductive proof on syntax and on reductions. Note that the definition of the compatible refinement $\widehat{\mathcal{R}}$ involves only case analysis over syntax. And, inlining the compatible refinement $\widehat{\mathcal{R}}$ in the definition of the candidate relation would reveal inductive use of the candidate relation $\mathcal{R}^\bullet$.

The relevant properties of a candidate relation $\mathcal{R}^\bullet$ are summed up below:

Lemma 2. *Let $\mathcal{R} \subseteq P_\varnothing \times P_\varnothing$ be a preorder (reflexive, transitive relation) on closed process terms. Then the following rules are valid*

$$\frac{}{p\,\mathcal{R}^\bullet\,p}\ \textsc{Cand Ref} \qquad \frac{p\,\mathcal{R}^\circ\,q}{p\,\mathcal{R}^\bullet\,q}\ \textsc{Cand Sim}$$

$$\frac{p\,\widehat{\mathcal{R}^\bullet}\,q}{p\,\mathcal{R}^\bullet\,q}\ \textsc{Cand Cong} \qquad \frac{p\,\mathcal{R}^\bullet\,r \quad r\,\mathcal{R}^\circ\,q}{p\,\mathcal{R}^\bullet\,q}\ \textsc{Cand Right}$$

$$\frac{p_1\,\mathcal{R}^\bullet\,q_1 \quad p_2\,\mathcal{R}^\bullet\,q_2}{p_1[p_2/x]\,\mathcal{R}^\bullet\,q_1[q_2/x]}\ \textsc{Cand Subst}$$

Corollary 1. *We have $\mathcal{R}^\bullet \subseteq {\mathcal{R}^\bullet}_\varnothing{}^\circ$ as an immediate consequence of rule* Cand Subst *and rule* Cand Ref

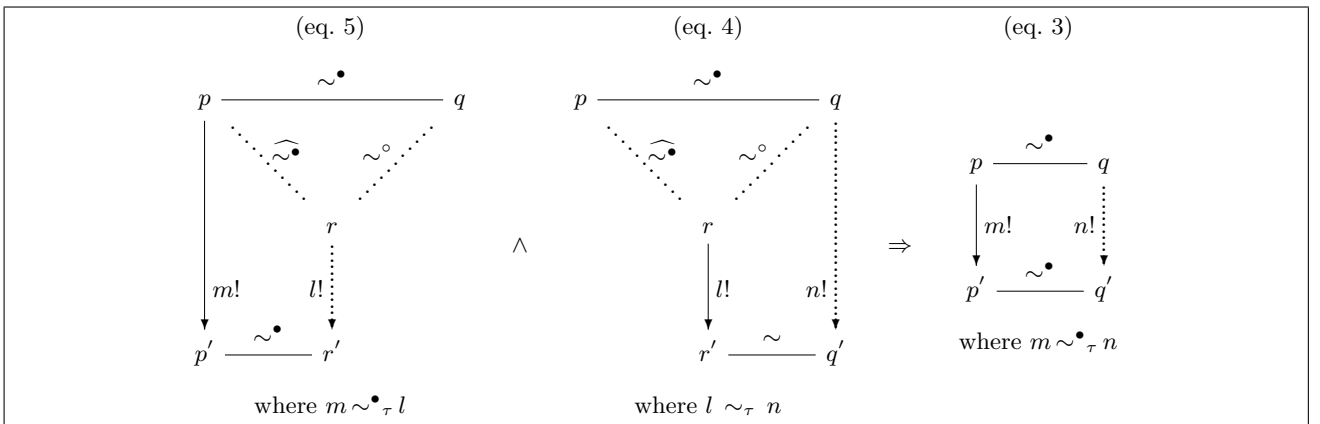

Figure 4: Transmit Lemma

LEMMA 3. *Let $\mathcal{R} \subseteq P \times P$ be an equivalence relation. Then $\mathcal{R}^{\bullet *}$ is symmetric.*

The next lemma says that if two candidate-related processes $p \mathcal{R}^\bullet q$ are closed terms then there is a derivation which involves only closed terms in the last derivation step.

