The Join Calculus: a Language for Distributed Mobile Programming

Cédric Fournet[1] and Georges Gonthier[2⋆]

[1] Microsoft Research
[2] INRIA Rocquencourt

Abstract In these notes, we give an overview of the join calculus, its semantics, and its equational theory. The join calculus is a language that models distributed and mobile programming. It is characterized by an explicit notion of locality, a strict adherence to local synchronization, and a direct embedding of the ML programming language. The join calculus is used as the basis for several distributed languages and implementations, such as JoCaml and functional nets.
Local synchronization means that messages always travel to a set destination, and can interact only after they reach that destination; this is required for an efficient implementation. Specifically, the join calculus uses ML's function bindings and pattern-matching on messages to program these synchronizations in a declarative manner.
Formally, the language owes much to concurrency theory, which provides a strong basis for stating and proving the properties of asynchronous programs. Because of several remarkable identities, the theory of process equivalences admits simplifications when applied to the join calculus. We prove several of these identities, and argue that equivalences for the join calculus can be rationally organized into a five-tiered hierarchy, with some trade-off between expressiveness and proof techniques.
We describe the mobility extensions of the core calculus, which allow the programming of agent creation and migration. We briefly present how the calculus has been extended to model distributed failures on the one hand, and cryptographic protocols on the other.

⋆ This work is partly supported by the RNRT project MARVEL 98S0347

Applied Semantics Summer School (Caminha, 9–15 September 2000)
Draft 7/01, pp. 1–66,

Contents

Introduction

Wide-area distributed systems have become an important part of modern programming, yet most distributed programs are still written using traditional languages, designed for sequential architectures. Distribution issues are typically relegated to libraries and informal design patterns, with little support in the language for asynchrony and concurrency. Conversely, distributed constructs are influenced by the local programming model, with for instance a natural bias towards RPCs or RMIs rather than asynchronous message passing.

Needless to say, distributed programs are usually hard to write, much harder to understand and to relate to their specifications, and almost impossible to debug. This is due to essential difficulties, such as asynchrony or partial failures. Nonetheless, it should be possible to provide some high-level language support to address these issues.

The join calculus is an attempt to provide language support for asynchronous, distributed, and mobile programming. While it is clearly not the only approach, it has a simple and well-defined model, which has been used as the basis for several language implementations and also as a specification language for studying the properties of such programs. These notes give an overview of the model, with an emphasis on its operational semantics and its equational theory.

JoCaml [27] is the latest implementation of the join calculus; it is a distributed extension of Objective Caml [29], a typed high-level programming language with a mix of functional, object-oriented, and imperative features. OCaml already provides native-code and bytecode compilers, which is convenient for mobile code. The JoCaml language extends OCaml, in the sense that OCaml programs and libraries are just a special kind of JoCaml programs and libraries. JoCaml also implements strong mobility and provides support for distributed execution, including a dynamic linker and a garbage collector. The language documentation includes extensive tutorials; they can be seen as a concrete counterpart for the material presented in these notes (Sections 1 and 5) with larger programming examples.

In these notes, we present a core calculus, rather than a full-fledged language. This minimalist approach enables us to focus on the essential features of the language, and to develop a simple theory: the calculus provides a precise description of how the distributed implementation should behave and, at the same time, it yields a formal model that is very close to the actual programming language. Thus, one can directly study the correctness of distributed programs, considering them as executable specifications. Ideally, the model should provide strong guiding principles for language design and, conversely, the model should reflect the implementation constraints.

The join calculus started out as an attempt to take the models and methods developed by concurrency theory, and to adapt and apply them to the programming of systems distributed over a wide area network. The plan was to start from Milner's pi calculus [31,30,41], extend it with constructs for locality and mobility, and bring to bear the considerable body of work on process calculi and their equivalences on the problem of programming mobile agents. During

the course of this work, the implementation constraints of asynchronous systems suggested changing the pi calculus's CCS-based communication model. The idea was that the model had to stick to what all basic protocol suites do, and decouple transmission from synchronization, so that synchronization issues can always be resolved locally.

To us, the natural primitives for doing this in a higher-order setting were message passing, function call, and pattern-matching, and these suggested a strong link with the programming language ML. This connection allowed us to reuse a significant portion of the ML programming technology—notably the ML type system—for the "programming" part of our project. Thus the join calculus was born out of the synergy between asynchronous process calculi and ML.

Many of the ideas that we adapted from either source took new meaning in the location-sensitive, asynchronous framework of the join calculus. The connection with ML illuminates the interaction between functional and imperative features, down to their implications for typing. The highly abstract chemical abstract machine of Berry and Boudol [7] yields a much more operational instance for the join calculus. The intricate lattice of equivalences for synchronous and asynchronous calculi [19] simplifies remarkably in the pure asynchronous setting of the join calculus, to the point that we can reasonably concentrate on a single five-tiered hierarchy of equivalences, where each tier can be clearly argued for. These lecture notes contains a summary of these results, as well as much of the rationale that joins them into a coherent whole.

These notes are organized as follows. In Section 1, we gradually introduce the join calculus as a concurrent extension of ML, describe common synchronization patterns, and give its chemical semantics. In Section 2, we define and motivate our main notions of observational equivalences, and illustrate the main proof techniques for establishing equivalences between processes. In Section 3, we refine our framework for reasoning about processes. We introduce intermediate notions of equivalence that account for fairness and gradual commitment, and organize all our equivalences in a hierarchy, according to their discriminating power. In Section 4, we supplement this framework with labeled semantics, and discuss their relation. In Section 5, we finally refine the programming model to account for locality information and agent migration.

1 The core join calculus

Although the join calculus was designed as a process calculus for distributed and mobile computation, it turned out to be a close match to ML-style (impure) functional programming. In these notes we will use this connection to motivate and explain the computational core of the calculus.

First we will introduce the primitives of the join calculus by showing how they mesh in a typical 'mini-ML' lambda-like calculus. At a more concrete level, there is a similar introduction to the JoCaml language as an extension of the OCaml programming language [27].

We will then show how a variety of synchronization primitives can be easily encoded in the join calculus and argue that the general join calculus itself can be encoded by its "asynchronous" fragment, i.e., that function calls are just a form of message-passing. This allows us to limit the formal semantics of the join calculus to its asynchronous core.

We conclude the section by exposing the computational model that underlies the formal semantics of the join calculus. This simple model guarantees that distributed computation can be faithfully represented in the join calculus.

1.1 Concurrent functional programming

Our starting point is a small ML-like syntax for the call-by-value polyadic lambda calculus:

$E, F ::=$		expressions
	x, y, f	variable
$\|$	$f(\widetilde{E})$	function call
$\|$	$\texttt{let}\ x = E\ \texttt{in}\ F$	local value definition
$\|$	$\texttt{let}\ f(\widetilde{x}) = E\ \texttt{in}\ F$	local recursive function definition

The notation $\widetilde{x}$ stands for a possibly empty comma-separated tuple $x_1, x_2, \ldots, x_n$ of variables; similarly $\widetilde{E}$ is a tuple of expressions. We casually assume that the usual Hindley-Milner type system ensures that functions are always called with the proper number of arguments.

We depart from the lambda calculus by taking the ML-style function definition as primitive, rather than λ-expressions, which can be trivially recovered as $\lambda x.E \stackrel{\text{def}}{=} \texttt{let}\ f(x) = E\ \texttt{in}\ f$. This choice is consistent with the usual ML programming practice, and will allow us to integrate smoothly primitives with side effects, such as the join calculus primitives, because the `let` provides a syntactic "place" for the state of the function. An immediate benefit, however, is that values in this calculus consist only of *names*: (globally) free variables, or `let`-defined function names. This allows us to omit altogether curried function calls from our syntax, since the function variable in a call can only be replaced by a function name in a call-by-value computation.

We are going to present the join calculus as an extension of this functional calculus with concurrency. The most straightforward way of doing this would

be to add a new `run` E primitive that returns immediately after starting a concurrent evaluation of E. However, we want to get a model in which we can reason about concurrent behavior, not just a programming language design. In order to develop an operational semantics, we also need a syntax that describes the situation *after* `run` E returns, and the evaluation of E and the evaluation of the expression F that contained the `run` E proceed concurrently. It would be quite awkward to use `run` for this, since it would force us to treat E and F asymmetrically.

It is much more natural to denote the concurrent computation of E and F by $E \mid F$, using a parallel composition operator '|'. However, we must immediately realize the $E \mid F$ cannot be an "expression", since an expression denotes a computation that returns a result, and there is no natural way of defining a unique "result" for $E \mid F$. Therefore we need to extend our calculus with a second sort for *processes*, i.e., computations that aren't expected to produce a result (and hence, don't terminate). In fact, the operands E and F of $E \mid F$ should also be processes, rather than expressions, since they aren't expected to return a result either. Thus, '|' will simply be an operation on processes.

We will use the letters P, Q, R for processes. We still need the `run` P primitive to launch a process from an expression; conversely, we allow `let` constructs in processes, so that processes can evaluate expressions.

P, Q, R	::=		processes
		`let` $x = E$ `in` P	compute expression
	∥	`let` $f(\widetilde{x}) = E$ `in` P	local recursive function definition
	∥	$P \mid Q$	parallel composition
	∥	$\mathbf{0}$	inert process

E, F	::=		expressions
		...	
	∥	`run` P	spawn process

In this minimal syntax, invoking an abstraction for a process P is quite awkward, since it involves computing an expression that calls a function whose body will execute `run` P. To avoid this, we add a "process abstraction" construct to mirror the function abstraction construct; we use a different keyword '`def`' and different call brackets '$\langle\ \rangle$' to enforce the separation of expressions and processes.

P, Q, R	::=		processes
		...	
	∥	$p\langle\widetilde{E}\rangle$	execute abstract process
	∥	`def` $p\langle\widetilde{x}\rangle \triangleright P$ `in` Q	process abstraction

E, F	::=		expressions
		...	
	∥	`def` $p\langle\widetilde{x}\rangle \triangleright P$ `in` E	process abstraction

Operationally, "executing" an abstract process really means computing its parameters and shipping their values to the '`def`', where a new copy of the process

body can be started. Hence, we will use the term *channel* for the "abstract process" defined by a `def`, and *message send* for a "process call" $p\langle\widetilde{E}\rangle$.

Our calculus allows the body E of a function f to contain a subprocess P. From P's point of view, returning the value for f's computation is just sending data away on a special "channel"; we extend our calculus with a `return` primitive to do just that. Since messages carry a tuple of values, the `return` primitive also gives us a notation for returning several values from a function; we just need to extend the `let` so it can bind a tuple of variables to the result of such tuple-valued functions.

$$
\begin{array}{rll}
P, Q, R ::= & & \text{processes} \\
& \ldots & \\
& \mid \ \texttt{return}\ \widetilde{E}\ \texttt{to}\ f & \text{return value(s) to function call} \\
& \mid \ \texttt{let}\ \widetilde{x} = E\ \texttt{in}\ P & \text{compute expression} \\
\\
E, F ::= & & \text{expressions} \\
& \ldots & \\
& \mid \ \texttt{let}\ \widetilde{x} = E\ \texttt{in}\ F & \text{local definition(s)}
\end{array}
$$

Note that the tuple may be empty, for functions that only perform side effects; we write the matching `let` statement $E; P$ (or $E; F$) rather than $\texttt{let} = E\ \texttt{in}\ P$. Also, we will omit the `to` part for `return`s that return to the closest lexically enclosing function.

With the $E; P$ and `return` statements, it is now more convenient to write function bodies as processes rather than expressions. We therefore extend the `def` construct to allow for the direct definition of functions with processes.

$$
\begin{array}{rll}
P, Q, R ::= & & \text{processes} \\
& \ldots & \\
& \mid \ \texttt{def}\ f(\widetilde{x}) \triangleright P\ \texttt{in}\ Q & \text{recursive function definition} \\
\\
E, F ::= & & \text{expressions} \\
& \ldots & \\
& \mid \ \texttt{def}\ f(\widetilde{x}) \triangleright P\ \texttt{in}\ E & \text{recursive function definition}
\end{array}
$$

Hence $\texttt{def}\ f(\widetilde{x}) \triangleright P$ is equivalent to $\texttt{let}\ f(\widetilde{x}) = \texttt{run}\ P$. Conversely, $\texttt{let}\ f(\widetilde{x}) = E$ is equivalent to $\texttt{def}\ f(\widetilde{x}) \triangleright \texttt{return}\ E$, so, in preparation for the next section, we take the `def` form as primitive, and treat the $\texttt{let}\ f(\widetilde{x}) = \ldots$ form as an abbreviation for a `def`.

1.2 Synchronization by pattern-matching

Despite its formal elegance, the formalism we have developed so far has limited expressiveness. While it allows for the generation of concurrent computation, it provides no means for joining together the results of two such computations, or for having any kind of interaction between them, for that matter. Once spawned, a process will be essentially oblivious to its environment.

A whole slew of stateful primitives have been proposed for encapsulating various forms of inter-process interaction: concurrent variables, semaphores, message-passing, futures, rendez-vous, monitors, ... just to name a few. The join calculus distinguishes itself by using that basic staple of ML programming, *pattern-matching*, to provide a declarative means for specifying inter-process synchronization, thus leaving state inside processes, where it rightfully belongs.

Concretely, this is done by allowing the joint definition of several functions and/or channels by matching concurrent call and message patterns; in a nutshell, by allowing the '|' operator on the left of the '$\triangleright$' definition symbol. The syntax for doing this is a bit more complex, partly because we also want to allow for multiple patterns, so we create new categories for *definitions* and *join patterns*.

P, Q, R	::=		processes
		...	
	$\parallel$	$\texttt{def}\ D\ \texttt{in}\ P$	process/function definition
E, F	::=		expressions
		...	
	$\parallel$	$\texttt{def}\ D\ \texttt{in}\ E$	process/function definition
D	::=		definitions
		$J \triangleright P$	execution rule
	$\parallel$	$D \wedge D'$	alternative definitions
	$\parallel$	$\top$	empty definition
J	::=		join patterns
		$x\langle\widetilde{y}\rangle$	message send pattern
	$\parallel$	$x(\widetilde{y})$	function call pattern
	$\parallel$	$J \mid J'$	synchronization

Definitions whose join pattern consists of a single message pattern (or a single call pattern) correspond to the abstract processes (or functions) presented above. More interestingly, the meaning of a joint definition $p\langle x\rangle \mid q\langle y\rangle \ \triangleright P$ is that, each time messages are concurrently sent on *both* the p and q channels, the process P is run with the parameters x and y set to the contents of the p and q messages, respectively. For instance, this two-message pattern may be used to join the results of two concurrent computations:

$$
\begin{array}{l}
\texttt{def}\ \mathit{jointCall}(f_1, f_2, t) \triangleright \\
\quad \texttt{def}\ p\langle x\rangle \mid q\langle y\rangle \triangleright \texttt{return}\ x, y\ \texttt{in} \\
\quad p\langle f_1(t)\rangle \mid q\langle f_2(t)\rangle\ \texttt{in} \\
\texttt{let}\ x, y = \mathit{jointCall}(\cos, \sin, 0.1)\ \texttt{in}\ \ldots
\end{array}
$$

In this example, each call to $\mathit{jointCall}(f_1, f_2, t)$ starts two processes $p\langle f_1(t)\rangle$ and $q\langle f_2(t)\rangle$ that compute $f_1(t)$ and $f_2(t)$ in parallel and send their values on the local channels p and q, respectively. When both messages have been sent, the

inner `def` rule joins the two messages and triggers the `return` process, which returns the pair $f_1(t), f_2(t)$ to the caller of $jointCall(f_1, f_2, t)$.

If there is at least a function call in the pattern J of a rule $J \triangleright P$, then the process body P may contain a `return` for that call, as in the functional core. We can thus code an asynchronous pi calculus "channel" x as follows

$$\begin{array}{l}\texttt{def } \overline{x}\langle v\rangle \mid x() \triangleright \texttt{return } v \texttt{ in}\\ \ldots \overline{x}\langle E\rangle \ldots \mid \texttt{let } u = x() \texttt{ in } P\end{array}$$

Since a pi calculus channel x supports two different operations, sending and receiving, we need two join calculus names to implement it: a channel name $\overline{x}$ for sending, and a function name x for receiving a value. The meaning of the joint definition is that a call to $x()$ returns a value v that was sent on an $\overline{x}\langle\rangle$ message. Note that the pi calculus term $\overline{x}\langle v\rangle$, which denotes a primitive "send" operation on a channel x, gets encoded as $\overline{x}\langle v\rangle$, which sends a message on the (ordinary) channel name $\overline{x}$ in the join calculus. The primitive pi calculus reception process $x(u).P$, which runs $P\{v/u\}$ after receiving a single $\overline{x}\langle v\rangle$ message, gets encoded as $\texttt{let } u = x() \texttt{ in } P$.

In the example above, u will be thus bound to the value of E for the execution of P—if there are no other $x()$ calls or $\overline{x}\langle v\rangle$ messages around. If there are, then the behavior is not deterministic: the join calculus semantics does ensure that each $x()$ call grabs at most one $\overline{x}\langle v\rangle$ message, and that each $\overline{x}\langle v\rangle$ message fulfills at most one $x()$ call (the `def` rule *consumes* its join pattern), but it does not specify how the available $x()$ and $\overline{x}\langle v\rangle$ are paired. Any leftover calls or messages (there cannot be both[1]) simply wait for further messages or calls to complete; however the calculus makes no guarantee as to the order in which they will complete. To summarize, running

$$\overline{x}\langle 1\rangle \mid \overline{x}\langle 2\rangle \mid \overline{x}\langle 3\rangle \mid (print(x()); print(x()); \mathbf{0})$$

can print `1 2` and stop with a leftover $\overline{x}\langle 3\rangle$ message, or print `3 2` and stop with a leftover $\overline{x}\langle 1\rangle$ message, etc. The join calculus semantics allows any of these possibilities. On the other hand,

$$\overline{x}\langle 1\rangle \mid \big(print(x()); (\overline{x}\langle 2\rangle \mid print(x()); \mathbf{0})\big)$$

can only print `1 2`, as the $\overline{x}\langle 2\rangle$ message is sent only after the first $x()$ call has completed.

We can use higher-order constructs to encapsulate this encoding in a single *newChannel* function that creates a new channel and returns its interface:

$$\begin{array}{l}\texttt{def } newChannel() \triangleright\\ \quad \texttt{def } send\langle v\rangle \mid receive() \triangleright \texttt{return } v \texttt{ in}\\ \quad \texttt{return } send, receive \texttt{ in}\\ \texttt{let } \overline{x}, x = newChannel() \texttt{ in}\\ \texttt{let } \overline{y}, y = newChannel() \texttt{ in } \ldots\end{array}$$

[1] Under the fairness assumptions usually provided by implementations, and implied by most process equivalences (see Section 3.2).

(Because the join calculus has no data structures, we encode the channel "objects" by the tuple of join calculus names $send, receive$ that implement their operations. In JoCaml, we would return a record.)

This kind of higher-order abstraction allows us to return only some of the names that define an object's behavior, so that the other names remain *private*. An especially common idiom is to keep the *state* of a concurrent object in a single private message, and to use function names for the *methods*. Since the state remains private, it is trivial to ensure that there is always exactly one state message available. For example, here is the join calculus encoding of a "shared variable" object.

$$\begin{array}{l}\texttt{def}\ \mathit{newVar}(v_0) \triangleright \\ \quad \texttt{def}\ \mathit{put}(w) \mid \mathit{val}\langle v\rangle \triangleright \mathit{val}\langle w\rangle \mid \texttt{return} \\ \quad\quad \wedge\ \mathit{get}() \mid \mathit{val}\langle v\rangle \triangleright \mathit{val}\langle v\rangle \mid \texttt{return}\ v\ \texttt{in} \\ \quad \mathit{val}\langle v_0\rangle \mid \texttt{return}\ \mathit{put}, \mathit{get}\ \texttt{in}\ \ldots\end{array}$$

The inner definition has two rules that define three names—two functions *put* and *get*, and a channel *val*. The *val* name remains private and always carries a single state message with the current value; it initially carries the value v_0 passed as a parameter when the shared variable is created. Note that since the state must be joined with a call to run a method, it is easy to ensure that at most one method runs at a time, by reissuing the state message only when the method completes. This is the classical *monitor* construct (also known as a *synchronized object*).

