# A Fully Abstract May Testing Semantics for Concurrent Objects

Alan Jeffrey
CTI, DePaul University
Chicago, IL, USA
ajeffrey@cs.depaul.edu

Julian Rathke
COGS, University of Sussex
Brighton, UK
julianr@cogs.susx.ac.uk

## Abstract

*This paper provides a fully abstract semantics for a variant of the concurrent object calculus. We define may testing for concurrent object components and then characterise it using a trace semantics inspired by UML interaction diagrams. The main result of this paper is to show that the trace semantics is fully abstract for may testing. This is the first such result for a concurrent object language.*

## 1. Introduction

Abadi and Cardelli's [1] object calculus is a minimal language for investigating features of object languages such as encapsulated state, subtyping, and self variables. Gordon and Hankin [7] added concurrent features to the object calculus, to produce the concurrent object calculus.

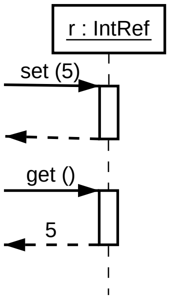
Prior work on the object calculus has concentrated on the operational behaviour of object systems, and type systems which provide type safety guarantees. The closest paper to ours is Gordon and Rees's [8] fully abstract semantics for the immutable single-threaded object calculus. There has been no work on providing fully abstract semantics for concurrent mutable objects.

In this paper, we present the first fully abstract testing semantics for a variant of Gordon and Hankin's concurrent object calculus without subtyping. The lack of subtyping here affords a simpler presentation of the labelled transitions and traces but we anticipate that the proof techniques used here are robust enough to cater for subtyping also. This semantics was inspired by UML interaction diagrams [4], which are a common tool for visualising interactions with object systems.

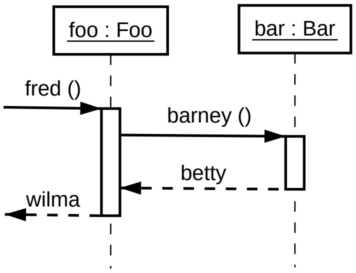
### 1.1. Interaction diagrams

Interaction diagrams (in particular sequence diagrams) were developed by Jacobson, and are now part of the Unified Modeling Language standard [4]. Interaction diagrams record the messages sent between objects of a component in an object system. These messages include method calls and returns (interaction diagrams include other forms of message, but we will not use these in this paper).

A simple interaction with an integer reference object `r` of type `IntRef` has it receive two incoming method calls `set(5)` and `get()`, for which it produces appropriate return values:



A more complex interaction allows a method call on one object to call methods on other objects:



Here, the object `foo` has one incoming call to `fred()`, makes one outgoing call to `barney()`, receives the result `betty` back, then returns `wilma` itself. This illustrates the four messages which may be sent during an interaction: incoming and outgoing method calls, and matching outgoing and incoming returns.

In this paper, we use a textual representation of an interaction, as a trace, which is just a sequence of messages. In the above example, `foo` has the trace:

$$\begin{array}{l}\langle \mathsf{call\,foo.fred()}\rangle ?\\ \langle \mathsf{call\,bar.barney()}\rangle !\\ \langle \mathsf{return\,betty}\rangle ?\\ \langle \mathsf{return\,wilma}\rangle !\end{array}$$

where we mark incoming messages with ? and outgoing messages with !. The object `bar` has the matching trace:

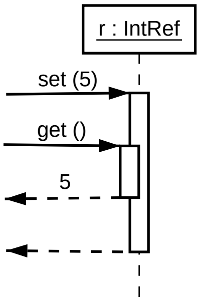
$$\begin{array}{l}\langle \mathsf{call\,bar.barney()}\rangle ?\\ \langle \mathsf{return\,betty}\rangle !\end{array}$$

and so composing these two traces together, we get that the whole system has the trace:

$$\begin{array}{l}\langle \mathsf{call\,foo.fred()}\rangle ?\\ \langle \mathsf{return\,wilma}\rangle !\end{array}$$

There are two additions we will make to the UML message notation: adding thread identifiers, and making name scope more explicit.

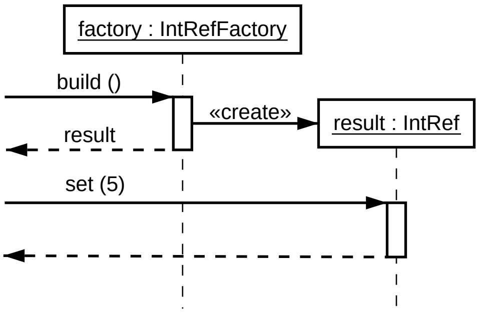
Sequence diagrams can be used for multithreaded applications, for example:



Here, two threads independently call methods of the object `r`, creating a race condition. In our textual representation, we give the threads names, and we decorate each message with the thread responsible for the message:

$$\begin{array}{l}\mathsf{thread1}\langle \mathsf{call\,r.set(5)}\rangle ?\\ \mathsf{thread2}\langle \mathsf{call\,r.get()}\rangle ?\\ \mathsf{thread2}\langle \mathsf{return\,5}\rangle !\\ \mathsf{thread1}\langle \mathsf{return}\rangle !\end{array}$$

The other addition we make to the notation is to make the scope of names more explicit. For example, consider the following interaction with a factory object, which builds new integer reference objects:



In the textual representation of this trace, we need to make clear that the `result` object has not been seen before by the environment (it is a genuinely new object, not a recycled object). We do this by decorating the label with $\nu$ to indicate that the `result` object is new:

$$\begin{array}{l}\mathsf{thread1}\langle \mathsf{call\,factory.build()}\rangle ?\\ \nu(\mathsf{result:IntRef})\,.\,\mathsf{thread1}\langle \mathsf{return\,result}\rangle !\\ \mathsf{thread1}\langle \mathsf{call\,result.set(5)}\rangle ?\\ \mathsf{thread1}\langle \mathsf{return}\rangle !\end{array}$$

As well as allowing the system to generate new names on outgoing messages, we allow the environment to generate new names on incoming messages. This style of dealing with fresh names comes originally from the $\pi$-calculus [19, 18], and has since been used in other languages, notably the $\nu$-calculus [23].

We have now presented informally all of the machinery required by our semantics for objects:

- The semantics of a system is given by a set of traces, where a trace is a sequence of messages corresponding to one interaction.
- Messages are incoming or outgoing message calls, or matching outgoing or incoming returns.
- Messages are decorated with thread identifiers.
- Messages may include fresh names.

We have only used a very small subset of sequence diagrams, which in turn is a very small subset of UML, but in this paper we will show that this small subset is very expressive, and in particular provides a fully abstract semantics.

### 1.2. The object calculus

The object calculus is a minimal language for modelling object-based programming. Abadi and Cardelli [1] provided a type system and operational semantics for a variety of object calculi, and proved type safety for them. Gordon and Hankin [7] have since extended this language to include concurrent features.

In this paper, we shall investigate a variant of Gordon and Hankin's concurrent object calculus, which includes:

- A heap of named objects and threads.
- Threads can call or update object methods, can compare object or thread names for equality, can create new objects and threads and can discover their own thread name.
- An operational semantics based on the π-calculus [19, 18], and a simple type system.
- A trace semantics as discussed in Section 1.1.

We are not considering many of the more advanced features of the object calculus or the concurrent object calculus, such as recursive types, object cloning and object locking. This is just for simplicity, we do not see any technical problems with incorporating these features into our language.

In another strand of research Di Blasio and Fisher [3] also designed a calculus for modelling imperative, concurrent object-based systems. As with Abadi and Cardelli's object calculus and its various extensions, the emphasis in Di Blasio and Fisher's work is again on type systems and safety properties for them.