LEMMA 4 (CLOSED MIDDLE).

$$\forall p, q \in P_\varnothing \,.\, p \mathcal{R}^\bullet q \;\Rightarrow\; \exists r \in P_\varnothing \,.\, p \,\widehat{\mathcal{R}^\bullet}\, r \,\mathcal{R}^\circ q$$

4.3 Congruence of the Equivalence Relation

Now our goal of is twofold: first, to show that the candidate relation $\sim^\bullet$ coincides with the open extension $\sim^\circ$, that is $\sim^\bullet = \sim^\circ$; and second, to use this fact to complete the congruence proof.

First, we will fix the underlying relation $\mathcal{R}$ to be $\sim$. We already know $\sim^\circ \subseteq \sim^\bullet$ from Lemma 2 (rule CAND SIM). To show the converse we begin by proving that the closed restriction of the candidate relation $\sim^\bullet_\varnothing$ is a simulation. This requires showing that the two simulation conditions of Definition 2 hold.

We split the proof into two lemmas: Lemma 5 (Receive) and Lemma 6 (Transmit), which prove the respective conditions. Similarly to the standard proof, the parallel composition case is the most difficult and actually requires a stronger receive condition to hold. The first lemma (Receive) below proves a restriction of the first condition. Then the second lemma (Transmit) below proves the second condition and makes use of the Receive Lemma.

LEMMA 5 (RECEIVE). *Let $p, q \in P_\varnothing$ be two closed processes and $p \sim^\bullet q$. Then $\forall m, n, p'$.*

$$p \xrightarrow{m?} p' \;\wedge\; m \sim^\bullet_\tau n \;\Rightarrow\; (\exists q' \,.\, q \xrightarrow{n?} q' \;\wedge\; p' \sim^\bullet q') \qquad (2)$$

PROOF. Relatively straightforward proof by induction on the height of inference of transition $p \xrightarrow{m?} p'$. The only interesting case is the case of rule $x?p_1 \xrightarrow{q?} p_1[q/x]$, which makes use of the substitutivity of the candidate relation $\sim^\bullet$. □

LEMMA 6 (TRANSMIT). *Let $p, q \in P_\varnothing$ be two closed processes and $p \sim^\bullet q$. Then $\forall m, p'$.*

$$p \xrightarrow{m!} p' \;\Rightarrow\; (\exists n, q' \,.\, q \xrightarrow{n!} q' \;\wedge\; p' \sim^\bullet q' \;\wedge\; m \sim^\bullet_\tau n) \qquad (3)$$

PROOF. First note that from Lemma 4 (Closed Middle) we have that

$$\exists r \in P_\varnothing \,.\, p \,\widehat{\sim^\bullet}\, r \;\sim^\circ\; q$$

Also, from the definition of equivalence $\sim$ and the fact that r and q are closed processes we know that

$$\forall l, r' \,.\, r \xrightarrow{l!} r' \;\Rightarrow\; (\exists n, q' \,.\, q \xrightarrow{n!} q' \wedge r' \;\sim\; q' \wedge l \;\sim_\tau n) \qquad (4)$$

It remains only to prove that $\forall m, p'$.

$$p \xrightarrow{m!} p' \;\Rightarrow\; (\exists l, r' \,.\, r \xrightarrow{l!} r' \;\wedge\; p' \;\sim\; r' \;\wedge\; m \;\sim_\tau\; l) \qquad (5)$$

Joining statements (5) and (4), and using Lemma 2 (rule CAND RIGHT) to infer $p' \sim^\bullet q'$ and $m \sim^\bullet_\tau n$ gives us the result. Figure 4 shows the idea pictorially (normal lines, respectively dotted lines, are universally, respectively existentially quantified).

To prove the statement (5) we proceed by induction on the height of inference of transition $p \xrightarrow{m!} p'$. We only describe the most interesting case – parallel composition.