It is often natural to use different channel names to denote different synchronization states of an object. Compare, for instance, the encoding for a shared variable above with the encoding for a one-place buffer:

$$\begin{array}{l}\texttt{def}\ \mathit{newBuf}() \triangleright \\ \quad \texttt{def}\ \mathit{put}(v) \mid \mathit{empty}\langle\rangle \triangleright \mathit{full}\langle v\rangle \mid \texttt{return} \\ \quad\quad \wedge\ \mathit{get}() \mid \mathit{full}\langle v\rangle \triangleright \mathit{empty}\langle\rangle \mid \texttt{return}\ v\ \texttt{in} \\ \quad \mathit{empty}\langle\rangle \mid \texttt{return}\ \mathit{put}, \mathit{get}\ \texttt{in}\ \ldots\end{array}$$

The join calculus gives us considerably more flexibility for describing the synchronization behavior of the object. For instance, the state may at times be divided among several concurrent asynchronous state messages, and method calls may be joined with several of these messages. In short, we may use a Petri net rather than a finite state machine to describe the synchronization behavior. For example, a concurrent two-place buffer might be coded as

$$\begin{array}{l}\texttt{def}\ \mathit{newBuf2}() \triangleright \\ \quad \texttt{def}\ \mathit{put}(v) \mid \mathit{emptyTail}\langle\rangle \triangleright \mathit{fullTail}\langle v\rangle \mid \texttt{return} \\ \quad\quad \wedge\ \mathit{emptyHead}\langle\rangle \mid \mathit{fullTail}\langle v\rangle \triangleright \mathit{fullHead}\langle v\rangle \mid \mathit{emptyTail}\langle\rangle \\ \quad\quad \wedge\ \mathit{get}() \mid \mathit{fullHead}\langle v\rangle \triangleright \mathit{emptyHead}\langle\rangle \mid \texttt{return}\ v\ \texttt{in} \\ \quad \mathit{emptyHead}\langle\rangle \mid \mathit{emptyTail}\langle\rangle \mid \texttt{return}\ \mathit{put}, \mathit{get}\ \texttt{in}\ \ldots\end{array}$$

Note that these concurrent objects are just a programming idiom; there is nothing specific to them in the join calculus, which can accomodate other programming models equally well. For instance, we get the *actors* model if we make

the "methods" asynchronous, and put the methods' code inside the state, i.e., the state contains a function that processes the method message and returns a function for processing the next message:

$$\begin{array}{l}\texttt{def } \mathit{newActor}(\mathit{initBehavior}) \triangleright \\ \quad \texttt{def } \mathit{request}\langle v\rangle \mid \mathit{state}\langle \mathit{behavior}\rangle \triangleright \mathit{state}\langle \mathit{behavior}(v)\rangle \texttt{ in} \\ \quad \mathit{state}\langle \mathit{initBehavior}\rangle \mid \texttt{return } \mathit{request} \texttt{ in } \ldots\end{array}$$

We can also synchronize several calls, to model for instance the CCS *synchronous channels*. (In this case, we have to specify to which calls the `return` statements return).

$$\begin{array}{l}\texttt{def } \mathit{newCCSchan}() \triangleright \\ \quad \texttt{def } \mathit{send}(v) \mid \mathit{receive}() \triangleright \texttt{return to } \mathit{send} \mid \texttt{return } v \texttt{ to } \mathit{receive} \texttt{ in} \\ \texttt{return } \mathit{send}, \mathit{receive} \texttt{ in} \\ \ldots\end{array}$$

The Ada *rendez-vous* can be modeled similarly; in this case, the "acceptor" task sends a message-processing function, and the function result is returned to the "caller" task:

$$\begin{array}{l}\texttt{def } \mathit{newRendezVous}() \triangleright \\ \quad \texttt{def } \mathit{call}(v) \mid \mathit{accept}(f) \triangleright \\ \quad\quad \texttt{let } r = f(v) \texttt{ in } (\texttt{return } r \texttt{ to } \mathit{call} \mid \texttt{return to } \mathit{accept}) \texttt{ in} \\ \quad \texttt{return } \mathit{call}, \mathit{accept} \texttt{ in } \ldots\end{array}$$

The `let` ensures that the acceptor is suspended until the rendez-vous processing has completed, so that the message-processing "function" can freely access the acceptor's imperative variables without race conditions. An `accept` $e(x)$ `do` E `end` Ada statement would thus be modeled as $\mathit{accept}_e(\lambda x.E)$.

In theory, adding any of the above constructs—even the imperative variable—to the concurrent lambda calculus of Section 1.1 gives a formalism equivalent to the join calculus in terms of expressiveness. What are, then, the advantages of the join pattern construct? If a specific synchronization device is taken as primitive (say, the one-place buffer), other devices must be coded in terms of that primitive. These encodings are usually abstracted as functions (often the only available form of abstraction). This means that the synchronization behavior of an application that relies mostly on non-primitive devices is mostly hidden in the side effects of functions. On the contrary, the join calculus encodings we have presented above make the synchronization behavior of the encoding devices explicit. The join calculus gives us a general language for writing synchronization devices; devices are just common idioms in that language. This makes it much easier to turn from one type of device to another, to use several kinds of devices, or even to *combine* devices, e.g., provide method rendez-vous for a synchronized object.

Also, the join calculus syntax favors statically binding code to a synchronization event. This increases the "referential transparency"[2] of join calculus

[2] A misnomer, of course, since a language with synchronization must have side effects.

programs, because this code is easily found by a lexical lookup of the functions and channels involved in the event. In other words, this code gives a first approximation of what happens when the corresponding functions are called, and the corresponding messages are sent. The longer the code in the definition, the better the approximation. It should be noted that for most of the devices we have presented here this static code is very short. The pi calculus asynchronous channel encoding is probably a worst case. It only says "a value sent on $\overline{x}$ can be returned by an $x()$ call", so that any other properties of the channel have to be inferred from the dynamic configuration of the program.

Finally—and this was actually the main motivation for the join calculus design—the join calculus synchronization can always be performed *locally*. Any contention between messages and/or calls is resolved at the site that holds their joint definition. The transmission of messages, calls, and returns, on the other hand, can never cause contention. As we will see in section 4, this property is required if the calculus is to be used for modeling distributed systems, and *a fortiori* mobility.

This property is the reason for which the CCS (or pi calculus) "external choice" operator is conspicuously absent from our list of encodings : this device can express an atomic choice between communication offers on arbitrary channels, and thus intrinsically creates non-local contention. Its join calculus encoding would necessarily include some rather cumbersome mechanism for resolving this contention (see [35]). We should however point out that global atomic choice is a rarely needed device, and that the join calculus provides versatile local choice in the form of alternative rules.

Even without choice, the pi calculus does not enjoy the locality property (unlike many other models, such as monitors or actors, which do exhibit locality). This is because the contention between senders on one hand, and receivers on the other hand, cannot be both resolved simultaneously and locally at a sender or at a receiver site. The join calculus encoding introduces an explicit "channel definition" site in which the resolution can take place.

1.3 The asynchronous core

In spite of some simplifications at the end of Section 1.1, the syntax of the general join calculus is rather large and complex. It would be preferable to carry out a few other simplifications before trying to formulate a precise operational semantics for the calculus. In particular, the semantics of rendez-vous type definitions, where two calls are synchronized, is going to be painful to describe formally.

A standard trick in operational semantics is to use $Q\{E/f(\widetilde{v})\}$ to denote a state where Q is computing an expression E for the function call $f(\widetilde{v})$. This neatly averts the need for a new syntax for such states, but clearly does not work if E is run by a rendez-vous between two calls $f(\widetilde{v})$ and $g(\widetilde{w})$; we would need new syntax for that. For that matter, we would also need new syntax for the case where E has forked off a separate process P that contains some `return` F `to` f primitives. We would also need special rules to allow messages (and possibly definitions) to move in and out of such "inner processes".

Fortunately, we can avoid these complications by *defining* function calls by a message protocol. We will take literally the analogy we used to introduce the `return` primitive, and actually implement the `return` with a message send on a continuation channel. The function call will be implemented by a message carrying both the arguments and the continuation channel. The precise "wiring" of the continuations will specify exactly the evaluation order.

We assume that, for each function name f, we have a fresh channel name κ_f for the return channel of f (we will reuse the function name f for the call channel). Then the continuation-passing style (CPS) encoding of function calls in the join calculus can be specified by the equations:

$$
\begin{aligned}
f(\widetilde{x}) &\stackrel{\text{def}}{=} f\langle \widetilde{x}, \kappa_f \rangle \qquad \text{(in join patterns } J\text{)} \\
\texttt{return } \widetilde{E} \texttt{ to } f &\stackrel{\text{def}}{=} \kappa_f\langle \widetilde{E} \rangle \\
p\langle E_1, \ldots, E_n \rangle &\stackrel{\text{def}}{=} \texttt{let } v_1 = E_1 \texttt{ in} \\
&\quad\ \vdots \\
&\quad \texttt{let } v_n = E_n \texttt{ in } p\langle v_1, \ldots, v_n \rangle \\
&\quad \text{(when at least one } E_i \text{ is not a variable)} \\
\texttt{let } v = u \texttt{ in } P &\stackrel{\text{def}}{=} P\{u/v\} \\
\texttt{let } \widetilde{x} = f(\widetilde{E}) \texttt{ in } P &\stackrel{\text{def}}{=} \texttt{def } \kappa\langle \widetilde{x} \rangle \triangleright P \texttt{ in } f\langle \widetilde{E}, \kappa \rangle \\
\texttt{let } \widetilde{x} = \texttt{def } D \texttt{ in } E \texttt{ in } P &\stackrel{\text{def}}{=} \texttt{def } D \texttt{ in let } \widetilde{x} = E \texttt{ in } P \\
\texttt{let } \widetilde{x} = \texttt{let } \widetilde{y} = F \texttt{ in } E \texttt{ in } P &\stackrel{\text{def}}{=} \texttt{let } \widetilde{y} = F \texttt{ in let } \widetilde{x} = E \texttt{ in } P \\
\texttt{let } = \texttt{run } P \texttt{ in } Q &\stackrel{\text{def}}{=} P \mid Q
\end{aligned}
$$

The equations above make the general assumption that there are no spurious variable captures : the names κ and $v_1, \ldots, v_n$ are fresh, and the names defined by D or bound by $\widetilde{y}$ are not free in P in the `let-def` and `let-let` equations. Expanded repeatedly, these definitions translate up to alpha-conversion any full join calculus process into an equivalent asynchronous join calculus process—one that does not involve function calls or `let`, `return`, or `run` primitives.

In addition to choosing an evaluation order, the translation assigns a "continuation capture" semantics to multiple returns: if several `return`s are executed for the same function call, then the calling context will be executed several times with different return values. While this feature may be useful for implementing, e.g., a fail-retry construct, it is not really compatible with the stack implementation of function calls, so JoCaml for instance puts severe syntactic restrictions on the use of `return` statements to rule out multiple returns.

We could apply further internal encodings to remove other "complex" features from the join calculus: alternative definitions, n-way join patterns for $n \neq 2$, n-ary messages for $n \neq 1$, even $\mathbf{0}$... However, none of these "simplifications" would really simplify the operational semantics, and the behavior of the encoding would be significantly more complex than behavior of the encoded term. This is not the case for the CPS encoding presented above; we are simply providing, within the asynchronous join calculus, the "extra syntax" that was called for at the top of this section.

$P, Q, R ::=$		processes
	$x\langle\widetilde{y}\rangle$	asynchronous message
$\parallel$	$\texttt{def}\ D\ \texttt{in}\ P$	local definition
$\parallel$	$P \mid Q$	parallel composition
$\parallel$	$\mathbf{0}$	inert process
$D ::=$		definitions
	$J \triangleright P$	reaction rule
$\parallel$	$D \wedge D'$	composition
$\parallel$	$\top$	void definition
$J ::=$		join patterns
	$x\langle\widetilde{y}\rangle$	message pattern
$\parallel$	$J \mid J'$	synchronization

Figure 1. Syntax for the core join calculus

1.4 Operational semantics

Since in the asynchronous join calculus the only expressions are variables, we can altogether do away with the "expressions" syntactic class. The remaining syntax, summarized in Figure 1, is quite regular: messages and parallel composition in both processes and patterns, plus definitions in processes.

The precise binding rules for the asynchronous join calculus are those of mutually-recursive functions in ML:

(*i*) A rule $x_1\langle\widetilde{y}_1\rangle \mid \cdots \mid x_v\langle\widetilde{y}_n\rangle \triangleright P$ binds the formal parameters $\widetilde{y}_1, \ldots, \widetilde{y}_n$ with scope P; the variables in each tuple $\widetilde{y}_i$ must be distinct, and the tuples must be pairwise disjoint. Also, the rule *defines* the channel names $x_1, \ldots, x_n$.

(*ii*) A definition $\texttt{def}\ J_1 \triangleright P_1 \wedge \ldots \wedge J_n \triangleright P_n\ \texttt{in}\ Q$ recursively binds in $Q, P_1, \ldots, P_n$ all the channel names defined in $J_1 \triangleright P_1, \ldots, J_n \triangleright P_n$.

We will denote $\mathrm{rv}(J)$ the set of variables bound by a join-pattern J in (*i*), and $\mathrm{dv}(D)$ the set of channel names defined by a definition D in (*ii*). We will denote $\mathrm{fv}(P)$ the set of free names or variables in a process P. Similarly, $\mathrm{fv}(D)$ will denote the set of free variables in a definition D; by convention we take $\mathrm{dv}(D) \subseteq \mathrm{fv}(D)$. The inductive definition for $\mathrm{rv}(J)$, $\mathrm{dv}(D)$, $\mathrm{fv}(D)$, and $\mathrm{fv}(P)$ appears in Figure 5 page 47.

Since we have eliminated all the synchronous primitives by internal translation, our operational semantics only needs to define two operations:

(a) sending a message on a channel name
(b) triggering a definition rule whose join pattern has been fulfilled.

Operation (a) means moving the message from its sending site to its definition site. This operation is not quite as trivial as it might first appear to be, because this move might conflict with the scoping rules of the join calculus: some of the message's arguments might be locally defined channel names, as in

$$\texttt{def}\ p\langle x\rangle \triangleright P\ \texttt{in}\ (\texttt{def}\ q\langle y\rangle \triangleright Q\ \texttt{in}\ p\langle q\rangle)$$

In the lambda calculus, this problem is solved by combining the sending and triggering operations, and directly replacing $p\langle q\rangle$ by $P\{q/x\}$ in the process above. This solution does not work for the join calculus in general, however, since a rule might need several messages from several sites to trigger, as in

$$\texttt{def}\ p\langle x\rangle \mid p'\langle x'\rangle \triangleright P\ \texttt{in}\ (\texttt{def}\ q\langle y\rangle \triangleright Q\ \texttt{in}\ p\langle q\rangle) \mid (\texttt{def}\ q\langle y\rangle \triangleright Q'\ \texttt{in}\ p'\langle q\rangle)$$

The solution, which was first discovered for the pi calculus [32], lies in doing things the other way around: rather than moving the contents of the outer definition inside the inner one, we extend the scope of the argument's definition to include that of the message's channel. This operation is called a *scope extrusion*. This is a fairly complex operation, since it involves doing preliminary alpha-conversion to avoid name captures, and moving a full definition, along with all the messages associated with it.

In contrast, the trigger operation (b) means selecting a particular rule $J \triangleright P$, assembling a group M of messages that match the rule's join pattern J, and simply replacing M with an appropriate instance of P (with the arguments of M substituted for the parameters $\mathrm{rv}(J)$ of the rule).

Although it might seem much simpler, only operation (b) has real computational contents in the core join calculus (for the distributed join calculus, moving a message across sites does impact computation in subtle ways—see Section 5). Operation (a) only rearranges the order of subterms in a way that preserves all bindings. The selecting and assembling steps of operation (b) can also be viewed as similar rearrangements. Finally, we note that such rearrangements never take place in *guarded* subterms (subterms of a process P that appears on the right hand side of a definition rule $J \triangleright P$).

Thus, if we denote by $P \equiv Q$ the *structural equivalence* relation "P and Q are the same up to alpha-conversion and rearrangement of unguarded subterms that preserve bindings", then the entire operational semantics of the join calculus can be expressed by a single rule:

$$\frac{\begin{array}{c} R \equiv \texttt{def}\ J \triangleright P \wedge D\ \texttt{in}\ J\rho \mid Q \\ R' \equiv \texttt{def}\ J \triangleright P \wedge D\ \texttt{in}\ P\rho \mid Q \\ \rho \text{ maps variables in } \mathrm{rv}(J) \text{ to channel names} \end{array}}{R \longrightarrow R'}$$

where the process R is decomposed as follows: $J \triangleright P$ is the active rule, $J\rho$ are the messages being consumed, and D and Q collect all other rules and processes of R. The '$\equiv$' in the second premise could be replaced by '$=$' since it is only necessary to shift messages and definitions around before the trigger step. However, this '$\equiv$' gives us additional flexibility in writing series of reduction steps, since it allows us to keep the syntactic shape of the term by undoing the "rule selection" steps, and moving back the process $P\rho$ to the original place of one of the triggering messages.

The structural equivalence relation $\equiv$ itself is easily (but tediously) axiomatized as the least equivalence relation such that:

$$
\begin{array}{rll}
P \equiv P' & & \text{if } P \text{ and } P' \text{ are alpha-equivalent} \\
D \equiv D' & & \text{if } D \text{ and } D' \text{ are alpha-equivalent} \\
P \mid Q \equiv P' \mid Q' & & \text{if } P \equiv P' \text{ and } Q \equiv Q' \\
\texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ D'\ \texttt{in}\ P' & & \text{if } D \equiv D' \text{ and } P \equiv P' \\
D_1 \wedge D_2 \equiv D_1' \wedge D_2' & & \text{if } D_1 \equiv D_1' \text{ and } D_2 \equiv D_2' \\
P \mid \mathbf{0} \equiv P & & \\
P \mid (Q \mid R) \equiv (Q \mid P) \mid R & & \\
D \wedge \top \equiv D & & \\
D_1 \wedge (D_2 \wedge D_3) \equiv (D_2 \wedge D_1) \wedge D_3 & & \\
\texttt{def}\ \top\ \texttt{in}\ P \equiv P & & \\
(\texttt{def}\ D\ \texttt{in}\ P) \mid Q \equiv \texttt{def}\ D\ \texttt{in}\ (P \mid Q) & & \text{provided } \mathrm{dv}(D) \cap \mathrm{fv}(Q) = \emptyset \\
\texttt{def}\ D_1\ \texttt{in}\ \texttt{def}\ D_2\ \texttt{in}\ P \equiv \texttt{def}\ D_1 \wedge D_2\ \texttt{in}\ P & & \text{provided } \mathrm{dv}(D_2) \cap \mathrm{fv}(D_1) = \emptyset
\end{array}
$$

1.5 The reflexive chemical abstract machine

The operational semantics we just described may be technically sound and convenient to manipulate, but it does not quite give an intuitively satisfying account of the execution of processes. The reason is that, in order to simplify the exposition, we have lumped together and hidden in the "trivial rearrangements" equivalence '$\equiv$' a number of operations that must occur in a real implementation:

1. Programmed parallelism (the '|' operator) must be turned into runtime parallelism (multiple kernel or virtual machine threads) and, conversely, threads must terminate with either the null process **0** or with a message send.
2. The alpha-conversions required for scope extrusion are usually implemented by dynamically allocating a new data structure for the names of each local definition.
3. The selection of the definition containing the next rule to be triggered is done by thread scheduling.
4. The selection of the actual rule within the definition is done by a finite state automaton that tracks the names of messages that have arrived. This automaton also enables the scheduling of the definition in 3.
5. Messages must be routed to their definition, where they are sorted by name and queued.

All this should concern us, because there might be some other implementation issue that is hidden in the use of '$\equiv$' and that could not be resolved like the above. For instance, the asynchronous pi calculus has an equivalence-based semantics that is very similar to that of the join calculus. It has a a single reduction rule, up to unguarded contexts and '$\equiv$':

$$\overline{x}\langle\widetilde{v}\rangle \mid x(\widetilde{y}).P \longrightarrow P\{\widetilde{v}/\widetilde{y}\}$$

As we have seen in Section 1.2, despite the similarities with the join calculus, this semantics does not possess the important locality property for communications on channel x and, in fact, cannot be implemented without global synchronization. Term-based operational semantics may mask such implementation concerns, because by essence they can only describe global computation steps.

In this section, we address this issue by exhibiting a computational model for the join calculus, called the *reflexive chemical abstract machine* (RCHAM), which can be refined into an efficient implementation. Later, we will also use the RCHAM and its extensions to explicitly describe distributed computations. Our model addresses issues 1 and 2 directly, and resorts to structural (actually, denotational) equivalence for issues 3–5, which are too implementation-dependent to be described convincingly in a high-level model: issue 3 would require a model of thread scheduling, and issue 5 would require a model of the data structures used to organize threads; without 3 and 5, issue 4 is meaningless. However, we will show that the structural properties of the RCHAM ensure that issues 3–5 can be properly resolved by an actual implementation.

The state of the RCHAM tracks the various threads that execute a join calculus program. As is apparent from the discussion of 1–5, the RCHAM should contain two kinds of (virtual) threads:

- process threads that create new channels and end by sending a message; they will be represented by join calculus processes P.
- definition threads that monitor queued messages and trigger reaction rules; they will be represented by join calculus definitions D.

We do not specify the data structures used to organize those terms; instead, we just let the RCHAM state consist of a pair of multisets, one for definition threads, one for process threads.

Definition 1 (Chemical Solutions). *A chemical solution $\mathcal{S}$ is a pair $\mathcal{D} \vdash \mathcal{P}$ of a multiset $\mathcal{D} = \{\!|D_1, \ldots, D_m|\!\}$ of join calculus definitions, and a multiset $\mathcal{P} = \{\!|P_1, \ldots, P_n|\!\}$ of join calculus processes.*

The intrinsic reordering of multiset elements is the only structural equivalence we will need to deal with issues 3–5. The operational semantics of the RCHAM is defined up to this reordering. Chemical solutions and the RCHAM derive their names from the close parallel between this operational semantics and the reaction rules of molecular chemistry. This chemical metaphor, first coined in [7], can be carried out quite far:

- a message $M = x\langle v_1, \ldots, v_n\rangle$ is an *atom*, its channel name x is its *valence*.
- a parallel composition $M_1 \mid \cdots \mid M_n$ of atoms is a *simple molecule*; any other process P is a *complex molecule*.
- a definition rule $J \triangleright P$ is a *reaction rule* that specifies how a simple molecule may react and turn into a new, complex molecule. The rule actually specifies a reaction pattern, based solely on the valences of the reaction molecule. (And of course, in this virtual world, there is no conservation of mass, and P may be arbitrarily large or small.)