### 1.3. Full abstraction

The problem of full abstraction was first introduced by Milner [17], and investigated in depth by Plotkin [24]. Full abstraction was first proposed for variants of the λ-calculus, but has since been investigated for process algebras [9], the π-calculus [6, 10], the ν-calculus [23, 14], Concurrent ML [5, 15], and the immutable object calculus [8].

One way to define a semantics for a programming language is to define:

- A language of *typed components* $C$ which can be *composed* $C_1 \parallel C_2$. (In this paper, components are programs in the concurrent object calculus.)
- A notion of when a component *is successful*. (In this paper, we use a special `succ` method call to indicate a successful component although the theory is robust enough that any other suitable observable would suffice).

We can then define the *may testing preorder* [21, 9] as $C_1 \sqsubseteq_{\mathsf{may}} C_2$ whenever:

> for any appropriately typed $C$
> if $C_1 \parallel C$ is successful then $C_2 \parallel C$ is successful

Unfortunately, although it is very simple to define, and is quite intuitive, may testing is often very difficult to reason about directly, because of the quantification over 'any appropriately typed $C$'. In practice, we require a proof technique which we can use to show results about may testing.

One approach is to use a *trace* semantics, given by defining possible executions of components $C \overset{s}{\Longrightarrow} C'$ where $s$ is a sequence of messages. We then write $\mathsf{Traces}(C)$ for the set of all traces of $C$. We say that:

- Traces are *sound* for may testing when
  $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$ implies $C_1 \sqsubseteq_{\mathsf{may}} C_2$.
- Traces are *complete* for may testing when
  $C_1 \sqsubseteq_{\mathsf{may}} C_2$ implies $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$.
- Traces are *fully abstract* for may testing when
  they are both sound and complete.

A fully abstract trace model can be a useful tool in understanding a behavioural equivalence in the sense that, in order to be sound, the traces used to build the model must, at minimum, account for all of the possible interactions a system of objects may have with its environment and, in order to be complete, the interactions described by the traces must be genuine. This is taken to mean that for each interaction described by a trace there is an actual system of objects which can play the role of the environment in that interaction. Therefore, to obtain a fully abstract trace model it is necessary to describe all possible interactions accurately.

Establishing full abstraction for a language which includes features such as higher-order programming, new name generation, and heap-based objects is often non-trivial. For example, Pitts and Stark introduced the ν-calculus [23], as a minimal higher-order language with name generation, by extending the simply typed λ-calculus with an abstract type of names, together with a name generator and an equality test. Even this minimal language is remarkably difficult to reason about, and there is no known fully abstract semantics for it [15].

### 1.4. Contribution of this paper

In this paper, we present a variant of Gordon and Hankin's concurrent object calculus, which is in turn an extension of Abadi and Cardelli's object calculus. The only significant departures from Gordon and Hankin's concurrent object calculus is that we use named threads, where they use anonymous threads and we restrict the calculus to disallow subtyping and recursive types. Whilst this latter restriction does move us away from the essence of object-oriented programming it is imposed so as to keep the technical presentation as simple as possible at this stage. The re-introduction of these features into the type system would affect the behavioural theory in what we expect to be a predictable way and anticipate that techniques employed in [11] and those presented here can be combined to give a similar treatment for a concurrent object language with subtyping.

We provide the calculus with an operational semantics, and a trace semantics, and then show that the trace semantics is fully abstract for may testing. This is the first full abstraction result for a concurrent object-based language.

## 2. Concurrent objects

In this section, we will present the syntax, static semantics and dynamic semantics of our concurrent object calculus. This is a variant of Gordon and Hankin's concurrent object calculus with named rather than anonymous threads.

### 2.1. Syntax

The syntax for the concurrent object calculus we will use in this paper is given in Figure 1. For the remainder of this section, we will provide an informal description of the syntax.

In examples, we will often make use of base types such as integers and booleans: these are not part of our formal system, but will make examples easier to present. They could be comfortably included in the language without changing the theory significantly. We will also make use of some syntax sugar, which we will define formally at the end of this section.

A component $C$ is a collection of named objects $n[O]$ and threads $n\langle t\rangle$. For example, one possible component consisting of an integer reference $p$ and a thread $n$ which increments the reference is:

$$\begin{array}{l} p[\mathsf{contents} = 5] \parallel \\ n\langle \mathsf{let}\ x = p.\mathsf{contents}\ \mathsf{in}\ p.\mathsf{contents} := x+1\rangle \end{array}$$

We also use the ν-notation of the π-calculus [18] to indicate which names are private, and not known to the outside world. By default, names are public, and have to be marked by ν in order to be considered private. For example, $n$ is private, and $p$ is public in:

$$\begin{array}{l} \nu(n : \mathsf{thread})\,.\,( \\ \quad p[\mathsf{contents} = 5] \parallel \\ \quad n\langle \mathsf{let}\ x = p.\mathsf{contents}\ \mathsf{in}\ p.\mathsf{contents} := x+1\rangle \\ ) \end{array}$$

An object $[O]$ consists of a set of named methods, for example an integer reference with set and get methods might be written:

$$\begin{array}{l} [ \\ \quad \mathsf{contents} = 5, \\ \quad \mathsf{set} = \varsigma(\mathsf{this} : \mathsf{IntRef})\,.\,\lambda(x : \mathsf{Int})\,.\,\mathsf{this.contents} := x; x, \\ \quad \mathsf{get} = \varsigma(\mathsf{this} : \mathsf{IntRef})\,.\,\lambda()\,.\,\mathsf{this.contents} \\ ] \end{array}$$

Here we are using Abadi and Cardelli's [1] notation for fields as zero-argument methods. Each method $M$ consists of a self name as well as a list of parameters and a body. For example, the set method above has self name $(\mathsf{this} : \mathsf{IntRef})$, parameters $(x : \mathsf{Int})$, and body $(\mathsf{this.contents} := x)$.

(Readers familiar with Abadi and Cardelli's work will note that we are taking parameterized methods as primitive, rather than defining them as syntax sugar. This is necessary for our semantics, which is based on method calls with arguments and return values.)

A thread $\langle t\rangle$ consists of a stack of let-expressions, terminated either by a return value:

$$\langle \mathsf{let}\ x_1 : T_1 = e_1\ \mathsf{in}\ \cdots \mathsf{let}\ x_n : T_n = e_n\ \mathsf{in}\ v\rangle$$

or by a deadlocked stop thread:

$$\langle \mathsf{let}\ x_1 : T_1 = e_1\ \mathsf{in}\ \cdots \mathsf{let}\ x_n : T_n = e_n\ \mathsf{in}\ \mathsf{stop}\rangle$$

Each expression is either itself a thread, or:

- an if expression $\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$,
- a method call $v.l(\vec{v})$,
- a method update $v.l \Leftarrow M$,
- a new object $\mathsf{new}[O]$,
- a new thread $\mathsf{new}\langle t\rangle$, or
- the current thread name $\mathsf{currentthread}$.

Each value is either a name or a variable. We will discuss types in Section 2.2. In examples, we will often use some syntax sugar. We will elide types from variable and name binders, where they can be reconstructed. We write $e;t$ as syntax sugar for $\mathsf{let}\ x = e\ \mathsf{in}\ t$ when $x$ is a fresh variable. We use Abadi and Cardelli's definition of fields $f$ as zero-argument methods: a field declaration $f = v$ in an object is syntax sugar for a method declaration $f = \varsigma(n : T)\,.\,\lambda()\,.\,v$; a field type $f : T$ in an object type is syntax sugar for a method type $f : () \to T$; a field access expression $v.f$ is syntax sugar for a method call $v.f()$; and a field update expression $v.f := v'$ is syntax sugar for a method update $v.f \Leftarrow (\varsigma(n : T)\,.\,\lambda()\,.\,v)$.