Compose There are two parallel composition rules. Since the rules are symmetric we only show the proof for one of them. We know that $p \equiv p_1 | p_2$, and we have four related sub-processes: $p_1 \sim^\bullet r_1$ and $p_2 \sim^\bullet r_2$. Now suppose that m is a process, and that p made the following transition:

$$\frac{p_1 \xrightarrow{m!} p_1' \quad p_2 \xrightarrow{m?} p_2'}{p_1 | p_2 \xrightarrow{m!} p_1' | p_2'}$$

Since the candidate relation $\sim^\bullet$ contains its compatible refinement, it is enough to show that each sub-process of r can mimic the corresponding sub-process of p. For r_1, using the induction hypothesis we get that $r_1 \xrightarrow{l!} r_1'$ and $p_1' \sim^\bullet r_1'$, $m \sim^\bullet_\tau l$. For r_2, if we would use only the simulation condition, we would get $r_2 \xrightarrow{m?} r_2'$, and this would not allow us to show that r has a corresponding transition, since the inference of a transition requires both labels to be the same. At this point, we can use the stronger receive condition of Lemma 5 and we get precisely what we need, that is: $r_2 \xrightarrow{l?} r_2'$ and $p_2' \sim^\bullet r_2'$. The case when m is τ is

similar, but simpler since it does not require the use of Lemma 5 (Receive). □

With the two lemmas above we have established that the restriction of the candidate relation $\sim^\bullet_\varnothing$ is a simulation. Also, using Lemma 3 we get that $\sim^\bullet_{\varnothing^*}$ is symmetric, which means that it is a bisimulation. To conclude this section, we are now ready to state and prove the main proposition.

PROPOSITION 3. *$\sim^\circ$ is a congruence.*

PROOF. First we show that $\sim^\circ = \sim^\bullet$. From Lemma 2 (rule CAND SIM) we know $\sim^\circ \subseteq \sim^\bullet$. From the two lemmas above and Lemma 3 we know that $\sim^\bullet_{\varnothing^*}$ is a bisimulation. This implies $\sim^\bullet_\varnothing \subseteq \sim^\bullet_{\varnothing^*} \subseteq \sim$. Since open extension is monotone we have $\sim^\bullet_{\varnothing^\circ} \subseteq \sim^\circ$. By Corollary 1 we get $\sim^\bullet \subseteq \sim^\bullet_{\varnothing^\circ}$, and so $\sim^\bullet \subseteq \sim^\circ$.

As $\sim^\circ$ is an equivalence, and it is equal to the candidate relation, it contains its compatible refinement ($\widehat{\sim^\circ} \subseteq \sim^\circ$, Lemma 2 rule CAND CONG). By Lemma 1 this implies that $\sim^\circ$ is a congruence. □

5. WEAK BISIMULATION

For many purposes, strong applicative equivalence is too fine as it is sensitive to the number of silent transitions performed by process terms. Silent transitions represent local computation, and in many cases it is desirable to analyse only the communication behaviour of process terms, ignoring intermediate local computations. For example, the strong applicative equivalence distinguishes the following two terms that have the same communication behaviour:

$$(x?x) \triangleleft (x?x) \not\sim x?x$$

Equipped with the weak transition relation $\cdot \stackrel{\cdot}{\Longrightarrow} \cdot$ (Definition 10) we define the *weak simulation* and *weak bisimulation* in the standard way[11]. Weak equivalence $\approx$ is also defined in the standard way as a union of all weak bisimulation relations. Just as for the strong equivalence, we prove that $\approx$ is a weak bisimulation and that it is an equivalence (Proposition 4). Moreover, the technique used in proving that strong equivalence is a congruence works also for the weak equivalence (Proposition 5).

DEFINITION 10 (WEAK TRANSITION). *Let $\stackrel{\epsilon}{\Longrightarrow}$ be the reflexive transitive closure of $\stackrel{\tau!}{\longrightarrow}$. Then the weak transition is defined as*

$$\stackrel{a}{\Longrightarrow} \;=\; \begin{cases} \stackrel{\epsilon}{\Longrightarrow} & \text{if } a \equiv \tau! \\ \stackrel{\epsilon}{\Longrightarrow}\stackrel{a}{\longrightarrow} & \text{otherwise} \end{cases}$$

PROPOSITION 4.