– the multisets composing the RCHAM state are *chemical solutions*; multiset reordering is "Brownian motion".

The RCHAM is "reflexive" because its state contains not only a solution of molecules that can interact, but also the multiset of the rules that define those interactions; furthermore, this multiset can be dynamically extended with new rules for new valences (however, the rules for a given set of valences cannot be inspected, changed, or extended).

Computation on the RCHAM is compactly specified by the six rules given in Figure 2. By convention, the rule for a computation step shows only the processes and definitions that are involved in the step; the rest of the solution, which remains unchanged, is implicit. Rule STR-DEF is a bit of an exception: its side condition formally means that $\sigma(\mathrm{dv}(D)) \cap (\mathrm{fv}(\mathcal{D}) \cup \mathrm{fv}(\mathcal{P})) = \emptyset$, where $\mathcal{D} \vdash \mathcal{P}$ is the global state of the RCHAM. (There are several ways to make this a local operation, such as address space partitioning or random number generation, and it would be ludicrous to hard code a specific implementation in the RCHAM.)

Figure 2 defines two different kinds of computation steps:

– *reduction steps* '$\rightarrow$' describe actual "chemical" interactions, and correspond to join calculus reduction steps;
– *heating steps* '$\rightharpoonup$' describe how molecules interact with the solution itself, and correspond in part to join calculus structural equivalence. Heating is always reversible; the converse '$\rightharpoondown$' steps are called *cooling steps*; the STR-rules in Figure 2 define both kinds of steps simultaneously, hence the '$\rightleftharpoons$' symbol.

There is not a direct correspondence between '$\rightleftharpoons$' and '$\equiv$': while scope extrusion is obviously linked to STR-DEF, commutativity-associativity of parallel composition is rather a consequence of the multiset reordering. Unlike structural equivalence, heating rules have a direct operational interpretation:

– $\overset{\text{STR-PAR}}{\rightharpoonup}$ and $\overset{\text{STR-NULL}}{\rightharpoonup}$ correspond to forking and ending process threads (issue 1).
– $\overset{\text{STR-DEF}}{\rightharpoonup}$ corresponds to allocating a new address for a definition (issue 2), and forking a new definition thread there.
– $\overset{\text{STR-AND}}{\rightharpoonup}$ and $\overset{\text{STR-TOP}}{\rightharpoonup}$ correspond to entering rules in the synchronization automaton of a definition (issue 4 in part).

With one exception, cooling steps do not have a similar operational interpretation; rather, they are used to tie back the RCHAM computation to the join calculus syntax. The exception is for $\overset{\text{STR-PAR}}{\rightharpoondown}$ steps that aggregate simple molecules whose names all appear in the same join pattern; there $\overset{\text{STR-PAR}}{\rightharpoondown}$ steps model the queuing and sorting performed by the synchronization automaton of the definition in which that pattern appears (again, issue 4 in part). We will denote such steps by $\overset{\text{JOIN}}{\rightharpoondown}$.

These observations, and the connection with the direct join calculus operational semantics are supported by the following theorem (where the reflexive-transitive closure of a relation $\mathcal{R}$ is writtten $\mathcal{R}^*$, as usual).

$$
\begin{array}{ll}
\textsc{Str-Null} & \vdash \mathbf{0} \rightleftharpoons \vdash \\
\textsc{Str-Par} & \vdash P \mid P' \rightleftharpoons \vdash P,\ P' \\
\textsc{Str-Top} & \top \vdash \rightleftharpoons \vdash \\
\textsc{Str-And} & D \wedge D' \vdash \rightleftharpoons D,\ D' \vdash \\
\textsc{Str-Def} & \vdash \mathtt{def}\ D\ \mathtt{in}\ P \rightleftharpoons D\sigma \vdash P\sigma \\
 & \\
\textsc{React} & J \triangleright P \vdash J\rho \rightarrow J \triangleright P \vdash P\rho
\end{array}
$$

Side conditions:
in Str-Def, σ substitutes distinct fresh names for the defined names $\mathrm{dv}(D)$;
in React, ρ substitutes names for the formal parameters $\mathrm{rv}(J)$.

Figure 2. The reflexive chemical abstract machine (RCHAM)

Theorem 2. *Let P, P' and $\mathcal{S}, \mathcal{S}'$ be join calculus processes and* RCHAM *solutions, respectively. Then*

1. $P \equiv P'$ *if and only if* $\vdash P \rightleftharpoons^* \vdash P'$
2. $P \rightarrow P'$ *if and only if* $\vdash P \rightleftharpoons^* \rightarrow \rightleftharpoons^* \vdash P'$
3. $\mathcal{S} \rightleftharpoons^* \mathcal{S}'$ *if and only if* $\mathcal{S} \multimap^* \rightharpoonup^* \mathcal{S}'$
4. $\mathcal{S} \rightleftharpoons^* \rightarrow \rightleftharpoons^* \mathcal{S}'$ *if and only if* $\mathcal{S} \multimap^* \overset{\textsc{Join}}{\rightharpoonup}{}^* \rightarrow \rightharpoonup^* \mathcal{S}'$

Corollary 3. *If $P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_n$ is a join calculus computation sequence, then there exist chemical solutions $\mathcal{S}_1, \ldots, \mathcal{S}_n$ such that*

$$\vdash P_0 \ \multimap^* \overset{\textsc{Join}}{\rightharpoonup}{}^* \rightarrow \ \mathcal{S}_1 \ \multimap^* \overset{\textsc{Join}}{\rightharpoonup}{}^* \rightarrow \ \cdots \ \multimap^* \overset{\textsc{Join}}{\rightharpoonup}{}^* \rightarrow \ \mathcal{S}_n$$

with $\mathcal{S}_i \rightharpoonup^* \vdash P_i$ for $1 \leq i \leq n$.

We still have to examine issues 3–5, which we have deliberately abstracted. The RCHAM, and chemical machines in general, assumes that the atoms that interact are brought together by random motion. This is fine for theory, but a real implementation cannot be based only on chance. In our case, by Theorem 2, an RCHAM computation relies on this "magical mixing" only for $\overset{\textsc{Join}}{\rightharpoonup}$ and $\overset{\textsc{React}}{\rightarrow}$ steps. In both cases, we can show that no magic needs to be invoked:

- $\overset{\textsc{Join}}{\rightharpoonup}$ steps only brings together atoms that have been routed to the definition of their valence, and then only if these valences match one of the finite number of join patterns of that definition.
- $\overset{\textsc{React}}{\rightarrow}$ simply selects one matching in the finite set of completed matches that have been assembled at that definition.

Because synchronization decisions for a definition are always based on a finite fixed set of valences, they can be compiled into a finite state automaton; this is the compilation approach used in JoCaml—see [28] for an extensive discussion of this implementation issue.

To keep up with the chemical metaphor, definitions are very much like the *enzymes* of biochemistry: they enable reactions in a sparse solution, by providing

a fixed reaction site on which reactants can assemble. It is interesting to note that the chemical machine for the pi calculus, which is in many aspects very similar to the RCHAM, fails the locality property on exactly this count: it relies solely on multiset reordering to assemble senders and receivers on a channel.

2 Basic equivalences

So far, the join calculus is a calculus only by name. We have argued that it is a faithful and elegant *model* for concurrent programming. A true *calculus* implies the ability to do equational reasoning—i.e., to calculate. To achieve this we must equip the join calculus with a proper notion of "equality"; since join calculus expressions model concurrent programs, this "equality" will be a form of *program equivalence*. Unfortunately, finding the "right" equivalence for concurrent programs is a tall order:

1. Two programs P and Q should be equivalent only when they have exactly the same properties; in particular it should always be possible to replace one with the other in a system, without affecting the system's behavior in any visible way. This is a prerequisite, for instance, to justify the optimizations and other program transformations performed by a compiler.
2. Conversely, if P and Q are *not* equivalent, it should always be possible to exhibit a system for which replacing P by Q results in a perceivable change of behavior.
3. The equivalence should be technically tractable; at least, we need effective proof techniques for establishing identities, and these techniques should also be reasonably complete.
4. The equivalence should generate a rich set of identities, otherwise there won't be much to "calculate" with. We would at least expect some identities that account for asynchrony, as well as some counterpart of the lambda calculus beta equivalence.

Unfortunately, these goals are contradictory. Taken literally, goal 1 would all but force us to use syntactic equality, which certainly fails goal 4. We must thus compromise. We give top priority to goal 4, because it wouldn't make much sense to have a "calculus" without equations. Goal 3 is our next priority, because a calculus which requires an inordinate amount of efforts to establish basic identities wouldn't be too useful. Furthermore, few interesting program equivalences can be proven from equational reasoning alone, even with a reasonable set of identities; so it is important that reasoning from first principles be tractable.

Goal 1 thus only comes in third position, which means that we must compromise on soundness. Thus we will consider equivalences that do not preserve all observable properties, i.e., that only work at a certain level of *abstraction*. Common examples of such abstractions, for sequential programs, are observing runtime only up to a constant factor ($O(\cdot)$ complexity), not observing runtime at all (only termination), or even observing only return values (partial correctness). Obviously, we must very carefully spell out the set of properties that are

preserved, for it determines how the equations of calculus (and to some extent, the calculus itself) can be used.

Even though it comes last on our priority list, we should not neglect goal 2. Because of the abstractions it involves, a correctness proof based on a formal model provides only a relative guarantee. Hence such a proof is often more useful when if *fails*, because this allows one to *find errors* in the system. Goal 2 is to make sure that all such failures can be traced to actual errors, and not to some odd technical property of the equivalence that is used. The reason goal 2 comes last is that the proof process has to be tractable in the first place for this "debugging" approach to make any practical sense.[3]

The priorities we have set are not absolute, and therefore it makes sense to consider several equivalences, that strike different compromises between goals 1–4. We will study two of them in subsections 2.1 and 2.4: *may testing*, which optimizes goals 4 and 2, at the expense of goals 1 and 3, and *bisimilarity equivalence* (which we will often simplify to *bisimilarity* in the following), which sacrifices goal 2 to get a better balance between the other goals. May testing is in fact coarser than bisimilarity, so that in practice one can attempt to prove a bisimilarity, and in case of failure "degrade" to may testing to try to get a good counterexample. In sections 3 and 4 we will study three other equivalences; the complete hierarchy is summarized in Figure 3 on page 45.

2.1 May testing equivalence

The first equivalence we consider follows rigidly from the priorities we set above: it is the coarsest equivalence that reasonably preserves observations, and it also turns out to have reasonable proof techniques, which we will present in subsections 2.2 and 2.3. Moreover, may testing fulfills goal 2: when may testing equivalence fails, one can always exhibit a finite *counter-example trace*.

May testing is the equivalence that preserves all the properties that can be described in terms of a finite number of interactions (message exchanges in the case of the join calculus), otherwise known as *safety properties*. This restriction may seem severe, but it follows rather directly from the choice of the syntax and semantics of the join calculus. As we noted in Section 1.5, we deliberately chose to abstract away the scheduling algorithm. In the absence of any hypothesis on scheduling, it is in practice impossible to decide most complexity, termination, or even progress properties. Moreover, such *liveness* properties are just abstractions of the *timing* properties which the system really needs to satisfy; and timing properties can be obtained fairly reliably by measuring the actual implementation. So, it makes practical sense for now to exclude liveness properties; but we will reconsider this decision in sections 2.4 and 3.2, using abstract scheduling assumptions.

The "testing" in may testing refers to the way safety properties are characterized in the formal definition of the may testing equivalence. We represent a

[3] Also, a clear successful proof will identify the features that actually make the system work, and often point out simplifications that could be made safely.

safety property $\mathcal{P}$ by a *test* $T(\cdot)$, such that a process P has property $\mathcal{P}$ if and only if $T(P)$ *succeeds*. Hence, P and Q will be equivalent if, for any test T, $T(P)$ succeeds iff $T(Q)$ succeeds.

To make all this formal, we need to fix a formal syntax for tests, and a formal semantics for "succeeds". We restricted ourselves to properties that can be described in terms of finite message exchanges, which can certainly be carried out by join calculus programs, or rather *contexts*. Since the join calculus contains the lambda calculus, we can boldly invoke Church's thesis and assume that a test $T(\cdot)$ can always be represented by a pair $(C[\cdot], x)$ of a join calculus *evaluation context*, and channel name x, which $C[\cdot]$ uses to signal success: $T(P)$ succeeds iff $C[P]$ *may* send an $x\langle\ldots\rangle$ message.

Evaluation contexts Evaluation contexts are simply join calculus processes with a "hole" $[\cdot]_S$ that is not inside a definition (not *guarded*)—they are called static contexts in [30]. They are defined by the following grammar:

$C[\cdot]_S ::=$		evaluation contexts
	$[\cdot]_S$	hole
$\|$	$P \mid C[\cdot]_S$	left parallel composition
$\|$	$C[\cdot]_S \mid P$	right parallel composition
$\|$	$\texttt{def}\ D\ \texttt{in}\ C[\cdot]_S$	process/function definition

Formally, a context and its hole are *sorted* with a set S of *captured names*. If P is any process, $C[P]_S$ is the process obtained by replacing the hole in $C[\cdot]_S$ by P, after alpha converting bound names *not in* S of $C[\cdot]_S$ that clash with $\mathrm{fv}(P)$. Provided S contains enough channel names, for example all the channel names bound in $C[\cdot]_S$, or all the channel names free in P, this means replacing $[\cdot]_S$ by P without any conversion whatsoever. We will often drop the S subscript in this common case.

The structural equivalence and reduction relations are extended to contexts $C[\cdot]_S$ of the same sort S, with the proviso that alpha conversion of names in S is not allowed. The substitution relation is similarly extended to contexts: $C[C'[\cdot]_{S'}]_S$ is a context of type S'.

By convention, we only allow ourselves to write a reduction $C[P]_S \to C'[P']_{S'}$ when the identity of the hole is preserved—that is, when in fact we have

$$C[P \mid [\cdot]_{S\cap S'}]_S \to C'[P' \mid [\cdot]_{S\cap S'}]_{S'}$$

Similarly, $C[P] \to C'[P']$ really means that $C[P]_S \to C'[P']_{S'}$ for some suitably large sets S and S'.

Evaluation contexts are a special case of *program contexts* $P[\cdot]$, which are simply process terms with a (possibly guarded) hole. An equivalence relation $\mathcal{R}$ such that $Q\ \mathcal{R}\ Q'$ implies $P[Q]\ \mathcal{R}\ P[Q']$ for any program context $P[\cdot]$ is called a *congruence*; if $\mathcal{R}$ is only a preorder then it is called a *precongruence*. A congruence can be used to describe equations between subprograms. All of our equivalences are congruences for evaluation contexts, which means they can describe equations between subsystems, and most are in fact congruences (the exception only occurs for an extension of the join calculus with name testing).

Output observation The success of a test is signaled by the output of a specific message; this event can be defined syntactically.

Definition 4 (Output predicates). *Let P be a join calculus process and x be a channel name.*

- $P \downarrow_x$ *iff* $P = C[x\langle\widetilde{v}\rangle]_S$ *for some tuple* $\widetilde{v}$ *of names and some evaluation context* $C[\cdot]_S$ *that does not capture* x*—that is, such that* $x \notin S$.
- $P \Downarrow_x$ *iff* $P \to^* P'$ *for some* P' *such that* $P' \downarrow_x$.

The predicate $\downarrow_x$ tests syntactically for immediate output on x, and is called the strong barb *on x. The* (weak) barb *on x predicate $\Downarrow_x$ only tests for the possibility of output.*

The strong barb can also be defined using structural equivalence, as $P \downarrow_x$ if and only if $P \equiv \texttt{def}\ D\ \texttt{in}\ Q \mid x\langle\widetilde{v}\rangle$ for some $D, Q, \widetilde{v}$ such that $x \notin \mathrm{dv}(D)$.

We say that a relation $\mathcal{R}$ *preserves barbs* when, for all x, $P\ \mathcal{R}\ Q$ and $P \Downarrow_x$ implies $Q \Downarrow_x$. By the above, structural equivalence preserves barbs.

May testing With the preceding definitions, we can easily define may testing:

Definition 5 (May testing preorder and equivalence). *Let P, Q be join calculus processes, and let $C[\cdot]_S$ range over evaluation contexts.*

- $P \sqsubseteq_{\mathrm{may}} Q$ *when, for any* $C[\cdot]_S$ *and* x, $C[P]_S \Downarrow_x$ *implies* $C[Q]_S \Downarrow_x$.
- $P \simeq_{\mathrm{may}} Q$ *when, for any* $C[\cdot]_S$ *and* x, $C[P]_S \Downarrow_x$ *iff* $C[Q]_S \Downarrow_x$.

The preorder $\sqsubseteq_{\mathrm{may}}$ is called the may testing preorder, and the equivalence $\simeq_{\mathrm{may}}$ is called the may testing equivalence.

The may testing preorder $P \sqsubseteq_{\mathrm{may}} Q$ indicates that P's behaviors are included in those of Q, and can thus be used to formalize "P implements Q". Clearly, the equivalence $\simeq_{\mathrm{may}} = \sqsubseteq_{\mathrm{may}} \cap \sqsupseteq_{\mathrm{may}}$ is the largest symmetric subrelation of $\sqsubseteq_{\mathrm{may}}$.

As an easy example of may testing, we note that structural equivalence is finer than may testing equivalence: $P \equiv Q$ implies $C[P]_S \equiv C[Q]_S$, and $\equiv$ preserves barbs. By the same token, we note that we can drop the S sorts in the definition of $\simeq_{\mathrm{may}}$: $C[\cdot]_S, x$, $C'[\cdot]_S, x$ test the same property if $C[\cdot]_S \equiv C'[\cdot]_S$, so it is enough to consider contexts where no alpha conversion is needed to substitute P or Q.

As a first non-trivial example, let us show a simple case of beta conversion:

$$\texttt{def}\ x\langle y\rangle \triangleright P\ \texttt{in}\ C[x\langle u\rangle]_u \simeq_{\mathrm{may}} \texttt{def}\ x\langle y\rangle \triangleright P\ \texttt{in}\ C[P\{^u/_y\}]_u$$

For any test $C'[\cdot]_S, t$ we let $S' = S \cup \{x, u\}$ and define

$$C''[\cdot]_{S'} \stackrel{\mathrm{def}}{=} C'[\texttt{def}\ x\langle y\rangle \triangleright P\ \texttt{in}\ C[[\cdot]_{S'}]_u]_S$$

then we show that $C''[x\langle u\rangle]_{S'} \Downarrow_t$ iff $C''[P\{^u/_y\}]_{S'} \Downarrow_t$. The "if" is obvious, since $C''[x\langle u\rangle]_{S'} \to C''[P\{^u/_y\}]_{S'}$. (More generally, we have $\to \subset \sqsupseteq_{\mathrm{may}}$.) For the converse, suppose that $C''[x\langle u\rangle]_{S'} \to^* Q \downarrow_t$. If the reduction does not involve

the $x\langle u\rangle$ message, then $C''[R]_{S'} \Downarrow_t$ for any R—including $R = P\{^u/_y\}$. Otherwise, the definition that uses $x\langle u\rangle$ can only be $x\langle y\rangle \triangleright P$, hence we have

$$C''[x\langle u\rangle]_{S'} \to^* C'''[x\langle u\rangle]_{S'} \to C'''[P\{^u/_y\}]_{S'} \to^* Q \downarrow_t$$

whence we also have $C''[P\{^u/_y\}] \to^* C'''[P\{^u/_y\}]_{S'} \Downarrow_t$.

The same proof can be carried out if $x\langle u\rangle$ appears in a general context $Q[x\langle u\rangle]$, using the notion of general context reduction from Section 2.3. This gives *strong* beta equivalence, for we have

$$\begin{aligned}
& \texttt{let } f(x) = E \texttt{ in } Q[\texttt{let } z = f(u) \texttt{ in } R] \\
\stackrel{\text{def}}{=}\ & \texttt{let } f(x) = E \texttt{ in } Q[\texttt{def } \kappa\langle z\rangle \triangleright R \texttt{ in } f\langle u, \kappa\rangle] \\
\simeq_{\text{may}}\ & \texttt{let } f(x) = E \texttt{ in } Q[\texttt{def } \kappa\langle z\rangle \triangleright R \texttt{ in let } v = E\{^u/_x\} \texttt{ in } \kappa\langle v\rangle] \\
\simeq_{\text{may}}\ & \texttt{let } f(x) = E \texttt{ in } Q[\texttt{def } \kappa\langle z\rangle \triangleright R \texttt{ in let } v = E\{^u/_x\} \texttt{ in } R\{^v/_z\}] \\
\equiv\ & \texttt{let } f(x) = E \texttt{ in } Q[(\texttt{def } \kappa\langle z\rangle \triangleright R \texttt{ in } \mathbf{0}) \mid \texttt{let } z = E\{^u/_x\} \texttt{ in } R] \\
\simeq_{\text{may}}\ & \texttt{let } f(x) = E \texttt{ in } Q[\texttt{let } z = E\{^u/_x\} \texttt{ in } R]
\end{aligned}$$

since obviously $\texttt{def } D \texttt{ in } \mathbf{0} \simeq_{\text{may}} \mathbf{0}$ for any D.

2.2 Trace observation

We defined may testing in terms of evaluation contexts and barbs. This allowed us to stay as close as possible to our intuitive description, and exhibit compliance with goals 4 and 1. However, this style of definition does not really suit our other two goals.