In addition, we have restricted many subexpressions of an expression to be values rather than full expressions, for example in a method call $v.l(\vec{v})$ we require the object and the arguments to be values rather than expressions $e.l(\vec{e})$. This makes the operational semantics much easier to define, and does not restrict the expressivity of the language, for example we can define $(e.l(\vec{e})) \equiv (\mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{let}\ \vec{x} = \vec{e}\ \mathsf{in}\ x.l(\vec{x}))$.

Similarly, the distinction between threads and expressions makes the operational semantics much simpler, but we can treat any expression as a thread by η-converting it: $\langle e\rangle \equiv \langle \mathsf{let}\ x = e\ \mathsf{in}\ x\rangle$.

$$\begin{array}{llll}
\text{Components:} & C & ::= & \mathbf{0} \mid C \parallel C \mid \nu(n:T)\,.\,C \mid n[O] \mid n\langle t\rangle \\
\text{Objects:} & O & ::= & l = M, \ldots, l = M \\
\text{Methods:} & M & ::= & \varsigma(n:T)\,.\,\lambda(x:T,\ldots,x:T)\,.\,t \\
\text{Threads:} & t & ::= & v \mid \mathsf{stop} \mid \mathsf{let}\ x:T = e\ \mathsf{in}\ t \\
\text{Expressions:} & e & ::= & t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid v.l(v,\ldots,v) \mid n.l \Leftarrow M \mid \mathsf{new}[O] \mid \mathsf{new}\langle t\rangle \mid \mathsf{currentthread} \\
\text{Values:} & v & ::= & x \mid n \\
\text{Types:} & T & ::= & \mathsf{thread} \mid \mathsf{none} \mid [l:L,\ldots,l:L] \\
\text{Method types:} & L & ::= & (T,\ldots,T) \to T
\end{array}$$

We assume grammars for variables $x, y$, names $n, p$ and method identifiers $l$.
In objects and object types, we require method identifiers $l$ to be unique, and viewed up to reordering.

**Figure 1. Syntax of the concurrent object calculus**

### 2.2. Static semantics

The static semantics for our concurrent object calculus is given in Figures 2–6. Most of the rules are straightforward adaptations of those given by Abadi and Cardelli [1]. The main judgement is $\Delta \vdash C : \Theta$ which is read as 'the component $C$ uses names $\Delta$ and defines names $\Theta$'. For example, if we define $C_1(v)$, $C_2$ and $\mathsf{IntRef}$ as:

$$\begin{array}{l}
C_1(v) \equiv p[ \\
\quad \mathsf{contents} = v, \\
\quad \mathsf{set} = \varsigma(\mathsf{this}:\mathsf{IntRef})\,.\,\lambda(x:\mathsf{Int})\,.\,\mathsf{this.contents} := x; x, \\
\quad \mathsf{get} = \varsigma(\mathsf{this}:\mathsf{IntRef})\,.\,\lambda()\,.\,\mathsf{this.contents} \\
] \\
C_2 \equiv n\langle \\
\quad \mathsf{let}\ x = p.\mathsf{get}()\ \mathsf{in}\ p.\mathsf{set}(x+1); \mathsf{stop} \\
\rangle \\
\mathsf{IntRef} \equiv [ \\
\quad \mathsf{contents} : \mathsf{Int}, \mathsf{set} : (\mathsf{Int}) \to \mathsf{Int}, \mathsf{get} : () \to \mathsf{Int} \\
]
\end{array}$$

then we can deduce (if $v : \mathsf{Int}$):

$$\begin{array}{rll}
n:\mathsf{thread} & \vdash & C_1(v) : (p:\mathsf{IntRef}) \\
p:\mathsf{IntRef} & \vdash & C_2 : (n:\mathsf{thread}) \\
 & \vdash & (C_1(v) \parallel C_2) : (p:\mathsf{IntRef}, n:\mathsf{thread}) \\
 & \vdash & \nu(n:\mathsf{thread})\,.\,(C_1(v) \parallel C_2) : (p:\mathsf{IntRef})
\end{array}$$

We will now introduce an important requirement of our components, that they be *write closed*:

> Whenever $\Delta \vdash C : \Theta$ contains a subexpression of the form $n.l \Leftarrow M$ with $n$ free, then $n$ appears in $\Theta$.

This is intended to capture the common software engineering requirement that components should not export mutable fields, instead they should export suitable $\mathsf{get}$ and $\mathsf{set}$ methods. For example, the configurations $C_1$ and $C_2$ above are write closed, since the only updates are to $\mathsf{this}$, but the following component which writes directly to $p.\mathsf{contents}$ is not write closed:

$$C_2' \equiv n\langle \mathsf{let}\ x = p.\mathsf{contents}\ \mathsf{in}\ p.\mathsf{contents} := x+1; \mathsf{stop}\rangle$$

For the remainder of the paper we will require components to be write closed. This makes developing a fully abstract semantics much simpler, since we do not need to model method update directly.

### 2.3. Dynamic semantics

The dynamic semantics for our concurrent object calculus is given in Figures 7–10.

We define two relations:

- $C \xrightarrow{\tau} C'$ when $C$ can reduce to $C'$ by the interaction of a thread and an object (either a method call or a method update).
- $C \xrightarrow{\beta} C'$ when $C$ can reduce to $C'$ by a thread acting independently of any other threads or objects.

We write $C \to C'$ when either $C \xrightarrow{\tau} C'$ or $C \xrightarrow{\beta} C'$; we write $C \Rightarrow C'$ when $C \to^* C'$.

The important property of $\beta$-reductions is that they do not introduce race conditions (and hence nondeterminism), where $\tau$-reductions may introduce race conditions. This is discussed further in the full version of this paper.

For example, recalling the definition of $C_1(v)$ from Section 2.2 we have:

$$\begin{array}{l}
C_1(5) \parallel n\langle \mathsf{let}\ x = p.\mathsf{get}()\ \mathsf{in}\ p.\mathsf{set}(x+1); \mathsf{stop}\rangle \\
\quad \xrightarrow{\tau} C_1(5) \parallel n\langle \mathsf{let}\ x = p.\mathsf{contents}\ \mathsf{in}\ p.\mathsf{set}(x+1); \mathsf{stop}\rangle \\
\quad \xrightarrow{\tau} C_1(5) \parallel n\langle \mathsf{let}\ x = 5\ \mathsf{in}\ p.\mathsf{set}(x+1); \mathsf{stop}\rangle \\
\quad \xrightarrow{\beta}{}^{*} C_1(5) \parallel n\langle p.\mathsf{set}(6); \mathsf{stop}\rangle
\end{array}$$

$$\frac{}{\Delta \vdash \mathbf{0} : ()} \qquad \frac{\Delta, \Theta_2 \vdash C_1 : \Theta_1 \quad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash (C_1 \parallel C_2) : (\Theta_1, \Theta_2)} \qquad \frac{\Delta \vdash C : \Theta, n : T}{\Delta \vdash \nu(n : T) \,.\, C : \Theta} \qquad \frac{; \Delta, n : T \vdash [O] : T}{\Delta \vdash n[O] : (n : T)} \qquad \frac{; \Delta, n : \mathsf{thread} \vdash t : \mathsf{none}}{\Delta \vdash n\langle t \rangle : (n : \mathsf{thread})}$$