1. *$\approx$ is a weak bisimulation*
2. *$\approx$ is an equivalence.*

PROPOSITION 5. *$\approx$ is a congruence.*

PROOF. The proof follows that of Proposition 3. An interesting difference is when the induction hypothesis is used (for example in the case of parallel composition). When using the induction hypothesis we get that the appropriate subterms can perform weak transition $\stackrel{a}{\Longrightarrow}$. Then we have to interleave the $\tau!$ transitions before possibly performing the action a. This interleaving is possible since every process can receive τ without any change to the process itself (see rule Silence in Figure 2). □

Syntax

$$\textit{Expressions}\;\; e \in E \;::=\; x \mid \lambda x.e \mid e_1\; e_2$$

Semantics

$$\frac{}{(\lambda x.e_1)\; e_2 \longrightarrow e_1[x := e_2]}\,\beta \qquad \frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2}\,\mu$$

Embedding

$$\begin{array}{rcl} [\![x]\!]_\lambda & \equiv & x \\ [\![\lambda x.e]\!]_\lambda & \equiv & x?[\![e]\!]_\lambda \\ [\![e_1\; e_2]\!]_\lambda & \equiv & [\![e_1]\!]_\lambda \triangleleft [\![e_2]\!]_\lambda \end{array}$$

Figure 5: Syntax, semantics and embedding of lazy λ-calculus.

6. EMBEDDINGS AND ENCODINGS

While a driving criterion in the design of HOBS is simplicity and resemblance to the Ethernet, a long term technical goal is to use this simple and uniform calculus to interpret other more sophisticated proposals for broadcasting calculi, such as the $b\pi$-calculus [5]. Having HOBS interpretations of hand-shake calculi such as π-calculus, π-calculus with groups [3], and CCS could also provide a means for studying the expressive power of these calculi.

In this section, we present the embedding of the lazy λ-calculus and its consequences, and briefly discuss possible encodings of CBS and the π-calculus.

6.1 Embedding of the λ-calculus

The syntax and semantics of the lazy λ-calculus, along with a direct translation into HOBS, are presented in Figure 5. The function $\cdot \longrightarrow \cdot \subseteq E \times E$ is the small-step semantics for the language.

Using weak equivalence $\approx$ we can prove that the analog of β-equivalence in HOBS holds. A simple proof of this proposition using weak bisimulation up to $\approx$ technique can be found in the extended version of the paper [14].

PROPOSITION 6. $[\![(\lambda x.e_1)e_2]\!]_\lambda \approx [\![e_1[x := e_2]]\!]_\lambda$

PROPOSITION 7. *(Soundness) Let $\simeq_\lambda$ be the standard λ-calculus notion of observation equivalence. Then*

$$[\![e_1]\!]_\lambda \approx [\![e_2]\!]_\lambda \;\Rightarrow\; e_1 \simeq_\lambda e_2$$

Being able to embed the λ-calculus justifies not having explicit construct in HOBS for either recursion or datatypes. Just as in the λ-calculus there is a recursive Y combinator, HOBS can express recursion by the derived construct rec defined on the left below. This recursive construct has the expected behaviour as stated on the right below.

$$\begin{array}{rcl} \mathsf{W}_x(p) & \equiv & y?(x?p \triangleleft (y \triangleleft y)) \\ \mathsf{rec}\; x.p & \equiv & \mathsf{W}_x(p) \triangleleft \mathsf{W}_x(p) \end{array} \qquad \mathsf{rec}\; x.p \approx p[\mathsf{rec}\; x.p/x]$$

From the point of view of the λ-calculus, it is significant that this embedding is possible. In particular, broadcasting can be viewed as an impurity, or more technically as a computational effect [13]. Yet the presence of this effect does not invalidate the β-rule. We view this as a positive indicator about the design of HOBS.