In general, proving that $P \sqsubseteq_{\text{may}} Q$ can be quite intricate, because it involves reasoning about $C[P]_S \Downarrow_x$—an arbitrary long reduction in an arbitrarily large context. This double quantification may not be much of an issue for showing simple, general reduction laws such as the above, but it can quickly become unmanageable if one wants to show, say, the correctness of a given finite-state protocol. Moreover it all but precludes the use of automated model-checking tools.

If $P \sqsubseteq_{\text{may}} Q$ fails, then there must be a test $C[\cdot]_S, x$ that witnesses this failure. However the existence of this $C[\cdot]_S, x$ only partly fulfills goal 2, because it illustrates the problem with the programs P and Q only indirectly, by the means of a third, unrelated program. Moreover, the definition of $\sqsubseteq_{\text{may}}$ gives no clue for deriving this witness from a failing proof attempt.

In this section we present an alternative characterization of $\sqsubseteq_{\text{may}}$ and $\simeq_{\text{may}}$ that mitigates the proof problem, and largely solves the counter-example problem. This characterization formalizes the fact, stated in Section 2.1, that $\simeq_{\text{may}}$ preserves properties based on *finite* interaction sequences. We formalize such sequences as *traces*, define the *trace-set* of a process P, and then show that $\sqsubseteq_{\text{may}}$ corresponds to the inclusion of trace-sets, and $\simeq_{\text{may}}$ to their equality.

Informally, a trace is simply an input/output sequence; however we must account for the higher-order nature of the join calculus, and this makes the definition more involved.

Definition 6 (Traces). *A trace T is a finite sequence $J_0 \triangleright M_0, \ldots, J_n \triangleright M_n$ such that*

1. *For each element $J_i \triangleright M_i$ of the trace, M_i is a single message $x_i\langle y_{i1}, \ldots, y_{il_i}\rangle$, and J_i is either $\mathbf{0}$ or a join pattern.*
2. *The variables y_{ik} and in $\mathrm{rv}(J_j)$ are all pairwise distinct.*
3. *Each channel name defined by J_j is a y_{ik} for some $i < j$ (hence, $J_0 = \mathbf{0}$).*
4. *Dually, $x_i \neq y_{jk}$ for any j, k, and $x_i \notin \mathrm{rv}(J_j)$ for any $j > i$.*

We set $T_I(i) = \{y_{jk} \mid 0 \leq j < i, 1 \leq k \leq l_j\}$, so that condition 3 can be stated as $\mathrm{dv}(J_i) \subseteq T_I(i)$.

Intuitively, to each element $J_i \triangleright M_i$ of a trace should correspond a reduction that behaves (roughly) as the join definition $J_i \triangleright M_i$: that is, the reduction should input J_i and output M_i. The names in the $T_I(i)$ and the $\mathrm{rv}(J_i)$ are formally bound in the trace. In matching a process run, the $\mathrm{rv}(J_i)$ names should be substituted by fresh names, and the y_{jk} by the arguments of the actual outputs.

Definition 7 (Trace sets). *A trace $T = J_0 \triangleright M_0, \ldots, J_n \triangleright M_n$ is allowed by a process P (notation $P \models T$), if there is a substitution σ that maps names in the $\mathrm{rv}(J_i)$ to distinct names not in $\mathrm{fv}(P)$, and names in the $T_I(i)$ to channel names, and a sequence $C_i[\cdot]_{S_i}$ of evaluation contexts of sort $S_i = \sigma(T_I(i))$, such that $P = C_0[\mathbf{0}]_\emptyset$ and*

$$C_i[J_i\sigma]_{S_i} \rightarrow^* C_{i+1}[M_i\sigma]_{S_{i+1}}$$

for each i, $0 \leq i \leq n$. Note that according to the convention we set in 2.1, this notation implies that the identity of the hole is preserved during the reduction.

The trace-set $\mathrm{Tr}(P)$ of P is the set of all its allowed traces.

$$\mathrm{Tr}(P) = \{T \mid P \models T\}$$

The main motivation for this rather complex set of definitions is the following result

Theorem 8 (Trace equivalence). *For any two processes P and Q, we have $P \sqsubseteq_{\mathrm{may}} Q$ if and only if $\mathrm{Tr}(P) \subseteq \mathrm{Tr}(Q)$, and consequently $P \simeq_{\mathrm{may}} Q$ if and only if $\mathrm{Tr}(P) = \mathrm{Tr}(Q)$.*

This theorem allows us to reach goal 2: if $P \sqsubseteq_{\mathrm{may}} Q$ fails, then there must be some finite input/output trace that is allowed by P but that is barred by Q. In principle, we can look for this trace by enumerating all the traces of P and Q. This search can also be turned into a proof technique for $P \sqsubseteq_{\mathrm{may}} Q$ that does not involve trying out arbitrary large contexts, and may be more amenable to model-checking. Note, however, that the technique still involves arbitrary long traces, and in that sense fails to meet goal 3.

As an application, let us prove the correctness of the compilation of a three-way join into two two-way joins:

$$\begin{aligned} & \texttt{def}\ x\langle\rangle \mid y\langle\rangle \triangleright t\langle\rangle \wedge t\langle\rangle \mid z\langle\rangle \triangleright u\langle\rangle\ \texttt{in}\ v\langle x, y, z\rangle \\ \simeq_{\mathrm{may}}\ & \texttt{def}\ x\langle\rangle \mid y\langle\rangle \mid z\langle\rangle \triangleright u\langle\rangle\ \texttt{in}\ v\langle x, y, z\rangle \end{aligned}$$

For either terms, a trace must start with $\mathbf{0} \triangleright v\langle x, y, z\rangle$, and thereafter consist of outputs of $u\langle\rangle$ after inputs of $x\langle\rangle$, $y\langle\rangle$, $z\langle\rangle$. In either case, there must be at least n $x\langle\rangle$s, $y\langle\rangle$s, and $z\langle\rangle$s in $J_1, \ldots, J_n$, so both terms have exactly the same set of traces.

2.3 Simulation and coinduction

While in many respects an improvement over the basic definition of $\simeq_{\text{may}}$, the trace sets do not really help with the real difficulty of $\simeq_{\text{may}}$ proofs—the quantification over arbitrary long reduction sequences that is hidden in the definition of barbs $\Downarrow_x$. Because the shape of a term changes during a reduction step, it is very hard to reason on the effect of several successive reduction steps. The kind of offhand argument we used for our first examples often turn out to be incorrect in more complex cases.

To do a rigorous equivalence proof, it is necessary to analyze reduction steps one case at a time. In many calculi this analysis is based on structural induction on the syntax, but in the join calculus this would be inappropriate because of structural equivalence. Hence we base our analysis on the triggered rule and the set of messages that match its pattern, since those are invariant under structural equivalence.

In this section we will present yet another characterization of may testing, which this time will be appropriate for step-by-step analysis. To avoid the complexity of higher-order traces, we will revert to modeling interaction with arbitrary evaluation contexts; those barely add to the overall complexity of a proof because interaction with them can be highly stylized, as Theorem 8 shows.

In order to formulate our new characterization, we turn to the technique of *coinductive definition*, which has been the power horse of most concurrent program equivalences. This simply means that we invoke Tarski's theorem to define an equivalence $\mathcal{E}$ as the greatest fixed point of a monotonic functional $\mathcal{F}$. The beauty of this definition is that it immediately gives us a generic method for proving $P\ \mathcal{E}\ Q$: $P\ \mathcal{E}\ Q$ iff we can find some *subfixpoint* relation $\mathcal{R}$ such that $P\ \mathcal{R}\ Q$, and that for any P', Q', $P'\ \mathcal{R}\ Q'$ implies $P'\ \mathcal{F}(\mathcal{R})\ Q'$.

To get a monotonic $\mathcal{F}$, we rely on definitions of the form "$P\ \mathcal{F}(\mathcal{R})\ Q$ iff $\mathcal{P}(P, Q, \mathcal{R})$" where $\mathcal{R}$ appears only in positive $P'\ \mathcal{R}\ Q'$ subformulas in $\mathcal{P}(P, Q, \mathcal{R})$. In this case a relation $\mathcal{R}$ is a subfixpoint of $\mathcal{F}$ iff $P\ \mathcal{R}\ Q$ implies $\mathcal{P}(P, Q, \mathcal{R})$, a property which we will denote by $\mathcal{P}^*(\mathcal{R})$. As $\cdot^*$ distributes over conjunction, we will generally define our equivalences by a conjunction of such properties: "$\mathcal{E}$ is the coarsest relation such that $\mathcal{P}_1^*(\mathcal{E})$ and ... and $\mathcal{P}_n^*(\mathcal{E})$". In fact, we have already encountered such $\mathcal{P}_i^*$s:

1. Barb preservation: "if $P\ \mathcal{R}\ Q$ then for any x, $P \Downarrow_x$ implies $Q \Downarrow_x$".
2. Symmetry: "if $P\ \mathcal{R}\ Q$ then $Q\ \mathcal{R}\ P$".
3. Precongruence for evaluation contexts:
   "if $P\ \mathcal{R}\ Q$ then for any $C[\cdot]_S$, $C[P]_S\ \mathcal{R}\ C[Q]_S$".

From this observation, we get our first coinductive definition of $\simeq_{\text{may}}$: it is the greatest symmetric relation that preserves barbs and is a congruence for evaluation contexts. Note that property 1 really means that $\mathcal{R}$ is contained in a fixed

relation, in this case the *barb inclusion* relation $\sqsubseteq_{\Downarrow}$, defined by "$P \sqsubseteq_{\Downarrow} Q$ iff for any x $P \Downarrow_x$ implies $Q \Downarrow_x$".

This first characterization of $\simeq_{\text{may}}$ merely rephrases Definition 5. To improve on this, it will prove extremely convenient to introduce diagrammatic notation. To describe a property $\mathcal{P}^*$, we lay out the relations occurring in $\mathcal{P}^*$ in a two-dimensional diagram. We indicate negatively occurring, universally-quantified relations by solid lines, and positively occurring, existentially quantified relations by dashed lines. For instance, property 3 is expressed by the diagram

$$\begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ C_S[P] & \overset{\mathcal{R}}{\cdots\cdots} & C_S[Q] \end{array}$$

The main property that we are interested in is commutation with the reduction relation $\rightarrow^*$, which is called *simulation*.

Definition 9 (Simulation). *A relation $\mathcal{R}$ between join calculus terms is a simulation if for any P, P', Q, if $P \rightarrow^* P'$ and $P \mathcal{R} Q$, there is a term Q' such that $Q \rightarrow^* Q'$ and $P' \mathcal{R} Q'$. In a diagram:*

$$\begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \Big\downarrow{\scriptstyle *} & & \Big\vdots{\scriptstyle *} \\ P' & \overset{\mathcal{R}}{\cdots\cdots} & Q' \end{array}$$

What we call here simulation is often called *weak simulation* in the literature, a simulation being a relation that commutes with single-step reductions. Counting steps makes no sense in the abstract, asynchronous setting of the join calculus, so we simply drop the "weak" adjective in the rest of these notes.

Let us say that a relation $\mathcal{R}$ preserves *immediate* barbs if $P \mathcal{R} Q$ and $P \downarrow_x$ implies $Q \Downarrow_x$. We can now use the simulation property to replace barb preservation by immediate barb preservation in the coinductive characterization of $\sqsubseteq_{\text{may}}$.

Theorem 10. *May testing preorder is a simulation, hence it is also the greatest simulation that is an evaluation context precongruence and that preserves immediate barbs.*

This is an improvement, since to consider immediate barbs it is not necessary to consider reduction steps. It would appear that we have only pushed the problem over to the simulation property, but this is not the case, as by a simple tiling argument we have

Theorem 11. *A relation $\mathcal{R}$ is a simulation iff*

$$\begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \Big\downarrow & & \Big\vdots{\scriptstyle *} \\ P' & \overset{\equiv\mathcal{R}}{\cdots\cdots} & Q' \end{array}$$

We consider a single step, rather than a series of steps on the left. The '$\equiv$' allows us to study reductions only up to structural equivalence. To illustrate the power of this new characterization, let us show that

Theorem 12. *May testing preorder is a precongruence, and may-testing equivalence is a (full) congruence.*

We begin with the following lemma

Lemma 13. *For any (well-typed) P, x, y, and any tuple $\widetilde{v}$ of distinct variables that matches the arity of both x and y, we have:*

$$\texttt{def}\ x\langle\widetilde{v}\rangle \triangleright y\langle\widetilde{v}\rangle\ \texttt{in}\ P \simeq_{\text{may}} P\{{}^{y}/_{x}\}$$

Hence, both $\simeq_{\text{may}}$ and $\sqsubseteq_{\text{may}}$ are closed under substitution.

The conclusion of lemma 13 also holds without the arity assumption, but with a more involved proof. Here, we just take a candidate relation $\mathcal{S}$ consisting of all pairs of processes structurally equivalent to $C[\texttt{def}\ x\langle\widetilde{v}\rangle \triangleright y\langle\widetilde{v}\rangle\ \texttt{in}\ P]_S$ or $C[P\{{}^{y}/_{x}\}]_S$, for some $P, x, y, \widetilde{v}$ satisfying the hypotheses of the lemma. Now $\mathcal{S}$ is obviously a congruence for evaluation contexts. It trivially preserves strong barbs $\downarrow_z$ in $C[\cdot]_S$ or even in P if $z \neq x$, and if $P\{{}^{y}/_{x}\} \downarrow_y$ because $P \downarrow_x$, then $(\texttt{def}\ x\langle\widetilde{v}\rangle \triangleright y\langle\widetilde{v}\rangle\ \texttt{in}\ P) \Downarrow_y$.

To show that $\mathcal{S}$ is a simulation, consider a reduction $C[\texttt{def}\ x\langle\widetilde{v}\rangle \triangleright y\langle\widetilde{v}\rangle\ \texttt{in}\ P]_S \to Q$; we must have $Q \equiv C'[\texttt{def}\ x\langle\widetilde{v}\rangle \triangleright y\langle\widetilde{v}\rangle\ \texttt{in}\ P']_{S'}$. If the rule used is $x\langle\widetilde{v}\rangle \triangleright y\langle\widetilde{v}\rangle$, then $C'[P'\{{}^{y}/_{x}\}]_{S'} = C[P\{{}^{y}/_{x}\}]_S$, else $C[P\{{}^{y}/_{x}\}]_S \to C'[P'\{{}^{y}/_{x}\}]_{S'}$. Conversely, if $C[P\{{}^{y}/_{x}\}]_S \to Q$, and we are not in the first case above, then it can only be because the rule used matches some ys that have replaced xs. But then the $x\langle\widetilde{v}\rangle \triangleright y\langle\widetilde{v}\rangle$ can be used to perform these replacements, and bring us back to the first case. Thus $\mathcal{S} \subseteq \simeq_{\text{may}}$, from which we deduce lemma 13.

To establish theorem 12, we need a careful definition of a multi-holed general context. A general context $P[\cdot]_S$ of sort S is a term which may contain several holes $[\cdot]_\sigma$, where σ is a substitution with domain S; different holes may have different σs. Bindings, alpha conversion, structural equivalence, and reduction are extended to general contexts, by taking $\text{fv}([\cdot]_\sigma) = \sigma(S)$. The term $P[Q]_S$ is obtained by replacing every hole $[\cdot]_\sigma$ in $P[\cdot]_S$ by $Q\sigma$, after alpha converting bound names in $P[\cdot]_S$ to avoid capturing names in $\text{fv}(Q) \setminus S$.

Consider the candidate relation

$$\mathcal{R} \stackrel{\text{def}}{=} \{(P[Q]_S, R) \mid Q \sqsubseteq_{\text{may}} Q' \text{ and } P[Q']_S \sqsubseteq_{\text{may}} R\}$$

$\mathcal{R}$ is trivially closed under evaluation contexts. Let $P'[\cdot]_S$ be obtained by replacing all unguarded holes $[\cdot]_\sigma$ in $P[\cdot]_S$ with $Q\sigma$, and similarly $P''[\cdot]_S$ be obtained by replacing $[\cdot]_\sigma$ with $Q'\sigma$. Then $P'[Q']_S \sqsubseteq_{\text{may}} P''[Q']_S = P[Q']_S$ by several applications of Lemma 13, hence $P'[Q']_S \sqsubseteq_{\text{may}} R$.

If $P[Q]_S = P'[Q]_S \downarrow_x$, the $x\langle\cdots\rangle$ is in $P'[\cdot]_S$, so $P'[Q']_S \downarrow_x$, hence $R \Downarrow_x$. Similarly, any reduction step in $P'[Q]_S$ must actually take place in $P'[\cdot]_S$, i.e., it must be a $P'[Q]_S \to P'''[Q]_S$ step with $P'[\cdot]_S \to P'''[\cdot]_S$. Thus we have $P'[Q']_S \to P'''[Q']_S$, hence $R \to^* R'$ for some R' such that $P'''[Q']_S \sqsubseteq_{\text{may}} R'$, hence such that $P'''[Q]_S\ \mathcal{R}\ R'$. So $\mathcal{R}$ is also a simulation, hence $\mathcal{R} \subset \sqsubseteq_{\text{may}}$.

2.4 Bisimilarity equivalence

Despite our success with the proofs of Lemma 13 and Theorem 12, in general the coinductive approach will not always allow us to avoid reasoning about arbitrarily long traces. This line of reasoning has only been hidden under the asymmetry of the simulation condition. This condition allows us to prove that $P \sqsubseteq_{\text{may}} Q$ with the candidate relation $\mathcal{R} = \{(P', Q) \mid P \to^* P'\}$, which is a simulation iff $P \sqsubseteq_{\text{may}} Q$. But of course, proving that $\mathcal{R}$ is a simulation is no easier than proving that $P \sqsubseteq_{\text{may}} Q$—it requires reasoning at once about all sequence $P \to^* P'$. So to really attain goal 3 we need to use a different equivalence.

There is, however, a more fundamental reason to be dissatisfied with $\simeq_{\text{may}}$: it only passes goal 1 for a very restrictive notion of soundness, by ignoring any sort of liveness properties. Thus it can label as "correct" programs that are grossly erroneous. For example, one can define in the join calculus an "internal choice" process $\bigoplus_{i\in I} P_i$ between different processes P_i, by

$$\texttt{def}\ \bigwedge_{i\in I}(\tau\langle\rangle \triangleright P_i)\ \texttt{in}\ \tau\langle\rangle$$

Let us also write $P_1 \oplus P_2$ for $\bigoplus_{i=1}^{2} P_i$. Then we have the following for any P:

$$P \oplus \mathbf{0} \simeq_{\text{may}} P$$

This equation states that a program that randomly decides whether to work at all or not is equivalent to one that always works! In this example, the "error" is obvious; however, may testing similarly ignores a large class of quite subtle and malign errors called *deadlock* errors. Deadlocks occur when a subset of components of a system stop working altogether, because each is waiting for input from another before proceeding. This type of error is rather easy to commit, hard to detect by testing, and often has catastrophic consequences.

For nondeterministic sequential system, this problem is dealt with by complementing may testing with a *must* testing, which adds a "must" predicate to the barbs:

$$P \,\square\!\downarrow_x \stackrel{\text{def}}{=} \text{if } P \to^* P' \not\to \text{, then } P' \downarrow_x$$

However, must testing is not very interesting for asynchronous concurrent computations, because it confuses all diverging behaviors. We refer to [26] for a detailed study of may and must testing in the join calculus.

It turns out that there is a technical solution to both of these problems, if one is willing to compromise on goal 2: simply require symmetry and simulation together.

Definition 14. *A* bisimulation *is a simulation $\mathcal{R}$ whose converse $\mathcal{R}^{-1}$ is also a simulation.*

The coarsest bisimulation that respects (immediate) barbs is denoted $\dot{\approx}$.

The coarsest bisimulation that respects (immediate) barbs and is also a congruence for evaluation context is called bisimilarity equivalence*, and denoted $\approx$.*

When no confusion arises, we will simply refer to $\approx$ as "bisimilarity". The definition of bisimilarity avoids dummy simulation candidates : since $\mathcal{R}^{-1}$ is also a simulation, $P\ \mathcal{R}\ Q$ implies that P and Q must advance in lockstep, making exactly the same choices at the same time. The erroneous $P \oplus \mathbf{0} \approx P$ is avoided in a similar way. This equation can only hold if $P \rightarrow^* Q \approx \mathbf{0}$, that is, if P is already a program that may not work at all.

In fact, we will show in Section 3.2 that under a very strong assumption of scheduling fairness, $\approx$ preserves liveness properties, so we meet goal 1 fairly well. However there are still classes of errors that we miss entirely, notably *livelocks*, i.e., situations where components careen endlessly sending useless messages to themselves, rather than producing useful output. However, we do detect livelocks where the components can *only* send useless messages to themselves, and in the cases where they *could* send useful output, randomized scheduling usually will avoid such livelocks.

Unfortunately, the previously perfect situation with respect to goals 4 and 2 is now compromised. While all of the example equations we have seen so far are still valid for $\approx$, the three-way to two-way join compilation of Section 2.2 now fails to check if we add arguments to the x, y, and z messages, and use several such messages with different values. The three-way join cannot emulate the decision of the two-way merger to group two x and y values, without deciding which z value will go with them. Nonetheless the compilation is intuitively correct, and does preserve all liveness properties; but $\approx$ fails, and the reasons for the failure are only technical.