**Figure 2. Rules for judgement $\Delta \vdash C : \Theta$**

$$\frac{\Gamma; \Delta \vdash M_1 : T.l_1 \quad \cdots \quad \Gamma; \Delta \vdash M_k : T.l_k}{\Gamma; \Delta \vdash [l_1 = M_1, \ldots, l_k = M_k] : T}$$

**Figure 3. Rule for judgement $\Gamma; \Delta \vdash [O] : T$ (when $T = [l_1 : L_1, \ldots, l_k : L_k]$)**

$$\frac{\Gamma, x_1 : T_1, \ldots, x_k : T_k; \Delta, n : T \vdash t : U}{\Gamma; \Delta \vdash \varsigma(n : T) \,.\, \lambda(x_1 : T_1, \ldots, x_k : T_k) \,.\, t : T.l}$$

**Figure 4. Rule for judgement $\Gamma; \Delta \vdash M : T.l$ (when $T = [\ldots, l : (T_1, \ldots, T_k) \rightarrow U, \ldots]$ and $T.l$ is the record $l$ selected from $T$)**

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash v_1 : T_1 \quad \Gamma; \Delta \vdash v_2 : T_1 \\ \Gamma; \Delta \vdash e_1 : T_2 \quad \Gamma; \Delta \vdash e_2 : T_2\end{array}}{\Gamma; \Delta \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2} \qquad \frac{\begin{array}{c}\Gamma; \Delta \vdash v : [\ldots, l : (T_1, \ldots, T_k) \rightarrow T, \ldots] \\ \Gamma; \Delta \vdash v_1 : T_1 \quad \cdots \quad \Gamma; \Delta \vdash v_k : T_k\end{array}}{\Gamma; \Delta \vdash v.l(v_1, \ldots, v_k) : T} \qquad \frac{\Gamma; \Delta \vdash n : T \quad \Gamma; \Delta \vdash M : T.l}{\Gamma; \Delta \vdash n.l \Leftarrow M : T}$$

$$\frac{\Gamma; \Delta \vdash [O] : T}{\Gamma; \Delta \vdash \mathsf{new}[O] : T} \qquad \frac{\Gamma; \Delta \vdash t : T}{\Gamma; \Delta \vdash \mathsf{new}\langle t \rangle : \mathsf{thread}} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{currentthread} : \mathsf{thread}}$$

$$\frac{\Gamma; \Delta \vdash e : T_1 \quad \Gamma, x : T_1; \Delta \vdash t : T_2}{\Gamma; \Delta \vdash \mathsf{let}\ x : T_1 = e\ \mathsf{in}\ t : T_2} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{stop} : T} \qquad \frac{}{\Gamma, x : T, \Gamma'; \Delta \vdash x : T} \qquad \frac{}{\Gamma; \Delta, n : T, \Delta' \vdash n : T}$$

**Figure 5. Rules for judgement $\Gamma; \Delta \vdash e : T$**

Variable contexts: $\Gamma \quad ::= \quad x : T, \ldots, x : T$ Name contexts: $\Delta, \Theta, \Sigma, \Phi \quad ::= \quad n : T, \ldots, n : T$

In variable contexts, variables must be unique, and are viewed up to reordering.
In name contexts, names must be unique, types must not be $\mathsf{none}$, and are viewed up to reordering.

**Figure 6. Syntax of name and variable contexts**

$$\mathbf{0} \parallel C \equiv C \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3) \qquad C_1 \parallel C_2 \equiv C_2 \parallel C_1$$

$$C_1 \parallel \nu(n:T)\,.\,C_2 \equiv \nu(n:T)\,.\,(C_1 \parallel C_2) \qquad \nu(n_1:T_1)\,.\,\nu(n_2:T_2)\,.\,C \equiv \nu(n_2:T_2)\,.\,\nu(n_1:T_1)\,.\,C$$

**Figure 7. Axioms for structural congruence (where $n$ is not free in $C_1$)**

$$\begin{array}{rcll}
n\langle \mathsf{let}\ x:T = v\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle t[v/x]\rangle & \\
n\langle \mathsf{let}\ x:T = (\mathsf{let}\ x_1:T_1 = e_1\ \mathsf{in}\ e_2)\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{let}\ x_1:T_1 = e_1\ \mathsf{in}\ (\mathsf{let}\ x:T = e_2\ \mathsf{in}\ t)\rangle & \\
n\langle \mathsf{let}\ x:T = (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{let}\ x:T = e_1\ \mathsf{in}\ t\rangle & \\
n\langle \mathsf{let}\ x:T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{let}\ x:T = e_2\ \mathsf{in}\ t\rangle & (v_1 \neq v_2) \\
n\langle \mathsf{let}\ x:T = \mathsf{new}[O]\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & \nu(p:T)\,.\,(p[O] \parallel n\langle \mathsf{let}\ x:T = p\ \mathsf{in}\ t\rangle) & (p \text{ is not free in } O \text{ or } t) \\
n\langle \mathsf{let}\ x:T = \mathsf{new}\langle f\rangle\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & \nu(p:T)\,.\,(p\langle f\rangle \parallel n\langle \mathsf{let}\ x:T = p\ \mathsf{in}\ t\rangle) & (p \text{ is not free in } t \text{ or } f) \\
n\langle \mathsf{let}\ x:T = \mathsf{currentthread}\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{let}\ x:T = n\ \mathsf{in}\ t\rangle & \\
n\langle \mathsf{let}\ x:T = \mathsf{stop}\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{stop}\rangle & \\
p[O] \parallel n\langle \mathsf{let}\ x:T = p.l(\vec{v})\ \mathsf{in}\ t\rangle & \xrightarrow{\tau} & p[O] \parallel n\langle \mathsf{let}\ x:T = O.l(p)(\vec{v})\ \mathsf{in}\ t\rangle & \\
p[O] \parallel n\langle \mathsf{let}\ x:T = p.l \Leftarrow M\ \mathsf{in}\ t\rangle & \xrightarrow{\tau} & p[O.l \Leftarrow M] \parallel n\langle \mathsf{let}\ x:T = p\ \mathsf{in}\ t\rangle &
\end{array}$$

**Figure 8. Axioms for reduction precongruence**

$$\frac{C \equiv\ \xrightarrow{\beta}\ \equiv C'}{C \xrightarrow{\beta} C'} \qquad \frac{C \xrightarrow{\beta} C'}{C \parallel C'' \xrightarrow{\beta} C' \parallel C''} \qquad \frac{C \xrightarrow{\beta} C'}{\nu(n:T)\,.\,C \xrightarrow{\beta} \nu(n:T)\,.\,C'}$$

$$\frac{C \equiv\ \xrightarrow{\tau}\ \equiv C'}{C \xrightarrow{\tau} C'} \qquad \frac{C \xrightarrow{\tau} C'}{C \parallel C'' \xrightarrow{\tau} C' \parallel C''} \qquad \frac{C \xrightarrow{\tau} C'}{\nu(n:T)\,.\,C \xrightarrow{\tau} \nu(n:T)\,.\,C'}$$

**Figure 9. Rules for reduction precongruence**

$$(\vec{l} = \vec{M}, l = M).l(p)(\vec{v}) = t[p/n, \vec{v}/\vec{x}] \qquad (\vec{l} = \vec{M}, l = M').l \Leftarrow M = (\vec{l} = \vec{M}, l = M)$$

**Figure 10. Definition of $O.l(p)(\vec{v})$ and $O.l \Leftarrow M$ where $M = \varsigma(n:T)\,.\,\lambda(\vec{x}:\vec{T})\,.\,t$**

$$
\begin{aligned}
&\xrightarrow{\tau} C_1(5) \parallel n\langle p.\mathsf{contents} := 6; 6; \mathsf{stop}\rangle \\
&\xrightarrow{\tau} C_1(6) \parallel n\langle p; 6; \mathsf{stop}\rangle \\
&\xrightarrow{\beta}{}^{*}\, C_1(6) \parallel n\langle \mathsf{stop}\rangle
\end{aligned}
$$

as expected.