6.2 Encodings of Concurrent Calculi

Having CBS as a precursor in the development of HOBS, it is natural to ask whether HOBS can interpret CBS. First, CBS assumes an underlying language with data types, which HOBS provides in the form of an embedded lazy λ-calculus, using the standard Church data encodings. Second, the CBS translator is the only construct that is not present in HOBS. To interpret it, we use a special parametrised queue construct. The queue construct together with the parameter (the translating function) is used as a one-way translator. Linking two of these to a process (via filter constructs) gives a CBS-style translator with decoupled translation.

Using Church numerals to encode channel names we can easily interpret the π-calculus without the new operator. Devising a sound encoding of the full π-calculus is more challenging, since there are several technical difficulties, for example explicit α-conversion, that have to be solved.

For further discussion and proposed solutions see the extended version of the paper [14]. The soundness proof for these encodings is ongoing work.

7. EXAMPLES

HOBS is equiped with relatively high-level abstractions for broadcasting communication. Because HOBS includes the lazy λ-calculus, we can extend it to a full functional language. This gives us tool for experimenting with broadcasting algorithms. As the theory develops, we hope that HOBS will also be a tool for formal reasoning about the algorithms.

In this section we present an implementation of a coordination formalism called associative broadcast [2], together with an implementation of a database consistency algorithm expressed in this formalism. Compared to previous implementations of this algorithm (see for example [6]), the HOBS implementation is generic, retains the full expressive power of associative broadcast, and allows straightforward representation of associative broadcast algorithms.

Our HOBS interpreter is implemented in OCaml. In addition, we also use an OCaml-like syntax for HOBS functional terms and datatypes, and use just juxtaposition instead of the feed construct symbol. For the purposes of this paper, the reader can treat the typing information as simply comments.

7.1 Associative Broadcast

Bayerdorffer [2] describes a coordination formalism called associative broadcast. This formalism uses broadcasting as a communication primitive. In this formalism, each object that participates in the communication has a profile. A profile can be seen as a set of attribute-value pairs. A special subset of these pairs contains values that are functions which modify the profile. This subset is divided into functions that are used in broadcasting, and those that are used locally. Since associative broadcast is a coordination formalism, all objects are locally connected to some external system, and they can invoke operations on that system. Conversely, the external system can invoke the local functions of the object.

Communication between objects proceeds as follows: each broadcast message contains a specification of the set of recipients, and an operation that recipients should execute. The specification is a first-order formula which examines a profile. An operation is a call to one of the functions that modify the profile. When a message is broadcasted by an object, it is received by all objects including the sender. Each object then evaluates the formula on its own profile to determine whether it should execute the operation. The operation is executed only if the profile satisfied the formula.

The generic part of associative broadcast is represented by so called RTS (run time system), which takes care of the communication protocol. The following are the basic definitions needed for an implementation of RTS in HOBS:

```
type 'a selector = 'a -> bool;;

type 'a operation = 'a -> ('a -> 'a) -> 'a;;

type 'a message =
  Message of 'a selector * 'a operation
| Internal of 'a operation;;

type 'a tag = In of 'a
            | Out of 'a

let rec p1 = x?match x with
                Out(p) -> (p!0)!p1
              | In(p)  -> (0 0)!p1
and     p2 = x?((Out(x))!0)!p2
and     p3 = x?match x with
                Out(p) -> (0 0)!p3
              | In(p)  -> (p!0)!p3
and     p4 = x?((In(x))!0)!p4;;

let rec m = x?match x with
               Out(p) -> (In(p))!m
             | In(p)  -> m;;

let rec obj profile =
  x?match x with
      Message(sel,op) -> if (sel profile) then
                           op profile obj
                         else
                           obj profile
    | Internal(op)    -> op profile obj
```