2.5 Bisimulation proof techniques

The basic definition of bisimulation give only a rough idea of how to prove a bisimilarity equation. There are a number of now well-established techniques for showing bisimilarity, notably to deal with contexts and captures, and to "close" large diagrams.

First of all, let us consider the problem of quantifying over arbitrary contexts. In Section 4 we present an extension of the join calculus, and a new equivalence, that avoid the need to consider arbitrary contexts altogether. For the time being, let us show that they are really not much of an issue. Suppose we want to show $P \approx Q$; then we should have (P, Q) in our candidate relation, as well as all terms $(C[P], C[Q])$. Now, as soon as P (and Q) starts exchanging messages with $C[\cdot]$, $C[\cdot]$ and P will become intermingled. However, if we use structural equivalence to decompose $C[\cdot]$, P, and Q, it becomes clear that the situation is not so intricate : we can take $C[\cdot] \equiv \texttt{def}\ D_C\ \texttt{in}\ (M_C \mid [\cdot])$, $P \equiv \texttt{def}\ D_P\ \texttt{in}\ M_P$, $Q \equiv \texttt{def}\ D_Q\ \texttt{in}\ M_Q$, where M_C, M_P, M_Q are parallel compositions of messages, and all bound names are fresh, except that D_C may define some free names of P and Q. With those notations, we see that elements of R will have the shape

$$(\texttt{def}\ D_C \wedge D_P\ \texttt{in}\ (M_C \mid M_P), \texttt{def}\ D_C \wedge D_Q\ \texttt{in}\ (M_C \mid M_Q))$$

To allow for extrusions from P to $C[\cdot]$, we simply allow D_C and M_C to contain channel names that have been exported by P and Q, and are thus defined in

both D_P and D_Q; this may require applying different substitutions σ_P and σ_Q to D_C and M_C, to account for different names in P and Q.

It should also be clear that in this setting, the reduction step analysis is not especially complicated by the presence of the arbitrary context. We can classify reduction steps in four categories:

1. reductions that use an unknown rule in D_C, with only unknown messages in M_C.
2. reductions that use an unknown rule in D_C, with some messages in M_P.
3. reductions that use a known rule in D_P, but with some unknown messages in $M_C\sigma_P$.
4. reductions that use a known rule in D_P, with only known messages in M_P.

The first two cases are easy, since a syntactically similar reduction must be performed by the right hand term. In the second case the messages in M_P must be matched by messages in M_Q, possibly after some internal computation using known rules in D_Q and other known messages in M_Q. Cases 3 and 4 may be harder, since Q need not match the reduction precisely. In case 3, the exact number and valence of the "unknown" messages is determined by the known rule and messages, and those messages are similarly available to Q.

Note that cases 2, 3, and 4 correspond directly to output, input, and internal steps in a trace or labeled semantics. Hence, the extra work required for those "arbitrary contexts" amounts to two generic placeholders D_C and M_C in the description of the candidate bisimulation, and one extra trivial case... that is, not very much. Furthermore, we can often simplify the relation by hiding parts common to D_P and D_Q, or M_P and M_Q, inside D_C or M_C, respectively. This is equivalent to the "up to context" labeled bisimulation proof technique.

There is one final wrinkle to this proof technique : to be an evaluation context congruence, a relation $\mathcal{R}$ of the shape above should contain all alpha variants of $\texttt{def}\ D_P\ \texttt{in}\ M_P$ and $\texttt{def}\ D_Q\ \texttt{in}\ M_Q$, to avoid clashes with the components of a new context $C'[\cdot]$ that is being added. This is easily handled by completing $\mathcal{R}$ with all pairs $(C[P]\rho, C[Q]\rho)$ for all injective renamings ρ. Reduction diagrams established for $\mathcal{R}$ also hold for the extended $\mathcal{R}$, and we can use the renamings to avoid name clashes with $C'[\cdot]$. In fact, we can further use the renamings to reserve a set of private bound names for the D_P and D_Q definitions.

The second technique we explore facilitates the diagram proof part, and makes it possible to meet the bisimulation requirement with a smaller relation. The general idea is to use equational reasoning to "close off" these diagrams, as we did in the proof of theorem 12. Unfortunately, the unrestricted use of $\approx$ in simulation diagrams is unsound: let $P = \texttt{def}\ x\langle\rangle \triangleright y\langle\rangle\ \texttt{in}\ x\langle\rangle$, and consider the singleton relation $\{(P, \mathbf{0})\}$. If $P \to Q$ then $Q \equiv \texttt{def}\ x\langle\rangle \triangleright y\langle\rangle\ \texttt{in}\ y\langle\rangle$, so $Q \approx P$ by the analog of lemma 13. So $\{(P, \mathbf{0})\}$ is a simulation up to $\approx$, and it also preserves immediate barbs (there are none). But $\{(P, \mathbf{0})\}$ does not preserve barbs ($P \Downarrow_y$ but $\mathbf{0} \not\Downarrow_y$), so it certainly is not a simulation.

To allow some measure of equational reasoning, we define a more restrictive notion of simulation, following [40]. This notion does not really have a valid semantic interpretation (it does step counting), but it is a convenient technical

expedient that allows the use of several important equations inside simulation diagrams, the most important of which is beta reduction.

Definition 15 (Tight simulations). *A relation $\mathcal{R}$ is a tight simulation when*

$$\begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \big\downarrow{\scriptstyle =} & & \vdots\!\!\downarrow{\scriptstyle =} \\ P' & \overset{\mathcal{R}}{\cdots\cdots} & Q' \end{array}$$

where $P \rightarrow^{=} P'$ means $P \rightarrow P'$ or $P = P'$.

Definition 16 (Expansion). *An* expansion *is a simulation whose converse, called a* compression*, is a tight simulation. The coarsest expansion that respects the barbs is denoted $\preceq$.*

A tight bisimulation *is a tight simulation whose converse is a tight simulation. The coarsest tight bisimulation that respects the barbs is denoted $\asymp$.*

These technical definitions find their use with the following reformulations:

Theorem 17 (Simulations up to).
A relation $\mathcal{R}$ is a tight bisimulation when

$$\begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \big\downarrow & & \vdots\!\!\downarrow{\scriptstyle =} \\ P' & \overset{(\asymp \cup \mathcal{R})^*}{\cdots\cdots} & Q' \end{array} \qquad \begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \vdots\!\!\downarrow{\scriptstyle =} & & \big\downarrow \\ P' & \overset{(\asymp \cup \mathcal{R})^*}{\cdots\cdots} & Q' \end{array}$$

A relation $\mathcal{R}$ is an expansion when

$$\begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \big\downarrow & & \vdots\!\!\downarrow{\scriptstyle *} \\ P' & \overset{\asymp \mathcal{R} \preceq}{\cdots\cdots} & Q' \end{array} \qquad \begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \vdots\!\!\downarrow{\scriptstyle =} & & \big\downarrow \\ P' & \overset{(\preceq \cup \mathcal{R})^*}{\cdots\cdots} & Q' \end{array}$$

A relation $\mathcal{R}$ is a bisimulation when

$$\begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \big\downarrow & & \vdots\!\!\downarrow{\scriptstyle *} \\ P' & \overset{\succeq \mathcal{R} \approx}{\cdots\cdots} & Q' \end{array} \qquad \begin{array}{ccc} P & \overset{\mathcal{R}}{\text{———}} & Q \\ \vdots\!\!\downarrow{\scriptstyle *} & & \big\downarrow \\ P' & \overset{\approx \mathcal{R} \preceq}{\cdots\cdots} & Q' \end{array}$$

Most of the equations established with $\simeq_{\text{may}}$ are in fact valid compressions, and so can be used to close diagrams in bisimulation proofs. In particular, beta reduction is a compression.

Theorem 18 (Beta compression). *If $Q[\cdot]_S$ is a general context that does not capture f or any names free in E ($S \cap (\{f\} \cup \mathrm{fv}(E)) = \emptyset$), then*

$$\begin{aligned} &\texttt{let}\ f(\widetilde{x}) = E\ \texttt{in}\ Q[\texttt{let}\ z = f(\widetilde{u})\ \texttt{in}\ R]_S \\ \succeq\ &\texttt{let}\ f(\widetilde{x}) = E\ \texttt{in}\ Q[\texttt{let}\ z = E\{{}^{\widetilde{u}}/_{\widetilde{x}}\}\ \texttt{in}\ R]_S \end{aligned}$$

3 A hierarchy of equivalences

In this section and the next one, we continue our comparative survey of equivalences and their proof techniques. We provide useful intermediate equivalences between may-testing and bisimilarity. We give finer labeled semantics with purely coinductive proofs of equivalence. We also discuss the trade-off between different definitions of these equivalences. At the end of this section, we summarize our results as a hierarchy of equivalences ordered by inclusion.

Although we develop this hierarchy for establishing the properties of programs written in the join calculus, most of these equivalences and techniques are not specific to the join calculus. In principle, they can be applied to any calculus with a small-step reduction-based semantics, evaluation contexts, and some notion of observation. They provide a flexible framework when considering new calculi, or variants of existing calculi (see for instance [15,1,17]).

3.1 Too many equivalences?

In concurrency theory, there is a frightening diversity of calculi and equivalences (see, for instance, Glabbeek's classification of weak equivalences [19]). Even in our restricted setting—asynchronous concurrent programs—there are several natural choices in the definition of process equivalence, and for many of these choices there is a record of previous works that exclusively rely on each resulting equivalence.

In Section 2, we detailed the (largely contradictory) goals for the "right" equivalences. Before introducing useful variants, we now highlight some of the technical choices in their definitions. In order to choose the variant best adapted to the problem at hand, we need to understand the significance of such choices. For example, different styles of definition may yield the same equivalence and still provide different proof techniques. Conversely, some slight changes in a definition may in fact strongly affect the resulting equivalence and invalidate its expected properties.

Context closures As illustrated in the previous section, one may be interested in several classes of contexts. Evaluation contexts are easy to interpret in a testing scenario, but more general context-closure properties may also be useful in lemmas (e.g., beta compression). Technically, the main novelty of general contexts is that different free variables appearing under a guard may be instantiated to the same name. In the join calculus, this is inessential because *relays* from one name to another have the same effect, as expressed in Lemma 13 for may testing, but this is not the case in extensions of the join calculus considered in Section 4.5.

Conversely, it may be convenient to consider smaller classes of contexts to establish context closure. For instance, one may constrain the names that appear in the context, considering only contexts with a few, chosen free variables, or contexts whose local names never clash with those of the processes at hand, or contexts with a single, "flat" definition.

In the following discussion, we write $\mathcal{R}^\circ$ for the congruence of relation $\mathcal{R}$, defined as $P\ \mathcal{R}^\circ\ Q$ iff $\forall C[\cdot], C[P]\ \mathcal{R}\ C[Q]$. As our usual equivalences are all closed by application of evaluation contexts, we use plain relation symbols ($\simeq$, $\approx$,...) for them, and "dotted" relation symbols for the sibling relations defined without a context-closure requirement ($\dot{\simeq}$, $\dot{\approx}$,...).

Primitive Observations The notion of basic observation is largely arbitrary. So far, we have been using the output predicates of Definition 4, given as the syntactic presence of particular messages on free names in evaluation contexts. Moreover, in our definitions of equivalences, we use a distinct predicate $\downarrow_x$ for every channel name, and we don't discriminate according to the content of message. Another, very detailed observation predicate of Section 2 is given by Definition 7, as one can test whether a particular execution trace is allowed by a process. Other, natural alternatives are considered below.

Fortunately, the details of observation often become irrelevant when considering relations that are (at least) closed by application of evaluation contexts. Indeed, as soon as we can use a primitive observation predicate to separate two processes, then any other "reasonable" notion of observation should be expressible using a particular context that observes it, then conveys the result of this internal observation by reducing to either of these two processes.

Existential Predicates. In the initial paper on barbed equivalences [33], and in most definitions of testing equivalences, a *single predicate* is used instead of an indexed family. Either there is a single observable action, often written ω, or there is a single, "existential" predicate that collectively tests the presence of any message on a free name. Accordingly, for every family of predicates such as $\Downarrow_x$, we may define an existential predicate $P \Downarrow \stackrel{\text{def}}{=} \exists x. P \Downarrow_x$, and obtain existential variants for any observation-based equivalence. For instance, we let $\simeq^{\exists}_{\text{may}}$ be the largest relation closed by application of evaluation contexts that refines $P \Downarrow$. As suggested above, existential equivalences coincide with their basic equivalence when they are context-closed. For instance, one can specifically detect $\Downarrow_x$ by testing $\Downarrow$ in a context that restricts all other free names of the process being tested, and one can test for $\Downarrow$ as a the conjunction of all predicates $\Downarrow_x$, hence $\simeq^{\exists}_{\text{may}} = \simeq_{\text{may}}$. In the following, we will consider equivalences whose existential variant is much weaker.

Transient Observations. In the join calculus, strong barbs $\downarrow_x$ appear as messages on free names, which are names meant to be defined by the context. Due to the locality property, these messages are preserved by internal reductions, hence the strong barbs are stable: if $P \downarrow_x$ and $P \rightarrow^* P'$, then also $P' \downarrow_x$).

This is usually not the case in process calculi such as CCS or the pi calculus, where messages on free names can disappear as the result of internal communication. For instance, the reduction $\overline{x}\langle\rangle \mid x() \rightarrow \mathbf{0}$ erases the barb $\downarrow_x$. Transient barbs may be harder to interpret in terms of observation scenarios, and they complicate the hierarchy of equivalence [15]. However, one can enforce

the permanency of barbs using a derived *committed message* predicate $P \Downarrow_x \stackrel{\text{def}}{=}$ if $Ps \to^* P'$ then $P' \downarrow_x$, instead of the predicate $\downarrow_x$ in every definition. One can also rely on the congruence property (when available) to turn transient barbs into permanent ones. In the pi calculus for instance, we let $T_x[\cdot] \stackrel{\text{def}}{=} \nu x.(x().\bar{t}\langle\rangle|[\cdot])$ and, for any process P where t is fresh, we have $T_x[P] \to^* \Downarrow_t$ iff $P \Downarrow_x$.

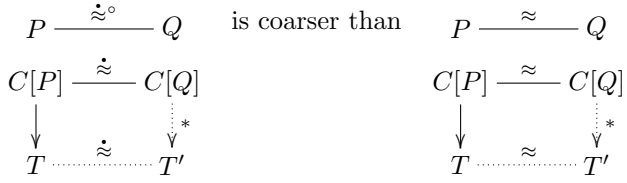
Relations between internal states So far, we considered a "pure" testing semantics $\simeq_{\text{may}}$ and a much finer bisimulation-based semantics $\approx$ that requires an exact correspondence between internal states. The appearance of bisimulation raises two questions:

Can we use a coarser correspondence between internal states? This is an important concern in our setting, because asynchronous algorithms typically use a series of local messages to simulate an atomic "distributed" state transition. Since these messages are received independently, there is a gradual commitment to this state transition, which introduces transient states. Hence, the two processes do not advance in lockstep, and they are not bisimilar. In Section 3.3, we explain how we can retain some benefits of coinductive definitions in such cases.

Should this correspondence depend on the context? In the previous section, we defined $\approx$ all at once, as the largest $\Downarrow_x$-preserving bisimulation that is also closed by application of evaluation contexts. This style of equivalence definition was first proposed for the ν-calculus in [21,23,22].

However, there is another, more traditional definition for CCS and for the pi calculus [33,38,41]. First, define the largest barbed bisimulation (written $\dot{\approx}$); then, take the largest subrelation of $\dot{\approx}$ that is closed by application of evaluation contexts (written $\dot{\approx}^\circ$). We believe that this original, two-stage definition has several drawbacks: the bare bisimulation $\dot{\approx}$ is very sensitive to the choice of observation predicates, and the correspondence between internal states may depend on the context.

The two diagrams below detail the situation: once a context has been applied on the left, the stable relation is a bisimulation, and not a congruence. Conversely, the bisimulation-and-congruence relation on the right retains the congruence property after matching reductions, and allows repeated application of contexts after reductions.



From the two definitions, we obtain the simple inclusion $\approx \subseteq \dot{\approx}^\circ$, but the converse $\dot{\approx}^\circ \subseteq \approx$ is far from obvious: the latter inclusion holds if and only if $\dot{\approx}^\circ$ is itself a bisimulation. In Section 3.4, we will sketch a proof that $\dot{\approx}^\circ \subseteq \approx$ in the

join calculus. Conversely, we will show that this is not the case for some simple variants of $\dot{\approx}$.

Even if the two definitions yield the same equivalence, they induce distinct proof techniques. In our experience, $\approx$ often leads to simpler proofs, because interaction with the context is more abstract: after reduction, the context may change, but remains in the class being considered in the candidate bisimulation-and-congruence.

3.2 Fair testing

We are now ready to revisit our definition of testing equivalences. May testing is most useful for guaranteeing safety properties but, as illustrated by the equation $P \oplus \mathbf{0} \simeq_{\text{may}} P$, it does not actually guarantee that anything useful happens.

As we argued in subsection 2.4, for concurrent programs it is not desirable to supplement may testing with must testing in order to capture liveness properties. We briefly considered the usual *must predicate* $\Box\downarrow_x$, but dismissed it because the predicate always failed on processes with infinite behavior (*cf.* page 29).

Instead, we use a stronger predicate that incorporates a notion of "abstract fairness". The *fair-must predicate* $\Box\Downarrow_x$ detects whether a process always retains the possibility of emitting on x:

$$P \,\Box\Downarrow_x \stackrel{\text{def}}{=} \text{if } P \to^* P', \text{ then } P' \to^* P'' \downarrow_x$$

Hence, $\Box\Downarrow_x$ tests for "permanent weak barbs". For all processes P, the test $P \,\Box\Downarrow_x$ implies $P \Downarrow_x$ and $P \,\Box\downarrow_x$. Conversely, (1) if all reductions from P are deterministic, then $\Box\Downarrow_x$ and $\Downarrow_x$ coincide; (2) if there is no infinite computation, then $\Box\downarrow_x$ and $\Box\Downarrow_x$ coincide.

Much like weak barbs, fair-must predicates induces a contextual equivalence:

Definition 19 (Fair testing equivalence). *We have $P \sqsubseteq_{\text{may}} Q$ when, for any evaluation context $C[\cdot]$ and channel name x, $C[P] \,\Box\Downarrow_x$ if and only if $C[Q] \,\Box\Downarrow_x$.*

That is, fair testing is the largest congruence that respects all fair-must predicates. Similar definitions appear in [9,34,10]. Fair testing detects deadlocks: we have $x\langle\rangle \oplus (x\langle\rangle \oplus \mathbf{0}) \simeq_{fair} x\langle\rangle \oplus \mathbf{0}$, but $x\langle\rangle \oplus \mathbf{0} \not\simeq_{fair} x\langle\rangle$ and $x\langle\rangle \oplus \mathbf{0} \not\simeq_{fair} \mathbf{0}$.

The particular notion of fairness embedded in fair testing deserves further explanations: both may and fair-must predicates state the existence of reductions leading to a particular message, but they don't provide a reduction strategy. Nonetheless, we can interpret $P \,\Box\Downarrow_x$ as a successful observation "P eventually emits the message $x\langle\rangle$". As we do so, we consider only infinite traces that emit on x and we disregard any other infinite trace. Intuitively, the model is the set of barbs present on finite and infinite fair traces, for a very strong notion of fairness. For example, we have the fair testing equivalence:

$$\texttt{def } t\langle\rangle \triangleright x\langle\rangle \;\wedge\; t\langle\rangle \triangleright t\langle\rangle \texttt{ in } t\langle\rangle \simeq_{fair} x\langle\rangle$$

where the first process provides two alternatives in the definition of t: either the message $x\langle\rangle$ is emitted, or the message $t\langle\rangle$ is re-emitted, which reverts the

process to its initial state. It is possible to always select the second, stuttering branch of the alternative, and thus there are infinite computations that never emit $x\langle\rangle$. Nonetheless, the possibility of emitting on x always remains, and any fair evaluation strategy should eventually select the first branch.

Fair testing may seem unrelated to may testing; at least, these relations are different, as can be seen using $x\langle\rangle \oplus \mathbf{0}$ and $x\langle\rangle$. Actually, fair testing is strictly finer: $\simeq_{fair} \subset \simeq_{\text{may}}$. Said otherwise, fair testing is also the largest congruence relation that refines both may- and fair-must predicates.

To prove that $\simeq_{fair}$ also preserves weak barbs $\Downarrow_x$, we use the congruence property with non-deterministic contexts of the form

$$C[\cdot] \stackrel{\text{def}}{=} \texttt{def}\ r\langle z\rangle \mid \mathit{once}\langle\rangle \triangleright z\langle\rangle\ \texttt{in}\ \big(\ r\langle y\rangle \mid \mathit{once}\langle\rangle \mid \texttt{def}\ x\langle\rangle \triangleright r\langle x\rangle\ \texttt{in}\ [\cdot]\ \big)$$

and establish that $P \Downarrow_x$ iff $C[P] \not\Downarrow_y$. This property of fair testing also holds in CCS, in the pi calculus, and for Actors, where a similar equivalence is proposed as the main semantics [5].

As regards discriminating power, fair testing is an appealing equivalence for distributed systems: it is stronger than may-testing, detects deadlocks, but remains insensitive to termination and livelocks. Note, however, that "abstract fairness" is much stronger than the liveness properties that are typically guaranteed in implementations (*cf.* the restrictive scheduling policy in JoCaml).