### 2.4. Testing preorder

We will now define the testing semantics for our concurrent object calculus. We will do this by defining a notion of *barb* for a component, and let a successful component be one which communicates on that barb. This is similar to the use of barbs in process algebra [20].

Let the type $\mathsf{barb}$ be defined:

$$\mathsf{barb} = [\mathsf{succ} : () \to \mathsf{none}]$$

for some fresh method name $\mathsf{succ}$. We say that a component *strongly barbs* on $b : \mathsf{barb}$ written $C{\downarrow_b}$ if and only if:

$$C \equiv \nu(\vec{n} : \vec{T})\,.\,(C' \parallel n\langle \mathsf{let}\ x : \mathsf{none} = b.\mathsf{succ}()\ \mathsf{in}\ t\rangle)$$

for $b \notin \vec{n}$ and *barbs* on $b : \mathsf{barb}$ written $C{\Downarrow_b}$ if and only if:

$$C \Rightarrow C'{\downarrow_b}$$

When $\Delta \vdash C_1 : \Theta$ and $\Delta \vdash C_2 : \Theta$ we write $\Delta \models C_1 \sqsubseteq_{\mathsf{may}} C_2 : \Theta$ if and only if:

for any $\Delta', \Theta, b : \mathsf{barb} \vdash C : \Delta$ if $(C_1 \parallel C){\Downarrow_b}$ then $(C_2 \parallel C){\Downarrow_b}$

This is a straightforward adaptation of the standard [9] definition of may testing for concurrent systems.

### 2.5. Trace semantics

The trace semantics for the concurrent object calculus is given by a labelled transition system (lts) with judgements:

$$(\Delta \vdash C : \Theta) \xrightarrow{\alpha} (\Delta' \vdash C' : \Theta')$$

The lts is given for components extended by introducing two new expressions:

$$e ::= \cdots \mid \mathsf{block} \mid \mathsf{return}\, v$$

These new threads are included purely to assist in the description of the lts and are intended to represent a command for a thread to wait for some unknown interaction with the environment and a command for a thread to report a value to the environment and then to go back to a $\mathsf{block}$ed state. There are no reductions associated with these commands and they may be typed as:

$$\frac{}{\Gamma;\Delta \vdash \mathsf{block} : T} \qquad \frac{\Gamma;\Delta \vdash v : T}{\Gamma;\Delta \vdash \mathsf{return}\, v : U}$$

where $T$ and $U$ are any types. The lts for our concurrent object language are given in Figures 11–14.

For example if we define:

$$
\begin{aligned}
\Theta &\equiv (p : \mathsf{IntRef}) \\
\Theta' &\equiv (p : \mathsf{IntRef}, n : \mathsf{thread})
\end{aligned}
$$

then (where $C_1(v)$ is defined in Section 2.2) we have:

$$
\begin{aligned}
&(\vdash C_1(5) : \Theta) \\
&\quad \xrightarrow{\nu(n{:}\mathsf{thread}).n\langle \mathsf{call}\ p.\mathsf{get}()\rangle?} \\
&(\vdash (C_1(5) \parallel n\langle \mathsf{let}\ x = p.\mathsf{get}()\ \mathsf{in}\ \mathsf{return}\, x\rangle) : \Theta') \\
&\quad \Rightarrow \\
&(\vdash (C_1(5) \parallel n\langle \mathsf{return}\, 5\rangle) : \Theta') \\
&\quad \xrightarrow{n\langle \mathsf{return}\, 5\rangle!} \\
&(\vdash (C_1(5) \parallel n\langle \mathsf{block}\rangle) : \Theta') \\
&\quad \xrightarrow{n\langle \mathsf{call}\ p.\mathsf{set}(6)\rangle?} \\
&(\vdash (C_1(5) \parallel n\langle \mathsf{let}\ x = p.\mathsf{set}(6)\ \mathsf{in}\ \mathsf{return}\, x\rangle) : \Theta') \\
&\quad \Rightarrow \\
&(\vdash (C_1(6) \parallel n\langle \mathsf{return}\, 6\rangle) : \Theta') \\
&\quad \xrightarrow{n\langle \mathsf{return}\, 6\rangle!} \\
&(\vdash (C_1(5) \parallel n\langle \mathsf{block}\rangle) : \Theta')
\end{aligned}
$$

For any component $(\Delta \vdash C : \Theta)$ we define its traces to be:

$$\mathsf{Traces}(\Delta \vdash C : \Theta) = \{ s \mid (\Delta \vdash C : \Theta) \overset{s}{\Longrightarrow} (\Delta' \vdash C' : \Theta') \}$$

We will now show that this trace semantics is fully abstract for may testing.

## 3. Soundness of traces for may testing

### 3.1. The merge operator

Define the partial *merge* operator $C_1 \mathbin{\wedge\!\!\wedge} C_2$ as the symmetric operator defined up to $\equiv$ where:

$$
\begin{aligned}
\mathbf{0} \mathbin{\wedge\!\!\wedge} C &\equiv C \\
(\nu(p : T)\,.\,C_1) \mathbin{\wedge\!\!\wedge} C_2 &\equiv \nu(p : T)\,.\,(C_1 \mathbin{\wedge\!\!\wedge} C_2) \\
(p[O] \parallel C_1) \mathbin{\wedge\!\!\wedge} C_2 &\equiv p[O] \parallel (C_1 \mathbin{\wedge\!\!\wedge} C_2) \\
(p\langle t\rangle \parallel C_1) \mathbin{\wedge\!\!\wedge} C_2 &\equiv p\langle t\rangle \parallel (C_1 \mathbin{\wedge\!\!\wedge} C_2) \\
(n\langle t_1\rangle \parallel C_1) \mathbin{\wedge\!\!\wedge} (n\langle t_2\rangle \parallel C_2) &\equiv n\langle t_1 \mathbin{\wedge\!\!\wedge} t_2\rangle \parallel (C_1 \mathbin{\wedge\!\!\wedge} C_2)
\end{aligned}
$$

when $n \notin \mathsf{dom}\,(C_2)$ and $p \notin \mathsf{fn}\,(C_2)$.

Define the partial merge operator $t_1 \mathbin{\wedge\!\!\wedge} t_2$ as the symmetric

$$(\Delta, n : \mathsf{thread} \vdash C : \Theta) \xrightarrow{n\langle \mathsf{call}\, p.l(\vec{v})\rangle ?} (\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = p.l(\vec{v})\ \mathsf{in}\ \mathsf{return}\, x\rangle : (n : \mathsf{thread}, \Theta))$$
$$\text{(when } ;\Delta, n : \mathsf{thread}, \Theta \vdash p.l(\vec{v}) : T \text{ and } p \in \Theta)$$

$$(\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle : \Theta) \xrightarrow{n\langle \mathsf{call}\, p.l(\vec{v})\rangle ?} (\Delta \vdash C \parallel n\langle \mathsf{let}\, y : U = p.l(\vec{v})\ \mathsf{in}\ \mathsf{let}\, x : T = \mathsf{return}\, y\ \mathsf{in}\ t\rangle : \Theta)$$
$$\text{(when } ;\Delta, \Theta \vdash p.l(\vec{v}) : U \text{ and } p \in \Theta)$$

$$(\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle : \Theta) \xrightarrow{n\langle \mathsf{return}\, v\rangle ?} (\Delta \vdash C \parallel n\langle t[v/x]\rangle : \Theta)$$
$$\text{(when } ;\Delta, \Theta \vdash v : T)$$

$$(\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = p.l(\vec{v})\ \mathsf{in}\ t\rangle : \Theta) \xrightarrow{n\langle \mathsf{call}\, p.l(\vec{v})\rangle !} (\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle : \Theta)$$
$$\text{(when } p \in \Delta)$$

$$(\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{return}\, v\ \mathsf{in}\ t\rangle : \Theta) \xrightarrow{n\langle \mathsf{return}\, v\rangle !} (\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle : \Theta)$$