Recipient specification formulas have type `'a selector`, operations have type `'a operation` which should be viewed as a function which takes a profile (type `'a`) and a continuation and returns a profile. Messages that an object can receive have type `'a message`. The key component of the RTS is the representation of an object. In HOBS an object can be implemented as:

$$\texttt{p1} \propto\!\!- (\texttt{p2} \propto\!\!- (\texttt{obj} \triangleleft \texttt{init\_profile}) \longleftarrow\!\!\circ \texttt{p3} \mid \texttt{m}) \longleftarrow\!\!\circ \texttt{p4}$$

where process `obj` executes the main object loop that runs the protocol. The filtering processes `p1, p2, p3, p4` and the "mirroring" process `m` take care of the broadcast message loopback, that is routing each message to all the other objects and the object itself. Loopback routing uses simple intuitive tagging of messages.

Having implemented the associative broadcast RTS, we can implement any associative broadcast algorithms by creating a profile required by the algorithm. To run the algorithm we only need to create a parallel composition with the appropriate number of objects with their profiles.

7.2 Database Consistency

In a distributed database there may be several copies of the same data entity, and ensuring the consistency of these

various copies becomes a concern. Inconsistency can arise if a transaction occurs while a connection between two nodes is broken. If we know that concurrent modifications are rare or that downtime is short, we can employ the following optimistic protocol [2]: When a failure occurs, network nodes are broken into partitions. While the network is down all nodes log their transactions. After discovering a network reconnection we construct a global precedence graph from the log of all transactions. If this graph does not contain cycles, then the database is consistent. A transaction t_1 precedes transaction t_2 if[1]:

- both t_1 and t_2 happened in one partition, and t_2 read data previously written by t_1
- both t_1 and t_2 happened in one partition, and t_1 read data later written by t_2
- t_1 and t_2 happened on different partitions, and t_1 read data written by t_2

The algorithm represents each item (table, row, etc., depending on the locking scheme) by an RTS coordination object. This object will keep the log of all local transactions, and so each object will hold a part of the precedence graph. To connect these parts into the full graph each object will broadcast a token to other objects along the precedence graph edges. When an object receives a token it will propagate the token along the precedence edges the object already maintains. If a token returns to its originating object then we have found an inconsistency. In parallel with token propagation each object also sends a merge message to actually merge (or update) values of the item in different partitions. And, if an object that modified its item receives merge message, it also declares inconsistency.

In what follows we present the key definitions in our implementation. The code for the full implementation can be found in Figure 6 on the last page. The profile used by each object is defined as follows:

```
type profile =
  {oid:int;
   mutable item_name: string;
   mutable item_value: int;
   mutable reads: transaction list;
   mutable written: bool;
   mutable partition: int;
   mutable merged: bool;
   mutable mcount: int;
   mutable tw: transaction;

   propagate: int -> int -> profile operation;
   merge: int -> bool -> profile operation;
   upon_partitioning: unit -> profile operation;
   upon_commiting_transaction:
      int -> profile operation;
   upon_detecting_reconnection:
      profile operation};;
```

Each object keeps a unique identifier `oid`, the name of the item it monitors, the value of that item, a set of transactions `reads`, a flag to signal if the item was written, and a partition number. It also keeps a set of local attributes: `merged` flag to check whether the item values are already merged between partitions, number of merge messages received `mcount`, and the last logged write transaction `tw`.

Each object has two broadcasting operations: `propagate` to propagate a token along the precedence edges it contains, and `merge` to possibly update the item to a new value. Each object has three local operations: `upon_partitioning` to record the local partition number, `upon_commiting_transaction` to record committed transactions, and `upon_detecting_reconnection` that starts the graph construction.

The functions that the external system is assumed to provide are: `local_partition_id` to get a partition identifier; `log` to log transactions; `modifies` to check whether a transaction modifies an item; `precedes` to check (using the local log) whether a transaction precedes other transaction; `declare_inconsistency` to declare database inconsistency; `delay_transactions` to pause the running transactions; `count_objects` to get the number of all objects; `count_local_objects` to get the number of objects in a partition; and `write_locked` to check whether an item is locked for writing.