Fair testing suffers from another drawback: direct proofs of equivalence are very difficult because they involve nested inductions for all quantifiers in the definition of fair-must tests in evaluation context. As we will see in the next section, the redeeming feature of fair testing is that it is coarser than bisimilarity equivalence ($\approx \subseteq \simeq_{fair}$). Thus, many equations of interest can be established in a coinductive manner, then interpreted in terms of may and fair testing scenarios. Precisely, we are going to establish an tighter characterization of fair testing in terms of *coupled simulations*.

3.3 Coupled Simulations

The relation between fair testing and bisimulations has been initially studied in CCS; in [9,34] for instance, the authors introduce the notion of fair testing (actually should testing in their terminology), and remark that weak bisimulation equivalences incorporates a particular notion of fairness; they identify the problem of gradual commitment, and in general of sensitivity to the branching structure, as an undesirable property of bisimulation; finally, they establish that observational equivalence is finer than fair testing and propose a simulation-based sufficient condition to establish fair testing.

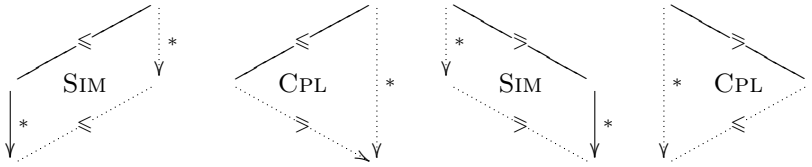
Independently, *coupled simulations* have been proposed to address similar issues [36]; this coarse simulation-based equivalence does not require an exact correspondence between the internal choices of processes, and is thus less sensitive than bisimulation to their branching structure. In our setting, we use a barbed counterpart of *weakly-coupled simulations* [37] that is not sensitive to divergence. A similar equivalence appears in the asynchronous pi calculus, where it is used to establish the correctness of the encoding of the choice operator [35].

Definition 20. *A pair of relations* $\leqslant, \geqslant$ *are* coupled simulations *when* $\leqslant$ *and* $\geqslant^{-1}$ *are two simulations that satisfy the coupling conditions* $\leqslant \subseteq \geqslant \leftarrow^*$ *and* $\geqslant \subseteq \rightarrow^* \leqslant$.

A barbed coupled-simulations relation *is an intersection* $\leqslant \cap \geqslant$ *for some pair* $\leqslant, \geqslant$ *such that* $\leqslant$ *and* $\geqslant^{-1}$ *preserve the barbs.*

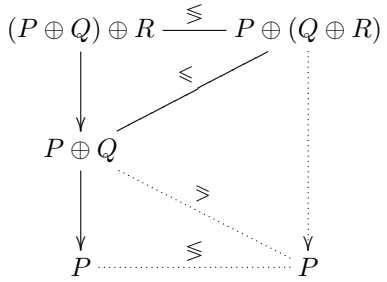
The coarsest barbed coupled-simulation relation is denoted $\dot{\lesssim}$. *The coarsest barbed coupled-simulation obtained from simulations that are also precongruences for evaluation contexts is called* coupled-similarity equivalence*, and denoted* $\lesssim$.

The definition can be stated in a more readable manner using diagrams for all simulation and coupling requirements:



When we also have $\leqslant = \geqslant$, the coupling diagrams are trivially verified, and the coupled-simulation relation is in fact a bisimulation—in particular, we immediately obtain the inclusions $\dot{\approx} \subseteq \dot{\lesssim}$ and $\approx \subseteq \lesssim$.

Typically, the discrepancy between $\leqslant$ and $\geqslant$ is used to describe processes that are in a transient state, bisimilar neither to the initial state nor to any final state. For instance, using our derived internal choice operator $\oplus$, we have the diagram



The precise relation between fair testing and coupled simulations is intriguing. These equivalences have been applied to the same problems, typically the analyses of distributed protocols where high-level atomic steps are implemented as a negotiation between distributed components, with several steps that perform a gradual commitment. Yet, their definitions are very different, and they both have their advantages: fair-testing is arguably more natural than coupled simulations, while coupled simulations can be established by coinduction.

It is not too hard to establish that barbed coupled-simulation relations also preserve fair-must barbs. The proof uses simulations in both directions, which somehow reflects the alternation of quantifiers in the definition of fair-must barbs.

Lemma 21. *Let $\leqslant, \geqslant$ be barbed coupled simulations. If $P \geqslant Q$ and $P \,\Box\!\!\Downarrow_x$, then also $Q \,\Box\!\!\Downarrow_x$.*

Proof. If $Q \to^* Q'$, these reductions can be simulated by $P \to^* P' \geqslant Q'$. Using the coupling condition, we also have $P' \to^* P'' \leqslant Q'$. By definition of $P \,\Box\!\!\Downarrow_x$, we have $P'' \Downarrow_x$. Finally, $\leqslant$ preserves weak barbs, and thus $Q' \Downarrow_x$.

As is the case for weak bisimulation, we can either add precongruence requirements to the definition of barbed coupled simulations and obtain a barbed coupled-simulations congruence ($\lesssim$) or take the largest congruence that is contained in the largest barbed coupled-simulation relation (written $\dot{\lesssim}^\circ$).

From these definitions and the previous lemma, we obtain the inclusions $\lesssim \subseteq \dot{\lesssim}^\circ \subseteq \simeq_{fair}$. Conversely, the congruence relation $\dot{\lesssim}^\circ$ is *not* a coupled-simulation relation, and thus $\lesssim$ is strictly finer than $\dot{\lesssim}^\circ$. The difference appears as soon as internal choices are spawned *between* visible actions. The counter-example is especially simple in asynchronous CCS, where we have:

$$a.\overline{b} \oplus a.\overline{c} \;\not\lesssim\; a.(\overline{b} \oplus \overline{c})$$
$$a.\overline{b} \oplus a.\overline{c} \;\dot{\lesssim}^\circ\; a.(\overline{b} \oplus \overline{c})$$

In these processes, the outcome of the internal choice becomes visible only after communication on a, but the choice can be immediately performed on the left only, e.g. $a.\overline{b} \oplus a.\overline{c} \to a.\overline{b}$. If the context is applied once for all, then we know whether that context can provide a message on a. If this is the case, then we simulate the step above by getting that message, communicating on a, and reducing $\overline{b} \oplus \overline{c}$ to $\overline{b}$ on the right. Otherwise, we simulate the step by doing nothing, because both processes are inert. In contrast, if the reduction occurs on the left before the context is chosen, then we cannot simulate it on the right in a uniform manner. A similar counter-example holds in the join calculus [13].

Our next theorem relates fair testing and barbed coupled similarities; it relies on the preceding remarks, plus the surprising inclusion $\simeq_{fair} \subseteq \dot{\lesssim}^\circ$, whose proof is detailed below.

Theorem 22. $\simeq_{fair} = \dot{\lesssim}^\circ \subset \lesssim$.

To prove $\simeq_{fair} \subseteq \dot{\lesssim}^\circ$, we develop a semantic model of coupled simulations. We first consider a family of processes whose behavior is especially simple. We say that a process P is *committed* when, for all x, we have $P \Downarrow_x$ iff $P \,\Box\!\!\Downarrow_x$. Then, no internal reduction may visibly affect P: let S be the set of names

$$S \stackrel{\text{def}}{=} \{x \mid P \,\Box\!\!\Downarrow_x\} = \{x \mid P \Downarrow_x\}$$

For all P' such that $P \to^* P'$, P' is still committed to S. In a sense, P has converged to S, which determines its outcome.

To every process P, we now associate the semantics $P^\flat \in \mathbb{P}(\mathbb{P}(\mathcal{N}))$ that collects these sets of names for all the committed derivatives of P:

$$P^\flat \stackrel{\text{def}}{=} \left\{ S \subseteq \mathcal{N} \mid \exists P' \cdot P \to^* P' \text{ and } S = \{x \,|\, P' \,\Box\!\!\Downarrow_x\} = \{x \,|\, P' \Downarrow_x\} \right\}$$

For example, $\mathbf{0}^\flat$ is the singleton $\{\emptyset\}$ and $(x\langle\rangle \oplus y\langle\rangle)^\flat$ is the pair $\big\{\{x\},\{y\}\big\}$. As is the case for weak barbs, $P^\flat$ decreases by reduction. The predicates $\Downarrow_x$ and $\Box\Downarrow_x$ can be recovered as follows: we have $P \Downarrow_x$ if and only if $x \in \bigcup P^\flat$, and $P \Box\Downarrow_x$ if and only if $x \in \bigcap P^\flat$.

Let $\subseteq_\flat$ be the preorder defined as $P \subseteq_\flat Q \stackrel{\text{def}}{=} P^\flat \subseteq Q^\flat$. By definition of may testing and fair testing preorders, we immediately obtain ${\subseteq_\flat}^\circ \subseteq \sqsubseteq_{may}$ and ${\subseteq_\flat}^\circ \subseteq \sqsubseteq_{fair}^{-1}$. Actually, the last two preorders coincide.

Lemma 23. ${\subseteq_\flat}^\circ = \sqsubseteq_{fair}^{-1}$.

Proof. For any finite sets of names S and N such that $S \subseteq N$ and $t \notin N$, we define the context

$$T_S^N[\cdot] \stackrel{\text{def}}{=} \texttt{def}\ \ once\langle\rangle \triangleright t\langle\rangle\ \ \wedge\ \ once\langle\rangle \,|\, \textstyle\prod_{x\in S} x\langle\rangle \triangleright \mathbf{0}\ \ \wedge \bigwedge_{x\in N\setminus S} x\langle\rangle \triangleright t\langle\rangle\ \ \texttt{in}\ \ once\langle\rangle \,|\, [\cdot]$$

and we show that $T_S^N[\cdot]$ fair-tests exactly one set of names in our semantics: for all P such that $\mathrm{fv}(P) \subseteq N$, we have $T_S^N[P] \Box\Downarrow_t$ if and only if $S \notin P^\flat$.

The first two clauses of the definition compete for the single message $once\langle\rangle$, hence at most one of the two may be triggered. The first clause ($once\langle\rangle \triangleright t\langle\rangle$) can be immediately triggered. The second clause can preempt this reduction only by consuming a message for each name in S. The third clause detects the presence of any message on a name in $N \setminus S$. The predicate $T_S^N[P] \Box\Downarrow_t$ holds iff all the derivatives of P keep one of the two possibilities to emit the message $t\langle\rangle$, namely either don't have messages on some of the names in S, or can always produce a message on a name outside of S.

Let $P \sqsubseteq_{fair}^{-1} Q$. We let $N = \mathrm{fv}(P) \cup \mathrm{fv}(Q)$ to establish $P^\flat \subseteq Q^\flat$. For every set of names $S \subseteq N$, we have $S \in P^\flat$ iff $T_S^N[P] \not\Box\Downarrow_t$; this entails $T_S^N[Q] \not\Box\Downarrow_t$ and $S \in Q^\flat$. For every other set of names S, neither $P^\flat$ nor $Q^\flat$ may contain S anyway. Thus $\sqsubseteq_{fair}^{-1} \subseteq \subseteq_\flat$, by context-closure for fair-testing $\sqsubseteq_{fair}^{-1} \subseteq {\subseteq_\flat}^\circ$, and, since the converse inclusion follows from the characterization of fair barbs given above, ${\subseteq_\flat}^\circ = \sqsubseteq_{fair}^{-1}$.

The next lemma will be used to relate $\subseteq_\flat$ to $\dot{\lesssim}$:

Lemma 24. $P^\flat \neq \emptyset$

Proof. For every process P, consider the series of processes $P'_0, P'_1, \ldots, P'_n$ such that $P = P'_0 \to^* P'_1 \to^* \cdots \to^* P'_n$ and such that $\{x \mid P'_i \Box\Downarrow_x\}$ strictly increases with i. There is a least one such series (P), and the length of any series is bounded by the number of names free in P, hence there is at least a series that is maximal for prefix-inclusion.

To conclude, we remark that $S \in P^\flat$ iff there is a maximal series of processes ending at P'_n such that $S = \{x \mid P'_n \Box\Downarrow_x\}$.

We now establish that our semantics refines barbed coupled similarity.

Lemma 25. *$(\subseteq_\flat, \supseteq_\flat)$ are coupled barbed simulations, and thus $\subseteq_\flat \cap \supseteq_\flat \subseteq \dot{\lesssim}$.*

Proof. We successively check that $\subseteq_\flat$ preserves the barbs, is a simulation, and meets the coupling diagram. Assume $P \subseteq_\flat Q$.

1. The barbs can be recovered from the semantics: $P \Downarrow_x$ iff $x \in \bigcup P^\flat$, and if $P \subseteq_\flat Q$ then also $x \in \bigcup Q^\flat$ and $Q \Downarrow_x$. hence $P \Downarrow_x$ implies $Q \Downarrow_x$.
2. Weak simulation trivially holds: by definition, $P^\flat$ decreases with reductions, and is stable iff $P^\flat$ is a singleton; for every reduction $P \to P'$, $P' \subseteq_\flat P \subseteq_\flat Q$, and thus reductions in P are simulated by the absence of reduction in Q.
3. $P^\flat$ is not empty, so let $S \in P^\flat$. By hypothesis, $S \in Q^\flat$ and thus for some process Q'_S we have $Q \to^* Q'_S$ and ${Q'_S}^\flat = \{S\} \subseteq P^\flat$, which provides the coupling condition from $\subseteq_\flat$ to $\supseteq_\flat$. □

By composing Lemmas 23 and 25, we obtain $\simeq_{fair} = \dot{\lesssim}^\circ$ (Theorem 22). The proof technique associated with this characterization retains some of the structure of the purely coinductive technique for the stronger relation $\lesssim$, so it is usually an improvement over the triple induction implied by the definition of $\simeq_{fair}$ (but not always, as was pointed out in subsection 2.4 for coinductive may-testing proofs).

3.4 Two notions of congruence

We finally discuss the relation between the equivalences $\approx$ and $\dot{\approx}^\circ$, which depend on the choice of observables. To this end, we consider bisimulations weaker than $\dot{\approx}$, obtained by considering only a finite number of observation predicates.

Let single-barbed bisimilarity $\dot{\approx}_\exists$ be the largest weak bisimulation that refines the barb $\Downarrow$, i.e. that detects the ability to send a message on any free name.

- The equivalence $\dot{\approx}_\exists$ partitions join calculus processes into three classes characterized by the predicates $\Box\Downarrow$, $\not\Downarrow$, and $\Downarrow \wedge \not\Box\Downarrow$. Hence, the congruence of single-barbed bisimilarity is just fair testing equivalence: $\dot{\approx}^\circ_\exists = \simeq_{fair}$. This characterization implies yet another proof technique for $\simeq_{fair}$, but the technique implied by Theorem 22 is usually better.
- In contrast, both $\approx_\exists$ and $\approx$ are congruences and weak bisimulations. Moreover, using the existential contexts given above, we can check that $\approx_\exists$ preserves nominal barbs $\Downarrow_x$ and that $\approx$ preserves existential barbs $\Downarrow$. This establishes $\approx_\exists = \approx$.

We thus obtain a pair of distinct "bisimulation congruences" $\dot{\approx}^\circ_\exists \neq \approx_\exists$.

While there is a big difference between one and several observation predicates, it turns out that two predicates are as good as an infinite number of them, even for the weaker notion of bisimulation congruence. In the following, we fix two nullary names x and y, and write $\dot{\approx}_2$ for the bisimilarity that refines $\Downarrow_x$ and $\Downarrow_y$. This technical equivalence is essential to prove $\approx = \dot{\approx}^\circ$, Precisely, we are going to establish

Theorem 26. $\dot{\approx}^\circ_2 = \approx$

Since we clearly have $\approx \subseteq \dot{\approx}^\circ \subseteq \dot{\approx}_2^\circ$, we obtain $\approx = \dot{\approx}^\circ$ as a corollary.

We first build a family of processes that are not $\dot{\approx}_2$-equivalent and retain this property by reduction. Informally, this show that there are infinitely many ways to hesitate between two messages in a branching semantics. We define an operator $\mathcal{S}(\cdot)$ that maps every finite set of processes to the set of its (strict, partial) internal sums:

$$\mathcal{S}(\mathcal{P}) \stackrel{\text{def}}{=} \left\{ \bigoplus_{P \in \mathcal{P}'} P \mid \mathcal{P}' \subseteq \mathcal{P} \text{ and } |\mathcal{P}'| \geq 2 \right\}$$

Lemma 27. *Let $\mathcal{R}$ be a weak bisimulation and $\mathcal{P}$ be a set of processes such that, for all $P, Q \in \mathcal{P}$, the relation $P \rightarrow^* \mathcal{R}\ Q$ implies $P = Q$. Then we have:*

1. *The set $\mathcal{S}(\mathcal{P})$ retains this property.*
2. *The set $\bigcup_{n \geq 0} \mathcal{S}^n(\mathcal{P})$ that collects the iterated results of $\mathcal{S}(\cdot)$ contains only processes that are not related by $\mathcal{R}$.*

Proof. We first show that (0) if $P \rightarrow^* \mathcal{R}\ Q$ for some $Q \in \mathcal{S}(\mathcal{P})$, then $P \notin \mathcal{P}$. Since Q has at least two summands, it must have a summand $Q' \neq P$. But then $P \rightarrow^* \mathcal{R}\ Q'$ since $\mathcal{R}$ is a bisimulation, and since $Q' \in \mathcal{P}$ we cannot have $P \in \mathcal{P}$.

We then deduce (1) of the lemma. Let P, Q be two processes in $\mathcal{S}(\mathcal{P})$ such that $P \rightarrow^* \mathcal{R}\ Q$. Let Q' be a summand of Q; we must have $P \rightarrow^* \mathcal{R}\ Q'$ since $\mathcal{R}$ is a bisimulation, and in fact $P \rightarrow P' \rightarrow^* \mathcal{R}\ Q'$ (since $Q' \in \mathcal{P}$, $Q'\ \mathcal{R}\ P$ would break (0)); but $P', Q' \in \mathcal{P}$, so $P' = Q'$ and Q' is also a summand of P. Now we must in fact have $P\ \mathcal{R}\ Q$ since $P \rightarrow P' \rightarrow^* \mathcal{R}\ Q$ would imply $P' \in \mathcal{P}$ and thus contradict (0). Hence by symmetry any summand of P is also a summand of Q, so $P = Q$.

To prove (2), let $P \in \mathcal{S}^n(\mathcal{P})$ and $Q \in \mathcal{S}^{n+k}(\mathcal{P})$ such that $P\ \mathcal{R}\ Q$. By induction and (1), if $k = 0$ then $P = Q$; and we must have $k = 0$, since otherwise we have $Q \rightarrow^* Q'$ for some $Q' \in \mathcal{S}^{n+1}(\mathcal{P})$, hence $P \rightarrow^* \mathcal{R}\ Q'$, which breaks (0).

As a direct consequence, the bisimilarity $\dot{\approx}_2$ separates numerous processes with a finite behavior. We build an infinite set of processes as follows:

$$\mathcal{P}_0 \stackrel{\text{def}}{=} \{\ \mathbf{0},\ x\langle\rangle,\ y\langle\rangle\ \} \qquad \mathcal{P}_{n+1} \stackrel{\text{def}}{=} \mathcal{S}(\mathcal{P}_n) \qquad \mathcal{P}_\omega \stackrel{\text{def}}{=} \bigcup_{n \geq 0} \mathcal{P}_n$$

The size of each layer $\mathcal{P}_n$ grows exponentially. Thus, $\mathcal{P}_\omega$ contains infinitely many processes that are not related by $\dot{\approx}_2$: if $P \in \mathcal{P}_n$ and $Q \in \mathcal{P}_{n+m}$, then we have $Q \rightarrow^m Q'$ for some $Q' \in \mathcal{P}_n \setminus \{P\}$, and by construction at rank n this series of reduction cannot be matched by any series reductions starting from P.

This construction captures only processes with finite behaviors up to our bisimilarity, whereas $\dot{\approx}_2$ has many more classes than those exhibited here (e.g. classes of processes that can reach an infinite number of classes in $\mathcal{P}_\omega$).

Note that the same construction applies for single-barb bisimilarity, but quickly converges. Starting from the set $\{\mathbf{0}, x\langle\rangle\}$, we obtain a third, unrelated process $\mathbf{0} \oplus x\langle\rangle$ at rank 1, then the construction stops.

The next lemma states that a process can effectively communicate any integer to the environment by hesitating between two exclusive barbs $\Downarrow_x$ and $\Downarrow_y$, thanks to the discriminating power of bisimulation.

In the following, we rely on encodings for booleans and for integers à la Church inside the join calculus. To every integer $n \in \mathbb{N}$, we associate the representation $\mathtt{n}$; we also assume that our integers come with operations $\mathbf{is\_zero}\langle\cdot\rangle$ and and $\mathbf{pred}\langle\cdot\rangle$.

To every integer, we associate a particular equivalence class of $\dot{\approx}_2$ in the hierarchy of processes $\mathcal{P}_\omega$, then we write a process that receives an integer and conveys that integer by evolving to its characteristic class. Intuitively, the context $N[\cdot]$ transforms integer-indexed barbs $int\langle\mathtt{n}\rangle$ (where int is a regular name of the join calculus) into the two barbs $\Downarrow_x$ and $\Downarrow_y$.