**Figure 11. Axioms for labelled transition system** $(\Delta \vdash C : \Theta) \xrightarrow{\alpha} (\Delta' \vdash C' : \Theta')$

$$\frac{(\Delta \vdash C : (\Theta, n : T)) \xrightarrow{a} (\Delta' \vdash C' : (\Theta', n : T))}{(\Delta \vdash \nu(n : T)\,.\,C : \Theta) \xrightarrow{a} (\Delta' \vdash \nu(n : T)\,.\,C' : \Theta')}\,(n \text{ is not free in } a)$$

$$\frac{(\Delta \vdash C : (\Theta, n : T)) \xrightarrow{\gamma !} (\Delta' \vdash C' : \Theta')}{(\Delta \vdash \nu(n : T)\,.\,C : \Theta) \xrightarrow{\nu(n:T).\gamma !} (\Delta' \vdash C' : \Theta')}\,(n \text{ is free in } \gamma)$$

$$\frac{(\Delta, n : T \vdash C : \Theta) \xrightarrow{\gamma ?} (\Delta' \vdash C' : \Theta')}{(\Delta \vdash C : \Theta) \xrightarrow{\nu(n:T).\gamma ?} (\Delta' \vdash C' : \Theta')}\,(n \text{ is free in } \gamma, T \text{ is not } \mathsf{none})$$

**Figure 12. Rules for labelled transition system** $(\Delta \vdash C : \Theta) \xrightarrow{\alpha} (\Delta' \vdash C' : \Theta')$

$$\frac{C \Rightarrow C'}{(\Delta \vdash C : \Theta) \xRightarrow{\varepsilon} (\Delta \vdash C' : \Theta)} \qquad \frac{(\Delta \vdash C : \Theta) \xrightarrow{a} (\Delta' \vdash C' : \Theta')}{(\Delta \vdash C : \Theta) \xRightarrow{a} (\Delta' \vdash C' : \Theta')}$$

$$\frac{(\Delta \vdash C : \Theta) \xRightarrow{s} (\Delta' \vdash C' : \Theta') \xRightarrow{s'} (\Delta'' \vdash C'' : \Theta'')}{(\Delta \vdash C : \Theta) \xRightarrow{ss'} (\Delta'' \vdash C'' : \Theta'')}$$

**Figure 13. Rules for trace semantics** $(\Delta \vdash C : \Theta) \xRightarrow{s} (\Delta' \vdash C' : \Theta')$

$$\begin{array}{lrcl}
\text{Basic labels:} & \gamma & ::= & n\langle \mathsf{call}\, p.l(\vec{v})\rangle \mid n\langle \mathsf{return}\, v\rangle \mid \nu(n : T)\,.\,\gamma \\
\text{Visible labels:} & a & ::= & \gamma ? \mid \gamma ! \\
\text{Traces:} & q, r, s & ::= & a \cdots a
\end{array}$$

**Figure 14. Syntax of labels and traces**

operator where:

$$\begin{aligned}
&(\mathsf{let}\ x = \mathsf{block}\ \mathsf{in}\ t) \mathbin{\wedge\!\!\wedge} \mathsf{stop}\\
&\quad\equiv\quad \mathsf{stop}\\
&(\mathsf{let}\ x = \mathsf{block}\ \mathsf{in}\ t) \mathbin{\wedge\!\!\wedge} v\\
&\quad\equiv\quad v\\
&(\mathsf{let}\ x = \mathsf{block}\ \mathsf{in}\ t_1) \mathbin{\wedge\!\!\wedge} (\mathsf{let}\ y = \mathsf{return}\, v\ \mathsf{in}\ t_2)\\
&\quad\equiv\quad (\mathsf{let}\ y = \mathsf{block}\ \mathsf{in}\ t_2) \mathbin{\wedge\!\!\wedge} (t_1[v/x])\\
&(\mathsf{let}\ x = \mathsf{block}\ \mathsf{in}\ t_1) \mathbin{\wedge\!\!\wedge} (\mathsf{let}\ y = e\ \mathsf{in}\ t_2)\\
&\quad\equiv\quad \mathsf{let}\ y = e\ \mathsf{in}\ ((\mathsf{let}\ x = \mathsf{block}\ \mathsf{in}\ t_1) \mathbin{\wedge\!\!\wedge} t_2)
\end{aligned}$$

when $e$ is block/return free and $y \notin \mathsf{fv}\ (t_2)$.

**Lemma 3.1** *If* $\Delta \vdash (C_1 \parallel C_2) : \Theta$ *then* $(C_1 \mathbin{\wedge\!\!\wedge} C_2) \equiv (C_1 \parallel C_2)$.

**Lemma 3.2** *If* $C_1 \mathbin{\wedge\!\!\wedge} C_2 \equiv C$ *then* $C{\downarrow_b}$ *if and only if* $C_1{\downarrow_b}$ *or* $C_2{\downarrow_b}$.

### 3.2. Trace composition and decomposition

Given a trace $s$ we write $\overline{s}$ for the complementary trace:

$$\overline{\varepsilon} = \varepsilon \qquad \overline{s_1 s_2} = \overline{s_1}\,\overline{s_2} \qquad \overline{\gamma?} = \gamma! \qquad \overline{\gamma!} = \gamma?$$

**Proposition 3.3 (Trace composition/decomposition)** *For any components* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma)$ *and* $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma)$ *such that* $C_1 \mathbin{\wedge\!\!\wedge} C_2 \equiv C$*, we have:*

*(i) If* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \overset{s}{\Longrightarrow} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$
*and* $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \overset{\overline{s}}{\Longrightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$
*then* $C \Rightarrow C'$
*where* $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \mathbin{\wedge\!\!\wedge} C_2') \equiv C'$.

*(ii) If* $C \Rightarrow C'$ *then there exists some trace* $s$
*such that* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \overset{s}{\Longrightarrow} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$
*and* $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \overset{\overline{s}}{\Longrightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$
*where* $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \mathbin{\wedge\!\!\wedge} C_2') \equiv C'$.

### 3.3. Proof of soundness

**Theorem 3.4 (Soundness of traces for may testing)**
*If* $\mathsf{Traces}(\Delta \vdash C_1 : \Theta) \subseteq \mathsf{Traces}(\Delta \vdash C_2 : \Theta)$
*then* $\Delta \models C_1 \sqsubseteq_{may} C_2 : \Theta$

**Proof:** Suppose that $\mathsf{Traces}(\Delta \vdash C_1 : \Theta) \subseteq \mathsf{Traces}(\Delta \vdash C_2 : \Theta)$ and that we have $(\Theta, b : \mathsf{barb} \vdash C_0 : \Delta)$ such that $(C_1 \parallel C_0){\Downarrow_b}$; we must show that $(C_1 \parallel C_0){\Downarrow_b}$ also.