[1]When an item is written it is also considered to be read.

8. RELATED WORK

In this section we review works related to our basic design choices and the central proof technique used in the paper.

8.1 Alternative Approaches to Modelling Dynamic Connectivity

One approach to modelling dynamic broadcast architectures is to support special messages that can change bridge behaviour. This corresponds to the transmission of channel names in the π-calculus [12]. Another approach is to allow processes be transmitted, so that copies can be run elsewhere in the system. This makes the calculus *higher order*, like CHOCS [18]. This is the approach taken in this paper. A preliminary variant of HOBS sketched in [15] retains the underlying language of messages and functions. The resulting calculus seems to be unnecessarily complex, and having the underlying language seems redundant.

In HOBS, processes are the only entities, and they are used to characterise bridges. Since processes can be broadcasted, it will be interesting to see if HOBS can model some features of the π-calculus. Because arrival at a bridge and delivery across it happen in sequence, HOBS avoids CBS's insistence that these actions be simultaneous. This comes at the cost of having less powerful synchronisation between subsystems.

8.2 Related Calculi

The $b\pi$-calculus [5] can be seen as a version of the π-calculus with broadcast communication instead of point-to-point. In particular, the $b\pi$-calculus borrows the whole channel name machinery of the π-calculus, including the operator for creation of new names. Thus the $b\pi$-calculus does not model the Ethernet directly, and is not obviously a mobile version of CBS. Reusing ideas from the sketched π-calculus encoding can yield a simple $b\pi$-calculus encoding. Using filterss to model scopes of new names seems promising. Moreover, such an encoding might be compositional. We expect that with an appropriate type system we can achieve a fully abstract encoding of the $b\pi$-calculus. The type system would be a mixture of Hindley/Milner and polymorphic π-calculus [19] type systems.

The Ambient calculus [4] is a calculus of mobile processes with computation based on a notion of movement. It is equipped with intra-ambient asynchronous communication similar to the asynchronous π-calculus. Since the choice of communication mechanism is independent from the mobility primitives, it may be interesting to study a broadcasting version of Ambient calculus. Also, broadcasting Ambient calculus might have simple encoding in HOBS.

Both HOBS and the join calculus [8] can be viewed as extensions of the λ-calculus. HOBS adds parallel composition and broadcast communication on top of the bare λ-calculus. The join calculus adds parallel composition and a parallel pattern on top of the λ-calculus with explicit let. The relationship between these two calculi remains to be studied.

The feed operator $\triangleleft$ is foreshadowed in implementations of CBS, see [16], that achieve apparent synchrony while allowing subsystems to fall behind.

8.3 Other Congruence Proofs

Ferreira, Hennessy and Jeffrey [7] use Howe's proof to show that weak bisimulation is a congruence for CML. They use late bisimulation. They leave open the question whether Howe's method can be applied to early bisimulation; this paper does not directly answer their question since late and early semantics and bisimulations coincide for HOBS. A proof for late semantics for HOBS is more elegant than the one here, and can be found in the extended version of the paper [14].

Thomsen proves congruence for CHOCS [18] by adapting the standard proof, but with non-well founded induction. That proof is in effect a similar proof to Howe's technique, but tailored specifically to CHOCS.

Sangiorgi [17] abandons higher order bisimulation for reasons specific to point-to-point communication with private channels, and uses context bisimulation where he adapts the standard proof. His proof is of similar difficulty to the proof presented here, especially in that the case of process application, involving substitution, is difficult.

9. ACKNOWLEDGEMENTS

We would like to thank Dave Sands for bringing Howe's method to our attention, Jörgen Gustavsson, Martin Weichert and Gordon Pace for many discussions, and anonymous referees for constructive comments.

10. REFERENCES

[1] Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research topics in Functional Programming*. Addison-Wesley, 1990.

[2] Bryan Bayerdorffer. Distributed programming with associative broadcast. In *Proceedings of the 27th Annual Hawaii International Conference on Systems Sciences. Volume 2: Software Technology (HICSS92-2)*, pages 353–362, Wailea, HW, USA, 1994.