Lemma 28. *There is an evaluation context $N[\cdot]$ such that, for any integers n and m, the three following statements are equivalent:*

1. $n = m$
2. $N[int\langle\mathtt{n}\rangle] \dot{\approx}_2 N[int\langle\mathtt{m}\rangle]$
3. $N[int\langle\mathtt{n}\rangle] \rightarrow^* \dot{\approx}_2 N[int\langle\mathtt{m}\rangle]$

To establish the lemma, we program the evaluation context $N[\cdot]$ as follows, and we locate the derivatives of $N[int\langle\mathtt{n}\rangle]$ in the hierarchy of processes $(\mathcal{P}^n)_n$.

$$N[\cdot] \stackrel{\text{def}}{=} \begin{array}{l} \mathtt{def}\ int\langle n\rangle | once\langle\rangle \triangleright \\ \left(\begin{array}{l} \mathtt{def}\ c\langle n,x,y,z\rangle \triangleright \\ \quad \mathtt{if}\ \mathbf{is\_zero}\langle\mathbf{n}\rangle\ \mathtt{then}\ z\langle\rangle \\ \quad \mathtt{else}\ c\langle\mathbf{pred}\langle\mathbf{n}\rangle,z,x,y\rangle \oplus c\langle\mathbf{pred}\langle\mathbf{n}\rangle,y,z,x\rangle\ \mathtt{in} \\ \mathtt{def}\ z\langle\rangle \triangleright \mathbf{0}\ \mathtt{in} \\ c\langle n,x,y,z\rangle \oplus c\langle n,y,z,x\rangle \oplus c\langle n,z,x,y\rangle \end{array}\right) \\ \mathtt{in}\ once\langle\rangle | [\cdot] \end{array}$$

In $N[\cdot]$, the name z is used to encode the process $\mathbf{0}$; hence the three processes in $\mathcal{P}_0$ are made symmetric, and we can use permutations of the names x, y, and z to represent them. Each integer n is associated with a ternary sum of nested binary sums in the $n+1$ layer of $\mathcal{P}$: when an encoded integer is received as $int\langle\mathtt{n}\rangle$, a reduction triggers the definition of int and yields the initial ternary sum; at the same time this reduction consumes the single message $once\langle\rangle$, hence the definition of int becomes inert.

The next lemma uses this result to restrict the class of contexts being considered in congruence properties to contexts with at most two free (nullary) variables.

Lemma 29. *Let S be a finite set of names. There is an evaluation context $C_2[\cdot]$ such that, for any processes P and Q with free variables in S, we have $P \dot{\approx} Q$ iff $C_2[P] \dot{\approx}_2 C_2[Q]$.*

In order to establish that $\dot{\approx}_2^\circ$ is a bisimulation, we need to retain the congruence property after matching reduction steps. Since we can apply only one context before applying the bisimulation hypothesis, this single context must be able to emulate the behavior of any other context, to be selected after the

reductions. We call such a context a "universal context". The details of the construction appear in [13].

The first step is to define an integer representation for every process P, written $[\![P]\!]$, and to build an interpreter D_e that takes (1) an integer representation $[\![P]\!] \in \mathbb{N}$ and (2) the encoding of an evaluation environment ρ that binds all the free names $\mathrm{fv}(P)$. The next lemma relates the source process P to its interpreted representation; this result is not surprising, inasmuch as the join calculus has well enough expressive power. Some care is required to restrict the types that may appear at P's interface. The lemma is established using a labeled bisimulation, as defined in Section 4. We omit the details of the encoding and of the interpreter.

Lemma 30. *Let Σ be a finite set of types closed by decomposition. There is a definition D_e such that, for every process P whose free variables can be typed in Σ and such that $\mathrm{fv}(P) \cap \{e, \rho\} = \emptyset$, and for every environment ρ such that $\forall x \in \mathrm{fv}(P), \rho([\![x]\!]) = x$, we have $\mathtt{def}\ D_e\ \mathtt{in}\ e\langle [\![P]\!], \rho\rangle \approx P$.*

The second step is to reduce quantification over all contexts first to quantification over all processes (using a specific context that forwards the messages), then to quantification over all integers (substituting the interpreter for the process). Finally, the universal context uses internal choice to select any integer, then either manifest this choice using integer barbs, or run the interpreter on this input with an environment ρ that manifests every barb using integer barbs. At each stage, the disappearance of specific integer barbs allows the bisimulation to keep track of the behavior of the context.

Lemma 31 (Universal Context). *Let S be a finite set of names There is an evaluation context $U_S[\cdot]$ such that, for all processes P and Q with $\mathrm{fv}(P) \cup \mathrm{fv}(Q) \subseteq S$, we have $U_S[P] \dot{\approx}_2 U_S[Q]$ implies $P \approx Q$*

Combining these encodings, we eventually obtain the difficult bisimulation property of $\dot{\approx}_2^\circ$, hence $\dot{\approx}_2^\circ = \approx$ and finally $\dot{\approx}^\circ = \approx$.

3.5 Summary: a hierarchy of equivalences

Anticipating on the labeled semantics in the next section, we summarize our results on equivalences in Figure 3. Each tier in the hierarchy correspond to a notion of equivalence finer than the lower tiers. When stating and proving equational properties in a reduction-based setting, this hierarchy provides some guidance. For instance, the same proof may involve lemmas expressed as equivalences much finer (and easier to establish) than the final result. Conversely, a counter-example may be easier to exhibit at a tier lower than required.

The reader may be interested in applications of these techniques to establish more challenging equations. For detailed applications, we refer for instance to [1] for a detailed proof of the security of a distributed, cryptographic implementation of the join calculus, to [13] for a series of fully abstract encodings between variants of the join calculus, and to [17] for the correctness proof of an implementation of Ambients in JoCaml using coupled simulations.

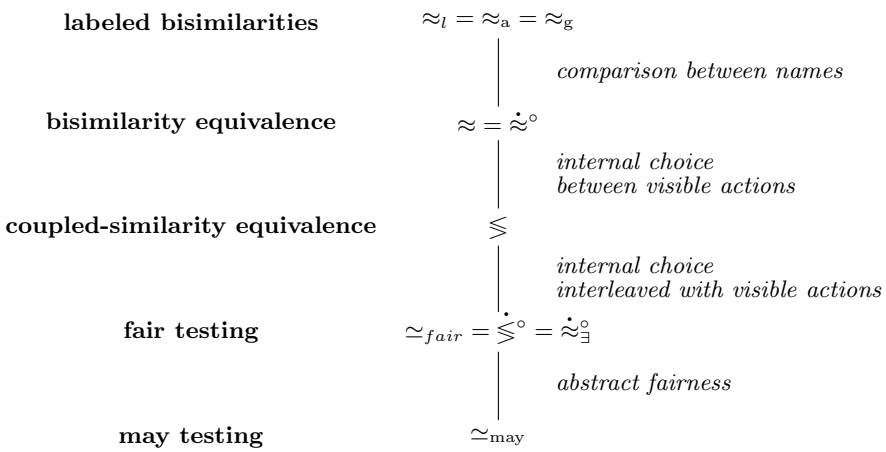


Figure 3. A hierarchy of equivalences for the join calculus

4 Labeled semantics

Labeled transition systems traditionally provide useful semantics for process calculi, by incorporating detailed knowledge of the operational semantics in their definitions and their proof techniques. Seen as auxiliary semantics for a reduction-based calculus, they offer several advantages, such as purely coinductive proofs and abstract models (e.g. synchronization trees). On the other hand, they are specific to the calculus at hand, and they may turn out to be too discriminating for asynchronous programming.

We present two variants of labeled semantics for the join calculus, and relate their notions of labeled bisimilarities to observational equivalence, thus comparing the discriminating power of contexts and labels. We refer to [16] for a more detailed presentation.

4.1 Open syntax and chemistry

In the spirit of the trace semantics given in Section 2.2, we introduce a refined semantics—the *open* RCHAM—that makes explicit the interactions with an abstract environment. Via these interactions, the environment can receive locally-defined names of the process when they are emitted on free names, and the environment can also emit messages on these names. We call these interactions *extrusions* and *intrusions*, respectively. To keep track of the defined names that are visible from the environment, definitions of the join calculus are marked with their extruded names when extrusions occur. In turn, intrusions are allowed only on names that are marked as extruded. The refined syntax for the join calculus has processes of the form $\mathtt{def}_S\ D\ \mathtt{in}\ P$, where S is the set of names defined by D and extruded to the environment. Informally, extruded names represent constants in the input interface of the process.

$$
\begin{array}{lll}
A, B ::= & & \text{open processes} \\
& P & \text{plain process} \\
\| & \mathtt{def}_S\ D\ \mathtt{in}\ P & \text{open definition} \\
\| & A \mid B & \text{parallel composition}
\end{array}
$$

Figure 4. Syntax for the open join calculus

As a first example, consider the process $\mathtt{def}_\emptyset\ x\langle\rangle \triangleright y\langle\rangle\ \mathtt{in}\ z\langle x\rangle$. The interface contains no extruded name and two free names y, z. The message $z\langle x\rangle$ can be consumed by the environment, thus exporting x:

$$\mathtt{def}_\emptyset\ x\langle\rangle \triangleright y\langle\rangle\ \mathtt{in}\ z\langle x\rangle \xrightarrow{\{x\}\overline{z}\langle x\rangle} \mathtt{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathtt{in}\ \mathbf{0}$$

Once x is known by the environment, it cannot be considered local anymore—the environment can emit on x—, but it is not free either—the environment cannot modify or extend its definition. A new transition is enabled:

$$\mathtt{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathtt{in}\ \mathbf{0} \xrightarrow{x\langle\rangle} \mathtt{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathtt{in}\ x\langle\rangle$$

Now the process can input more messages on x, and also perform the two transitions below to consume the message on x and emit a message on y:

$$
\begin{array}{rcl}
\mathtt{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathtt{in}\ x\langle\rangle & \rightarrow & \mathtt{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathtt{in}\ y\langle\rangle \\
& \xrightarrow{\{\}\overline{y}\langle\rangle} & \mathtt{def}_{\{x\}}\ x\langle\rangle \triangleright y\langle\rangle\ \mathtt{in}\ \mathbf{0}
\end{array}
$$

We now extend the RCHAM of [14] with extrusions, intrusions, and explicit bookkeeping of extruded names.

Definition 32. Open chemical solutions*, ranged over by $\mathcal{S}, \mathcal{T}, \ldots$, are triples $(\mathcal{D}, S, \mathcal{A})$, written $\mathcal{D} \vdash_S \mathcal{A}$, where $\mathcal{D}$ is a multiset of definitions, S is a subset of the names defined in $\mathcal{D}$, and $\mathcal{A}$ is a multiset of open processes with disjoint sets of extruded names that are not defined in $\mathcal{D}$.*

The interface of an open solution $\mathcal{S}$ consists of two disjoint sets of free names $\mathrm{fv}(\mathcal{S})$ and extruded names $\mathrm{xv}(\mathcal{S})$, defined in Figure 5. Functions $\mathrm{dv}(\cdot)$, $\mathrm{fv}(\cdot)$, and $\mathrm{xv}(\cdot)$ are extended to multisets of terms by taking unions for all terms in the multisets.

The chemical rules for the open RCHAM are given in Figure 6; they define families of transitions between open solutions $\rightleftharpoons$, $\rightarrow$, and $\xrightarrow{\alpha}$ where α ranges over labels of the form $S\overline{x}\langle\widetilde{v}\rangle$ and $x\langle\widetilde{v}\rangle$.

The structural rules and rule REACT are unchanged, but they now apply to open solutions. Rule STR-DEF performs the bookkeeping of exported names, and otherwise enforces a lexical scoping discipline with scope-extrusion for any name that is not exported. When applied to open solutions, these structural rules capture the intended meaning of extruded names: messages sent on extruded

In join patterns:

$\mathrm{rv}(x\langle\widetilde{v}\rangle) = \{u \in \widetilde{v}\}$ $\quad \mathrm{dv}(x\langle\widetilde{v}\rangle) = \{x\}$

$\mathrm{rv}(J \mid J') = \mathrm{rv}(J) \uplus \mathrm{rv}(J')$ $\quad \mathrm{dv}(J \mid J') = \mathrm{dv}(J) \uplus \mathrm{dv}(J')$

In definitions:

$\mathrm{dv}(J \triangleright P) = \mathrm{dv}(J)$ $\quad \mathrm{fv}(J \triangleright P) = \mathrm{dv}(J) \cup (\mathrm{fv}(P) \setminus \mathrm{rv}(J))$

$\mathrm{dv}(D \wedge D') = \mathrm{dv}(D) \cup \mathrm{dv}(D')$ $\quad \mathrm{fv}(D \wedge D') = \mathrm{fv}(D) \cup \mathrm{fv}(D')$

In processes:

$\mathrm{fv}(A \mid A') = (\mathrm{fv}(A) \cup \mathrm{fv}(A')) \setminus (\mathrm{xv}(A) \uplus \mathrm{xv}(A'))$ $\quad \mathrm{fv}(\mathbf{0}) = \emptyset$

$\mathrm{xv}(A \mid A') = \mathrm{xv}(A) \uplus \mathrm{xv}(A')$ $\quad \mathrm{xv}(\mathbf{0}) = \emptyset$

$\mathrm{fv}(\mathtt{def}_S\ D\ \mathtt{in}\ A) = (\mathrm{fv}(D) \cup \mathrm{fv}(A)) \setminus (\mathrm{dv}(D) \uplus \mathrm{xv}(A))$ $\quad \mathrm{fv}(x\langle\widetilde{v}\rangle) = \{x, \widetilde{v}\}$

$\mathrm{xv}(\mathtt{def}_S\ D\ \mathtt{in}\ A) = S \uplus \mathrm{xv}(A)$ $\quad \mathrm{xv}(x\langle\widetilde{v}\rangle) = \emptyset$

In chemical solutions:

$\mathrm{fv}(\mathcal{D} \vdash_S \mathcal{A}) = (\mathrm{fv}(\mathcal{D}) \cup \mathrm{fv}(\mathcal{A})) \setminus (\mathrm{dv}(\mathcal{D}) \uplus \mathrm{xv}(\mathcal{A}))$

$\mathrm{xv}(\mathcal{D} \vdash_S \mathcal{A}) = S \uplus \mathrm{xv}(\mathcal{A})$

Figure 5. Scopes in the open join calculus

Str-Null	$\vdash_S \mathbf{0}$	$\rightleftharpoons$	$\vdash_S$
Str-Par	$\vdash_S A \mid A'$	$\rightleftharpoons$	$\vdash_S A,\ A'$
Str-Top	$\top \vdash_S$	$\rightleftharpoons$	$\vdash_S$
Str-And	$D \wedge D' \vdash_S$	$\rightleftharpoons$	$D,\ D' \vdash_S$
Str-Def	$\vdash_S \mathtt{def}_{S'}\ D\ \mathtt{in}\ A$	$\rightleftharpoons$	$D\sigma \vdash_{S \uplus S'} A\sigma$
React	$J \triangleright P \vdash_S J\rho$	$\rightarrow$	$J \triangleright P \vdash_S P\rho$
Ext	$\vdash_S x\langle\widetilde{y}\rangle$	$\xrightarrow{S'\widetilde{x}\langle\widetilde{y}\rangle}$	$\vdash_{S \cup S'}$
Int	$\vdash_{S \cup \{x\}}$	$\xrightarrow{x\langle\widetilde{y}\rangle}$	$\vdash_{S \cup \{x\}} x\langle\widetilde{y}\rangle$

Side conditions on the reacting solution $\mathcal{S} = (\mathcal{D} \vdash_S \mathcal{A})$:
in Str-Def, σ substitutes distinct fresh names for $\mathrm{dv}(D) \setminus S'$;
in React, ρ substitutes names for $\mathrm{rv}(J)$;
in Ext, the name x is free, and $S' = \{\widetilde{y}\} \cap (\mathrm{dv}(\mathcal{D}) \setminus S)$;
in Int, the names $\widetilde{y}$ are either free, or fresh, or extruded.

Figure 6. The open RCHAM

names can be moved inside or outside their defining process. For instance, we have the structural rearrangement

$$\vdash_S \ x\langle\widetilde{v}\rangle \mid \mathtt{def}_{S'} \ D \ \mathtt{in} \ A \rightleftharpoons \vdash_S \ \mathtt{def}_{S'} \ D \ \mathtt{in} \ (x\langle\widetilde{v}\rangle \mid A)$$

for any extruded name x, and as long as the names in $\widetilde{v}$ are not captured by D ($\{\widetilde{v}\} \cap \mathrm{dv}(D) \subseteq S'$).

In addition, rules Ext and Int model interaction with the context. According to rule Ext, messages emitted on free names can be received by the environment; these messages export any defined name that was not previously known to the environment, thus causing the scope of its definition to be opened. This is made explicit by the set S' in the label of the transition $\xrightarrow{S'\overline{x}\langle\widetilde{v}\rangle}$. Names in S' must be distinct from any name that appears in the interface before the transition; once these names have been extruded, they cannot be α-converted anymore, and behave like constants. Our rule resembles the Open rule for restriction in the pi calculus [32], with an important constraint due to locality: messages are either emitted on free names, to be consumed by Ext, or on names defined in the open solution, to be consumed by React.

The rule Int enables the intrusion of messages on exported names names. It can be viewed as a disciplined version of one of the two Input rules proposed by Honda and Tokoro for the asynchronous pi calculus, which enables the intrusion of any message [21]. The side condition of Int requires that intruded messages do not clash with local names of processes. (More implicitly, we may instead rely on the silent α-conversion on those local names; this is the original meaning of "intrusion" in [32].)

4.2 Observational equivalences on open terms

The notions of reduction-based equivalence defined in sections 2 and 3 are easily extended to open processes, with the same definitions and the additional requirement that related processes have the same exported names. (Indeed, it makes little sense to compare processes with incompatible interfaces such as $\mathbf{0}$ and the open deadlocked solution $\mathtt{def}_{\{y\}} \ x\langle\rangle \mid y\langle\rangle \triangleright \mathtt{in} \ \mathbf{0}$.) Context-closure properties are also easily extended to take into account open contexts. Note that, whenever we apply a context, we implicitly assume that the resulting open process is well-formed. Finally, extrusions and strong barbs are in direct correspondence—we have $A \downarrow_x$ if and only if $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} A'$ for any S,$\widetilde{v}$, and A'.

In fact, the open syntax provides convenient notations to give selective give access to some defined names in the processes being compared, but it does not introduce any interesting new equation. Consider for instance bisimilarity equivalence on open terms:

Lemma 33. *For all processes P_1, P_2 and definitions D_1, D_2, let $\widetilde{x}$ be a tuple of names defined in both D_1 and D_2, and let plug be a fresh name. The three following statements are equivalent:*

1. $\mathtt{def}_{\{\widetilde{x}\}} \ D_1 \ \mathtt{in} \ P_1 \ \approx \ \mathtt{def}_{\{\widetilde{x}\}} \ D_2 \ \mathtt{in} \ P_2$

2. $\texttt{def}\ D_1\ \texttt{in}\ P_1 \mid plug\langle\widetilde{x}\rangle\ \approx\ \texttt{def}\ D_2\ \texttt{in}\ P_2 \mid plug\langle\widetilde{x}\rangle$
3. *for all* D *and* P *such that* $\mathrm{fv}(\texttt{def}\ D\ \texttt{in}\ P) \cap (\mathrm{dv}(D_1) \cup \mathrm{dv}(D_2)) \subseteq \{\widetilde{x}\}$ *and* $\mathrm{dv}(D) \cap (\mathrm{dv}(D_1) \cup \mathrm{dv}(D_2)) = \emptyset$*, we have* $\texttt{def}\ D \wedge D_1\ \texttt{in}\ P \mid P_1\ \approx\ \texttt{def}\ D \wedge D_2\ \texttt{in}\ P \mid P_2$.

The first formulation is the most compact; it relies on open terms. Instead, the second formulation makes explicit the communication of extruded names to the environment using a message on a fresh name *plug*; the third formulation is closed by application of evaluation contexts, and is often used in direct proofs of bisimilarity equivalence (see for instance, [1]).

4.3 Labeled bisimulation

By design, the open join calculus can also be equipped with the standard notion of labeled bisimilarity:

Definition 34. *A relation* $\mathcal{R}$ *on open processes is a* labeled simulation *if, whenever* $A\ \mathcal{R}\ B$*, we have*

1. *if* $A \to A'$ *then* $B \to^* B'$ *and* $A'\ \mathcal{R}\ B'$*;*
2. *if* $A \xrightarrow{\alpha} A'$ *then* $B \to^* \xrightarrow{\alpha} \to^* B'$ *and* $A'\ \mathcal{R}\ B'$*,*
   for all labels α *of shape* $x\langle\widetilde{v}\rangle$ *or* $S\overline{x}\langle\widetilde{v}\rangle$ *such that* $\mathrm{fv}(B) \cap S = \emptyset$*.*

A relation $\mathcal{R}$ *is a* labeled bisimulation *when both* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are labeled simulations.* Labeled bisimilarity $\approx_l$ *is the largest labeled bisimulation.*

The simulation clause for intrusions makes weak bisimulation sensitive to input interfaces: $A \approx_l B$ implies $\mathrm{xv}(A) = \mathrm{xv}(B)$. The simulation clause for extrusion does not consider labels whose set of extruded names S clashes with the free names of B, inasmuch as these transitions can never be simulated; this standard technicality does not affect the intuitive discriminating power of bisimulation, because names in S can be α-converted before the extrusion.

As opposed to contextual equivalences, it is possible to tell whether two processes are weakly bisimilar by comparing their labeled synchronization trees, rather than reasoning on their possible contexts. For example, $x\langle u\rangle \not\approx_l x\langle v\rangle$ because each process performs an extrusion with distinct labels. Likewise, $x\langle y\rangle \not\approx_l \texttt{def}\ z\langle u\rangle \triangleright y\langle u\rangle\ \texttt{in}\ x\langle z\rangle$ because the first process emits a free name (label $\overline{x}\langle y\rangle$) while the latter emits a local name that gets extruded (label $\{z\}\overline{x}\langle z\rangle$).