Now, since $(C_1 \parallel C_0){\Downarrow_b}$ we can find $C'$ such that:

$$(C_1 \parallel C_0) \Rightarrow C'{\downarrow_b}$$

By Lemma 3.1 we know that: $C_1 \parallel C_0 \equiv C_1 \mathbin{\wedge\!\!\wedge} C_0$ and so by Proposition 3.3 we have:

$$(\Delta, b : \mathsf{barb} \vdash C_1 : \Theta) \overset{s}{\Longrightarrow} (\Delta', b : \mathsf{barb} \vdash C_1' : \Theta', \Sigma')$$
$$(\Theta, b : \mathsf{barb} \vdash C_0 : \Delta) \overset{\overline{s}}{\Longrightarrow} (\Theta', b : \mathsf{barb} \vdash C_0' : \Delta', \Sigma')$$

where: $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta) \,.\, (C_1' \mathbin{\wedge\!\!\wedge} C_0') \equiv C'$. By Lemma 3.2 we know that either $C_1'{\downarrow_b}$ or $C_0'{\downarrow_b}$.

- **Case** $(C_1'{\downarrow_b})$. Since $C_1'{\downarrow_b}$ we can find a label $\omega!$ of the form:

  $$\omega! = \nu(\vec{n} : \vec{T}) \,.\, n\langle \mathsf{call}\, b.\mathsf{succ}() \rangle!$$

  such that:

  $$(\Delta', b : \mathsf{barb} \vdash C_1' : \Theta', \Sigma') \overset{\omega!}{\rightarrow}$$

  Since $\mathsf{Traces}(\Delta \vdash C_1 : \Theta) \subseteq \mathsf{Traces}(\Delta \vdash C_2 : \Theta)$ we have:

  $$(\Delta, b : \mathsf{barb} \vdash C_2 : \Theta) \overset{s}{\Longrightarrow} (\Delta', b : \mathsf{barb} \vdash C_2' : \Theta', \Sigma') \overset{\omega!}{\rightarrow}$$

  By Lemma 3.1 we know that $C_2 \parallel C_0 \equiv C_2 \mathbin{\wedge\!\!\wedge} C_0$ and so by Proposition 3.3 we have: $(C_2 \parallel C_0) \Rightarrow C''$ where:

  $$\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta) \,.\, (C_2' \mathbin{\wedge\!\!\wedge} C_0') \equiv C''$$

  By Lemma 3.2, since $C_2'{\downarrow_b}$ we have that $C''{\downarrow_b}$, and so $(C_2 \parallel C_0){\Downarrow_b}$ as required.

- **Case** $(C_0'{\downarrow_b})$. Similar to the above. □

### 3.4. Example

We can use the trace semantics now to show a simple equivalence, derived from one of the Meyer-Sieber examples [16] for idealised Algol. The following two components are may testing equivalent if $x \notin t, C$:

$$C \parallel n\langle \mathsf{let}\ x = \mathsf{new}[O]\ \mathsf{in}\ t \rangle \qquad \text{and} \qquad C \parallel n\langle t \rangle.$$

To show this we use a meta-property of reduction that traces are invariant under β-reductions so we reduce our obligation to showing equivalence of

$$(\nu(p) \,.\, p[O]) \parallel C \parallel n\langle t \rangle \qquad \text{and} \qquad C \parallel n\langle t \rangle.$$

This can be achieved by establishing a simple invariant (by examining each labelled transition rule) that each of these can perform the same actions to get to a similar syntactic form.

## 4. Completeness of traces for may testing

A key step in demonstrating completeness of traces for may testing is to find, for each trace, a component which exhibits that trace; we call this problem *definability*. However, we only actually require definability for traces which originated from well-typed components. To identify these, in the full version of the paper, we present a type system for traces $\Delta \vdash s : \mathsf{trace}\ \Theta$ which captures exactly those we require. For present purposes the reader may think of the well-typed traces $\Delta \vdash s : \mathsf{trace}\ \Theta$ simply as those which are generated by well-typed components $\Delta \vdash C : \Theta$. Also, due to an amount of latency and asynchrony in the labelled transition system, to demonstrate definability, we found it necessary to define an information order $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$ for typed traces which incorporates prefixing, input receptivity [12], and commutativity of certain actions. Details of this may be found in the full version of the paper.

**Lemma 4.1 (Trace Duality)** *If* $\Delta \vdash s : \mathsf{trace}\ \Theta$
*then* $\Theta \vdash \overline{s} : \mathsf{trace}\ \Delta$

**Proposition 4.2 (Trace Subject Reduction)** *If* $\Delta \vdash C : \Theta$ *and* $(\Delta \vdash C : \Theta) \xRightarrow{s} (\Delta' \vdash C' : \Theta')$ *then* $\Delta \vdash s : \mathsf{trace}\ \Theta$ *and* $\Delta' \vdash C' : \Theta'$.

**Lemma 4.3 (Information Order Duality)**
*If* $\Delta \vdash r\gamma! \sqsubseteq s\gamma! : \mathsf{trace}\ \Theta$ *and* $\mathsf{fn}\ (\gamma) \cap \Theta(s) = \emptyset$ *and* $\gamma! \notin s, r$
*then* $\Theta \vdash \overline{s} \sqsubseteq \overline{r} : \mathsf{trace}\ \Delta$.

**Proposition 4.4 (Information Order Closure)**
*If* $(\Delta \vdash C : \Theta) \xRightarrow{s}$ *and* $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$
*then* $(\Delta \vdash C : \Theta) \xRightarrow{r}$ .

In the full version of this paper, we define a component $\mathsf{Config}\ (\Delta \vdash s : \mathsf{trace}\ \Theta)$ for any typed trace $\Delta \vdash s : \mathsf{trace}\ \Theta$, and show the following results about it.

**Lemma 4.5** *If* $\Delta \vdash s : \mathsf{trace}\ \Theta$
*then* $\Delta \vdash \mathsf{Config}\ (\Delta \vdash s : \mathsf{trace}\ \Theta) : \Theta$.

**Proposition 4.6 (Definability)** *For any* $\Delta \vdash s : \mathsf{trace}\ \Theta$
*we have* $(\Delta \vdash \mathsf{Config}\ (\Delta \vdash s : \mathsf{trace}\ \Theta) : \Theta) \xRightarrow{r}$
*if and only if* $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$.

The definition of $\mathsf{Config}\ (\Delta \vdash s : \mathsf{trace}\ \Theta)$ is rather lengthy so rather than presenting it in full detail here we simply offer an indication of how it is constructed. Firstly, we construct two objects called Ref and State. The former contains a field holding a pointer to the latter. The State object provides type-indexed families of methods called out, inReturn, and inCall. We also provide object and thread definitions for all those references for which the type demands it, *i.e.* those in $\Theta$. The object definitions provide methods according to the object types, where the method bodies simply indirectly re-route all calls to the appropriate State.inCall. The thread definitions make indirect calls to State.out. It it through these that traces are begun.

The bodies for the out, inReturn, and inCall methods depend on the next action in the trace we are providing definability for. For instance, if the next action to be performed is an output $n\langle \mathsf{call}\ p.l(\vec{v})\rangle!$ then all of the bodies will be a stopped thread save for out which will have a method body which will check that the calling thread is $n$ and, if so, update Ref to point to a new State object which will perform the next action in the trace. It will then indirectly call State.inReturn with the result of calling $p.l(\vec{v})$ (on dangling $p$) to listen for an input interaction (*cf.* the labelled transition rule for output, any subsequent action at this thread must be an input). Having successfully observed an input interaction, the line of interrogation in this thread is complete so it must reset itself by returning to a state in which it makes an indirect call to State.out. Similar definitions are given for each type of action.

We provide no synchronisation in the $\mathsf{Config}\ (\Delta \vdash s : \mathsf{trace}\ \Theta)$ component so that there is no guarantee that the reductions will follow the precise sequence of calls needed to exhibit the trace. However, with respect to may testing, this is irrelevant as we are only looking for one possible successful sequence of execution. We do guarantee the existence of this in Proposition 4.6.