[3] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In Catuscia Palamidessi, editor, *CONCUR 2000*, volume 1877 of *LNCS*, University Park, PA, USA, August 2000. Springer.

[4] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, Jun 2000.

[5] Cristian Ene and Traian Muntean. Expressivness of point-to-point versus broadcast communications. In *Fundamentals of Computation Theory*, volume 1684 of *LNCS*, September 1999.

[6] Cristian Ene and Traian Muntean. A broadcast-based calculus for communicating systems. In *6th International Workshop on Formal Methods for Parallel Programming: Theory and Applications, San Francisco*, 2001.

[7] William Ferreira, Matthew Hennessy, and Alan Jeffrey. A theory of weak bisimulation for core cml. *Journal of Functional Programming*, 8(5):447–491, 1998.

[8] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, January 1996.

[9] Andrew D. Gordon. Bisimilarity as a theory of functional programming: mini-course. Notes Series BRICS-NS-95-3, BRICS, Department of Computer Science, University of Aarhus, July 1995.

[10] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1 February 1996.

[11] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[12] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[13] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[14] Karol Ostrovský, K. V. S. Prasad, and Walid Taha. Towards a primitive higher order calculus of broadcasting systems (extended version). URL: http://www.cs.chalmers.se/~karol/Papers/, March 2001.

[15] K. V. S. Prasad. Status report on ongoing work: Higher order broadcasting systems and reasoning about broadcasts. Unpublished manuscript, September 1994.

[16] K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25, 1995.

[17] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[18] Bent Thomsen. Plain CHOCS: A second generation calculus for higher order processes. *Acta Informatica*, 30(1):1–59, January 1993.

[19] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis CST–126–96, Dept. of Computer Science, University of Edinburgh, 1996. (also published as ECS-LFCS-96-345).

```
let upon_partitioning x p =
  p.partition<-local_partition_id;
  return p;;

let upon_commiting_transaction t p =
  log t;
  p.reads<-t::p.reads;
  if modifies t p.item_name then
    (p.written<-true;
     p.tw<-t);
  return p;;

let upon_detecting_reconnection p =
  broadcast (fun x -> x.item_name=p.item_name && x.partition!=p.partition,
             fun x -> x.merge p.item_value p.written x);
  iter (fun t ->
          broadcast (fun x -> x.item_name!=p.item_name && x.partition=p.partition && mem t x.reads,
                     fun x -> x.propagate t p.oid x))
       p.reads;
  if p.reads=[] && not p.merged then
    broadcast (fun x -> x.item_name=p.item_name && x.partition!=p.partition,
               fun x -> x.propagate (hd p.reads) p.oid x);
  if not p.merged then
    delay_transactions;
  return p;;

let propagate r label p =
  if label=p.oid
     && length (filter (fun u -> precedes r u || r=u
                               || (not (mem r p.reads) && (precedes p.tw u || p.tw=u)))
                       p.reads)>0
  then
    (declare_inconsistency;
     return p)
  else
    (if not (label=p.oid) then
       iter (fun t -> broadcast (fun x -> x.item_name!=p.item_name && x.partition=p.partition
                                       && mem t x.reads,
                                 fun x -> x.propagate t label x))
            (filter (fun t -> precedes r t || r=t
                           || (not (mem r p.reads) && (precedes p.tw t || p.tw=t)))
                    p.reads);
     if not p.merged then
        broadcast (fun x -> x.item_name=p.item_name && x.partition!=p.partition && x.written,
                   fun x->x.propagate (hd p.reads) label x);
     return p);;

let merge new_value modified p =
  if modified then
    if p.written || write_locked p.item_name then
       declare_inconsistency
    else
      p.item_value<-new_value;
  p.mcount<-p.mcount+1;
  if p.mcount=(count_objects-count_local_objects p.partition)
  then
    p.merged<-true;
  return p;;
```

Figure 6: Database Consistency Protocol Implementation