**Theorem 4.7 (Completeness of traces for may testing)**
*If* $\Delta \models C_1 \lesssim_{may} C_2 : \Theta$
*then* $\mathsf{Traces}(\Delta \vdash C_1 : \Theta) \subseteq \mathsf{Traces}(\Delta \vdash C_2 : \Theta)$.

**Proof:** Choose any trace $s_1$ such that:

$$(\Delta \vdash C_1 : \Theta) \xRightarrow{s_1} (\Delta' \vdash C_1' : \Theta')$$

By Proposition 4.2 we have that $\Delta \vdash s_1 : \mathsf{trace}\ \Theta$, and so by Lemma 4.1 we have that $\Theta \vdash \overline{s_1} : \mathsf{trace}\ \Delta$.

Pick a fresh $b : \mathsf{barb}$ and let $\omega!$ be:

$$\omega! = \nu(n : \mathsf{thread})\,.\, n\langle \mathsf{call}\ b.\mathsf{succ}()\rangle!$$

and let $C_0$ be:

$$C_0 = \mathsf{Config}\ (\Theta, b : \mathsf{barb} \vdash \overline{s_1}\omega! : \mathsf{trace}\ \Delta)$$

Then by Proposition 4.6 we have:

$$(\Theta, b : \mathsf{barb} \vdash C_0 : \Delta) \xRightarrow{\overline{s_1}} (\Theta', b : \mathsf{barb} \vdash C_0' : \Delta') \xrightarrow{\omega!}$$

and so $C_0' {\downarrow_b}$. Thus, by Lemma 3.1, Proposition 3.3, and Lemma 3.2 we have $(C_1 \parallel C_0){\Downarrow_b}$, and so $(C_2 \parallel C_0){\Downarrow_b}$ which means that $(C_2 \parallel C_0) \Rightarrow C''{\downarrow_b}$. Thus, by Lemma 3.1 and Proposition 3.3 we can find $s_2$ such that:

$$(\Delta, b : \mathsf{barb} \vdash C_2 : \Theta) \xRightarrow{s_2} (\Delta'', \Phi'' \vdash C_2'' : \Theta'', \Sigma'')$$
$$(\Theta, b : \mathsf{barb} \vdash C_0 : \Delta) \xRightarrow{\overline{s_2}} (\Theta'', \Phi'' \vdash C_0'' : \Delta'', \Sigma'')$$

where: $\nu(\Delta'', \Theta'', \Sigma'', \Phi'' \setminus \Delta, \Theta, b : \mathsf{barb})\,.\,(C_2'' \mathbin{/\!\!\!\setminus\!\!\!\setminus} C_0'') \equiv C''$.

Since $C''{\downarrow_b}$ and we chose $b$ to be fresh, we have that $C_0''{\downarrow_b}$ and hence: $(\Theta, b : \mathsf{barb} \vdash C_0 : \Delta) \xRightarrow{\overline{s_2}\omega!}$ so by Proposition 4.6: $\Theta, b : \mathsf{barb} \vdash \overline{s_2}\omega! \sqsubseteq \overline{s_1}\omega! : \mathsf{trace}\ \Delta$ and so by Lemma 4.3 and weakening: $\Delta \vdash s_1 \sqsubseteq s_2 : \mathsf{trace}\ \Theta$. Thus, by Proposition 4.4 we have: $(\Delta \vdash C_2 : \Theta) \xRightarrow{s_1} (\Delta' \vdash C_2' : \Theta')$ as required. □

## 5. Restricted sub-languages

The proof techniques use to obtain full abstraction here are quite robust and can also be carried out for two restricted sub-languages:

1. The single-threaded sub-language is given by only allowing one name of type thread, and removing new thread creation from the expression language. The definability result for Proposition 4.6 does not use thread creation, so the proof of full abstraction goes through with only minor changes to the proof of Theorem 4.7.
2. The sub-language with only field update (and no method update) can be given the same trace semantics. The definability result for Proposition 4.6 only uses field update, and so the proof of full abstraction goes through unchanged.

Thus, not only do we have a full abstraction result for the concurrent object calculus, we can also specialise the results to become full abstraction result for other related languages.

One change which cannot easily be made is to remove the restriction that components be write closed, since method, and even field, updates are not generally externally observable. It is unlikely that traces which represent write interactions will be definable in the current sense. However, we do believe that the restriction to write closed components is a reasonable one, since it corresponds to existing 'best practice' for component design.

## 6. Conclusions and future work

In this paper we have presented the first fully abstract semantics for concurrent objects. The semantics is fairly simple, and corresponds loosely to some of the messages used in UML interaction diagrams. We do need to road test the trace semantics with some reasonably sized examples to demonstrate that the calculation of traces is tractable.

There are a number of issues left open:

- Our semantics has much of the flavour of game semantics [2, 13], and this connection should be investigated.
- The trace semantics characterise may testing, rather than the more common must testing or bisimulation equivalence.
- The object calculus presented here does not include subtyping. We believe that the techniques of [11] should be applicable to the provision of a fully abstract semantics even in the presence of subtyping.

## References

[1] M. Abadi and L. Cardelli. *A Theory Of Objcets*. Springer-Verlag, 1996.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

[3] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *Proc. CONCUR*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language: User Guide*. Addison Wesley, 1999.

[5] W. Ferreira, M. Hennessy, and A. S. A. Jeffrey. A theory of weak bisimulation for core CML. *J. Functional Programming*, 8(5):447–491, 1998.

[6] M. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus. In *Proc. IEEE Conf. Logic in Computer Science*. IEEE Press, 1996.

[7] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In *Proc. High Level Concurrent Languages*, Electronic Notes in Computer Science. Elsevier, 1998.

[8] A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proc. ACM Symp. Principles of Programming Languages*, pages 386–395. ACM Press, 1996.

[9] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[10] M. Hennessy. A fully abstract denotational semantics for the π-calculus. Technical report 96:04, Univ. Sussex, 1996.

[11] M. Hennessy and J. Rathke. Typed behavioural equivalences for processes in the presence of subtyping. In *Proc. Computing: Australasian Theory Symposium*, Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[12] K. Honda and M. Tokoro. On asynchronous communication semantics. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc. ECOOP Workshop on Object-Based Concurrent Computing*, volume 612 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[13] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163:285–408, 2000.

[14] A. S. A. Jeffrey and J. Rathke. Towards a theory of weak bisimulation for local names. In *Proc. Logic In Computer Science*, pages 56–66. IEEE Computer Society Press, 1999.

[15] A. S. A. Jeffrey and J. Rathke. A theory of bisimulation for a fragment of Concurrent ML with local names. In *Proc. Logic in Computer Science*, pages 311–321. IEEE Press, 2000.

[16] A. Meyer and K. Sieber. Towards a fully abstract semantics for local variables. In *Proc. Symposium on Principles of Programming Languages*, San Diego, pages 191–203. ACM, 1988.

[17] R. Milner. Fully abstract semantics of typed λ-calculi. *Theoret. Comput. Sci.*, 4:1–22, 1977.

[18] R. Milner. *Communicating and Mobile Systems*. Cambridge University Press, 1999.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile proceses. *Inform. and Comput.*, 100(1):1–77, 1992.

[20] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. Int. Colloq. Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[21] J.-H. Morris. Lambda calculus models of programming languages. Dissertation, M.I.T., 1968.

[22] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[23] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. MFCS 93*, pages 122–141. Springer-Verlag, 1993. LNCS 711.

[24] G. Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5:223–256, 1977.