Besides, a whole range of "up to proof techniques" is available to reduce the size of the relation to exhibit when proving labeled bisimilarities [30,33,39,41]. For instance, one can reason up to other bisimilarities, or up to the restriction of the input interface.

While its definition does not mention contexts, labeled bisimilarity is closed by application of any context:

Theorem 35. *Weak bisimilarity is a congruence.*

The proof is almost generic to mobile process calculi in the absence of external choice (see, e.g., [21,6] for the asynchronous pi calculus); it relies on two simpler closure properties: $\approx_l$ is closed by application of evaluation contexts, and $\approx_l$ is closed by renamings. We refer to [16] for the details.

As an immediate corollary, we can place labeled bisimilarity in our hierarchy of equivalence, and justify its use as an auxiliary proof technique for observational equivalence: we have that $\approx_l$ is a reduction-based bisimulation that respects all barbs and that is closed by application of contexts, hence $\approx_l \subseteq \approx$. This inclusion is strict, as can be seen from the paradigmatic example of bisimilarity equivalence:

$$x\langle z\rangle \approx \texttt{def}\ u\langle v\rangle \triangleright z\langle v\rangle\ \texttt{in}\ x\langle u\rangle$$

That is, emitting a free name z is the same as emitting a bound name u that forwards all the messages it receives to z, because the extra internal move for every use of u is not observable. On the contrary, labeled bisimilarity separates these two processes because their respective extrusion labels reveal that z is free and u is extruded. Since the contexts of the open join calculus cannot identify names in messages, more powerful contexts are required to reconcile the two semantics (see Section 4.5).

4.4 Asynchronous bisimulation

In order to prove that two processes are bisimilar, a large candidate bisimulation can be a nuisance, as it requires the analysis of numerous transition cases. Although they are not necessarily context-closed, labeled bisimulations on open chemical solutions are typically rather large. For example, a process with an extruded name has infinitely many derivatives even if no "real" computation is ever performed. Consider the equivalence:

$$\texttt{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright P\ \texttt{in}\ z\langle x\rangle \approx_l \texttt{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright Q\ \texttt{in}\ z\langle x\rangle$$

These two processes are bisimilar because their join-pattern cannot be triggered, regardless of the messages the environment may send on x. Still, one is confronted with infinite models on both sides, with a distinct chemical solution for every multiset of messages that have been intruded on x so far. This problem with labeled bisimulation motivates an alternative formulation of labeled equivalence.

The join open RCHAM We modify the open RCHAM by allowing inputs only when they immediately trigger a guarded process. For example, the two processes above become inert after an extrusion $\{x\}\overline{z}\langle x\rangle$, hence trivially bisimilar. If we applied this refinement with the same labels for input as before, however, we would obtain a dubious result. The solution $x\langle\rangle \mid y\langle\rangle \mid z\langle\rangle \triangleright P \vdash_{\{x,y\}} z\langle\rangle$ can progress by first inputing two messages $x\langle\rangle$ and $y\langle\rangle$, then performing a silent step that consumes these two messages together with the local message $z\langle\rangle$ already in the solution. Yet, neither $x\langle\rangle$ nor $y\langle\rangle$ alone can trigger the process P, and

$$\textsc{React-Int} \qquad J \triangleright P \vdash_S M \xrightarrow{M'} J \triangleright P \vdash_S P\rho$$

Side conditions:
ρ substitute names for $\mathrm{rv}(J)$, $J\rho \equiv M \mid M'$, and $\mathrm{dv}(M') \subseteq S$.

The rules Str-(Null,Par,And,Def) and Ext are the same as in Figure 6.

Figure 7. The join open RCHAM

therefore this solution would become inert, too. This suggests the use of *join*-inputs on x and y in transitions such as

$$x\langle\rangle \mid y\langle\rangle \mid z\langle\rangle \triangleright P \vdash_{\{x,y\}} z\langle\rangle \xrightarrow{x\langle\rangle \mid y\langle\rangle} x\langle\rangle \mid y\langle\rangle \mid z\langle\rangle \triangleright P \vdash_{\{x,y\}} P$$

On the other hand, the solution $x\langle\rangle \mid y\langle\rangle \mid z\langle\rangle \triangleright P \vdash_{\{x\}} z\langle\rangle$ is truly inert, since the environment has no access to y, and thus cannot trigger P. In this case, our refinement suppresses all input transitions.

The join open RCHAM is defined in Figure 7 as a replacement for the intrusion rule. In contrast with rule Int of Figure 6, the new rule React-Int permits the intrusion of messages only if these messages are immediately used to trigger a process. This is formalized by allowing labels M' that are parallel compositions of messages. If the solution contains a complementary process M such that the combination $M \mid M'$ matches the join-pattern of a reaction rule, then the transition occurs and triggers this reaction rule. As for Int, we restrict intrusions in M' to messages on extruded names.

We identify intrusions in the case $M' = \mathbf{0}$ with silent steps; the rule React is thus omitted from the new chemical machine. Nonetheless, we maintain the distinction between internal moves and proper input moves in the discussion. In the sequel, we shall keep the symbol $\xrightarrow{\alpha}$ for the open RCHAM and use $\xrightarrow{\alpha}_{\mathrm{J}}$ for the join open RCHAM; we may drop the subscript J when no ambiguity can arise.

Each open process now has two different models: for instance, the process $\mathtt{def}_{\{x\}}\ x\langle\rangle \mid y\langle\rangle \triangleright P\ \mathtt{in}\ \mathbf{0}$ has no transition in the join open RCHAM, while it has infinite series of transitions $\xrightarrow{x\langle\rangle}\xrightarrow{x\langle\rangle}\xrightarrow{x\langle\rangle}\cdots$ in the open RCHAM. A comparison between the two transition systems yields the following correspondence between their intrusions:

Proposition 36. *Let A be an open process.*

1. *If $A \xrightarrow{x_1\langle\widetilde{v}_1\rangle \mid \cdots \mid x_n\langle\widetilde{v}_n\rangle}_{\mathrm{J}} B$, then $A \xrightarrow{x_1\langle\widetilde{v}_1\rangle} \cdots \xrightarrow{x_n\langle\widetilde{v}_n\rangle} B$.*
2. *If $A \mid x\langle\widetilde{u}\rangle \xrightarrow{M}_{\mathrm{J}} B$ and $x \in \mathrm{xv}(A)$, then*
   (a) either $A \xrightarrow{M}_{\mathrm{J}} A'$ with $A' \mid x\langle\widetilde{u}\rangle \equiv B$;
   (b) or $A \xrightarrow{M \mid x\langle\widetilde{u}\rangle}_{\mathrm{J}} B$.

Accordingly, we adapt the definition of labeled bisimulation (Definition 34) to the new join open RCHAM. Consider the two processes:

$$P \stackrel{\mathrm{def}}{=} \mathtt{def}\ x\langle\rangle \triangleright a\langle\rangle \wedge a\langle\rangle \mid y\langle\rangle \triangleright R\ \mathtt{in}\ z\langle x, y\rangle$$
$$Q \stackrel{\mathrm{def}}{=} \mathtt{def}\ x\langle\rangle \mid y\langle\rangle \triangleright R\ \mathtt{in}\ z\langle x, y\rangle$$

and assume $a \notin \mathrm{fv}(R)$. With the initial open RCHAM, the processes P and Q are weakly bisimilar. With the new join open RCHAM and the same definition of weak bisimulation, this does not hold because P can input $x\langle\rangle$ after emitting on z while Q cannot. But if we consider the bisimulation that uses join-input labels instead of single ones, Q can input $x\langle\rangle \mid y\langle\rangle$ while P cannot, and P and Q are still separated. It turns out that labeled bisimulation discriminates too much in the join open RCHAM.

In order to retain an asynchronous semantics, labeled bisimulation must be relaxed, so that a process may simulate a REACT-INT transition even if it does not immediately consume all its messages. This leads us to the following definition:

Definition 37. *A relation $\mathcal{R}$ is an* asynchronous simulation *if, whenever $A\ \mathcal{R}\ B$, we have*

1. *if $A \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} A'$ then $B \to^* \xrightarrow{S\overline{x}\langle\widetilde{v}\rangle} \to^* B'$ and $A'\ \mathcal{R}\ B'$ for all labels $S\overline{x}\langle\widetilde{v}\rangle$ such that $\mathrm{fv}(B) \cap S = \emptyset$;*
2. *if $A \xrightarrow{M} A'$, then $B \mid M \to^* B'$ and $A'\ \mathcal{R}\ B'$;*
3. $\mathrm{xv}(A) = \mathrm{xv}(B)$.

A relation $\mathcal{R}$ is an asynchronous bisimulation *when both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are asynchronous simulations.* Asynchronous bisimilarity $\approx_{\mathrm{a}}$ *is the largest asynchronous bisimulation.*

In the definition above, the usual clause for silent steps is omitted (it is subsumed by the clause for intrusions with $M = \mathbf{0}$). On the other hand, a clause explicitly requires that related solutions have the same extruded names.

Asynchronous bisimilarity and labeled bisimilarity do coincide. This validates asynchronous bisimulation as an efficient proof technique.

Theorem 38. $\approx_{\mathrm{a}} = \approx_l$.

To conclude our discussion on variants of labeled bisimulations, let us mention *ground bisimulation*, which is obtained by restricting the intrusions to labels that convey fresh names. As first observed in the pi calculus [21,6,8], asynchrony brings another interesting property as regards the number of transitions to consider: the ground variant of bisimilarities coincide with the basic one. This property also holds in the join calculus, thus providing proof techniques with, for every chemical solution, exactly one intrusion per extruded name when using labeled bisimulation, and one intrusion per "active" partial join-pattern when using asynchronous bisimulation.

4.5 The discriminating power of name comparison

Labeled bisimilarity is finer than (reduction-based, barbed) bisimilarity equivalence and, as in other process calculi, these two semantics coincide only if we add

an operator for name comparisons [23,6]. In this section, we extend the syntax of the join calculus accordingly, in the same style as [32].

$$A \stackrel{\text{def}}{=} \ldots \mid [x{=}y]\, A \qquad\qquad P \stackrel{\text{def}}{=} \ldots \mid [x{=}y]\, P$$

We also extend our chemical machines with a new reduction rule.

$$\textsc{Compare} \qquad\qquad \vdash_S [x{=}x]\, A \;\rightarrow\; \vdash A$$

A technical drawback of this extension is that renamings do not preserve bisimilarity anymore. For instance, $\mathbf{0} \approx_l [x{=}y]\, x\langle\rangle$, while after applying the renaming $\{^x/_y\}$, $\mathbf{0} \not\approx_l [x{=}x]\, x\langle\rangle$. Accordingly, labeled bisimilarity is not a congruence anymore. For instance, the context $C[\cdot] \stackrel{\text{def}}{=} \texttt{def}\ z\langle x, y\rangle \triangleright [\cdot]\ \texttt{in}\ z\langle u, u\rangle$ separates $\mathbf{0}$ and $[x{=}y]\, x\langle\rangle$. We consider equivalences that are closed only by application of evaluation contexts.

In the presence of comparisons, we still have:

Lemma 39. *Labeled bisimilarity is closed by application of evaluation contexts*

As regards observational equivalence, we let *bisimilarity equivalence* $\approx_{\text{be}}$ be the largest barb-preserving bisimulation in the open join calculus with name comparison that is closed by application of plain evaluation contexts. Bisimilarity equivalence now separates $x\langle z\rangle$ from $\texttt{def}\ u\langle v\rangle \triangleright z\langle v\rangle\ \texttt{in}\ x\langle u\rangle$ by using the context $\texttt{def}\ x\langle y\rangle \triangleright [y{=}z]\, a\langle\rangle\ \texttt{in}\ [\cdot]$, and labeled bisimilarity clearly remains finer than bisimilarity equivalence. More interestingly, the converse inclusion also holds:

Theorem 40. *With name comparison, we have* $\approx_{\text{be}} = \approx_l$.

To establish the theorem, it suffices to show that, for every label, there is an evaluation context that specifically "captures" the label. Intrusions are very easy, since it suffices to use the parallel context $x\langle\widetilde{y}\rangle \,|[\cdot]$. Extrusions are more delicate, inasmuch as a context of the join calculus must define a name in order to detect an output transition on that name; this case is handled by creating a permanent relay for all other messages on that name. Without additional care, this relay can be detected by name matching, so we use instead a family of contexts that separate two aspects of a name. For every name $x \in \mathcal{N}$, we let

$$R_x[\cdot] \stackrel{\text{def}}{=} \texttt{def}\ x\langle\widetilde{y}\rangle \triangleright x'\langle\widetilde{y}\rangle\ \texttt{in}\ v_x\langle x\rangle \,|[\cdot]$$

where the length of $\widetilde{y}$ matches the arity of x. Assuming $x \in \text{fv}(A)$, the process $R_x[A]$ uses x' as a free name instead of x, and forwards all messages from x to x'. The context should still be able to discriminate whether the process sends the name x or not by using name matching; this extra capability is supplied in an auxiliary message $v_x\langle x\rangle$. The next lemma captures the essential property of $R_x[\cdot]$:

Lemma 41 (Accommodating the extrusions). *For all open processes A and B such that $x \notin \text{xv}(A) \cup \text{xv}(B)$ and x', v_x are not in the interface of A and B, we have $A \approx_{\text{be}} B$ if and only if $R_x[A] \approx_{\text{be}} R_x[B]$.*

Informally, the contexts $R_x[\cdot]$ are the residuals of contexts that test for labels of the form $\{S\}\overline{x}\langle\widetilde{y}\rangle$. Once we have a context for every label, we easily prove that $\approx_{\mathrm{be}}$ is a labeled bisimulation. Remark that the proof of the theorem would be much harder if we were using the other notion of bisimilarity equivalence (see Section 3.1), because we would have to characterize the whole synchronization tree at once in a single context, instead of characterizing every label in isolation. This explains why many similar results in the literature apply only to processes with image-finite transitions.

5 Distribution and mobility

Although distributed programming is the main purpose of the join calculus, the distribution of resources has been kept implicit so far. As we described its semantics, we just argued that the join calculus had enough built-in locality to be implemented in a distributed, asynchronous manner.

This section gives a more explicit account of distributed programming. We extend the join calculus with *locations* and primitives for mobility. The resulting calculus allows us to express mobile agents that can move between physical sites. Agents are not only programs but core images of running processes with their communication capabilities and their internal state. Inevitably, the resulting *distributed join calculus* is a bit more complex than the core calculus of Section 1.

Intuitively, a location resides on a physical site, and contains a group of processes and definitions. We can move atomically a location to another site. We represent mobile agents by locations. Agents can contain mobile sub-agents represented by nested locations. Agents move as a whole with all their current sub-agents, thereby locations have a dynamic tree structure. Our calculus treats location names as first class values with lexical scopes, as is the case for channel names; the scope of every name may extend over several locations, and may be dynamically extended as the result of message passing or agent mobility. A location controls its own moves, and can move towards another location by providing the name of the target location, which would typically be communicated only to selected processes.

Since we use the distributed join calculus as the core of a programming language (as opposed to a specification language), the design for mobility is strongly influenced by implementation issues. Our definition of atomic reduction steps attempts to strike some balance between expressiveness and realistic concerns. Except for the (partial) reliable detection of physical failures, the refined operational semantics has been faithfully implemented [18,12]. In these notes, however, we omit any specific discussion of these implementations.

5.1 Distributed mobile programming

We introduce the language with a few examples that assume the same approach to runtime distribution as in JoCaml. Execution occurs among several machines,

which may dynamically join or quit the computation; the runtime support consists of several system-level processes that communicate using TCP/IP. Processes and definitions can migrate from one machine to another but, at any given point, every process and definition of the language is running at a single machine.

From an asynchronous viewpoint, and in the absence of partial failures, *locality is transparent*. Programs can be written independently of their runtime distribution, and their visible results do not depend on their localization. Indeed, it is "equivalent" to run processes P and Q at different machines, or to run the compound process $P \mid Q$ at a single machine. In particular, the scopes for channel names and other values do not depend on their localization: whenever a channel appears in a process, it can be used to form messages (using the name either as the address, or as the message contents) without knowing whether this channel is locally- or remotely-defined.

Of course, locality matters in some circumstances: side effects such as printing values on the local terminal depend on the current machine; besides, efficiency can be affected as message-sending over the network takes much longer than local calls; finally, the termination of some underlying runtime will affect all its local processes. For all these reasons, *locality is explicitly controlled in the language*; this locality can be adjusted using migration. In contrast, resources such as definitions and processes are not silently relocated or replicated by the system.

In JoCaml, programs being run on different machines do not initially share any channel name; therefore, they would normally not be able to interact with one another. To bootstrap a distributed computation, it is necessary to exchange a few names; this is achieved using a built-in library called the name server. Once this is done, these names can be used to communicate some more names and to build more complex communication patterns. The interface of the name server consists of two functions to register and look up arbitrary values in a "global table" indexed by plain strings (JoCaml actually performs some dynamic type checking at this stage). For instance, the process on the left below defines a local name *cos* and registers it to the name server:

$$\texttt{def } cos(x) = 1 - x^2/2 \texttt{ in} \quad NS.register(\text{"}cos\text{"}, cos)$$

$$\texttt{def } cos = NS.lookup(\text{"}cos''\text{) } \texttt{in} \quad print(cos(0.1)); \ldots$$

Using the same key "*cos*", a remote program (such as the one on the right) can obtain the name *cos* then perform remote calls. The computation takes place on the machine that defines *cos* (in the example, the machine hosting the program on the left).

More explicitly, the program defining *cos* may define a *named location* that wraps this function definition, and may also export the location name under the key "here":

$$\begin{array}{l}\texttt{def } here[\\ \quad cos(x) = 1 - x^2/2 \quad :\\ \quad NS.register(\text{"}cos''\text{}, cos); \ldots\ ] \texttt{ in}\\ NS.register(\text{"}here''\text{}, here); \ldots\end{array}$$

The location definition does not affect *cos*, which can still be called locally or remotely. In addition to remote access to *cos*, another program can now obtain the location name *here*, create locally a mobile sub-location—its "agent"—and relocate this agent to *here*. This makes sense, for instance, if the agent implements processes that often call *cos*. The code on the client program may be:

```
def f(machine) ▷
   agent[
      go machine;
      def cos = NS.lookup("cos") in
      def sum(s, n) = if n = 0 then s else sum(s + cos(n), n − 1) in
      return sum(0, 10) ]
print(f(NS.lookup("here")))); . . .
```

The new statement "go *location*; *P*" causes the enclosing location to migrate as a whole towards *location*'s machine before executing the following process *P*. In the program above, location *agent* migrates with its running process towards the machine that hosts *here* and *cos*, locally retrieves *cos* using the name server and runs some computation, then eventually returns a single result to *f*'s caller on the client machine.

A more complex example is an "applet server" that provides a function to create new instances of a library *at a remote location* provided by the caller. To this end, the server creates a mobile agent that wraps the instance of the library, migrates to the target location, and delivers the library interface once there. For instance, the code below implements a "one-place-buffer" library with some log mechanism:

```
def newOnePlaceBuffer(there) ▷
   def log(s) = print("the buffer is " + s) in
   def applet[
      go there;
      def put(v) | empty⟨⟩ ▷ log("full"); (full⟨v⟩ | return)
        ∧ get() | full⟨v⟩ ▷ log("empty"); (empty⟨⟩ | return v) in
      empty⟨⟩ | return put, get to newBuf ] in
   log("created"); in
NS.register("applet", newOnePlaceBuffer); . . .
```

A simple applet client may be:

```
def newBuf = NS.lookup("newBuffer") in
def here[
   def put, get = newBuf(here) in
   put(1); . . . ] in . . .
```

In contrast with plain code mobility, the new applet can keep static access to channels located at the applet server; in our case, every call to the one-place buffer is local to the applet client, but also causes a remote log message to be sent to the applet server.

5.2 Computing with locations

We now model locations and migrations as a refinement of the RCHAM. We proceed in two steps. First, we partition processes and definitions into several local chemical solutions. This flat model suffices for representing both local computation on different sites and global communication between them. Then, we introduce some more structure to account for the creation and the mobility of local solutions: we attach *location names* to solutions, and we organize them as a tree of nested locations. The refined syntax and chemical semantics appear in figures 8 and 9.

Distributed machines A distributed reflexive chemical machine (DRCHAM) is a multiset of RCHAMs. We write the global state of a DRCHAM as several *local solutions* $\mathcal{D} \vdash^{\alpha} \mathcal{P}$ connected by a commutative-associative operator $\|$ that represents distributed composition. Each local solution has a distinct label α—we will detail below the role of these labels.

Locally, every solution $\mathcal{D} \vdash^{\alpha} \mathcal{P}$ within a DRCHAM evolves as before, according to the chemical rules given for the join calculus in Figure 2. Technically, the chemical context law is extended to enable computation in any local solution, and the side condition in STR-DEF requires that globally-fresh names be substituted for locally-defined names.

Two solutions can interact by using a new rule COMM that models global communication. This rule states that a message emitted in a given solution α on a channel name x that is remotely defined can be forwarded to the solution β that contains the definition of x. Later on, this message can be used within β to assemble a pattern of messages and to consume it locally, using a local REACT step. This two-step decomposition of communication reflects the separation of message transport and message processing in actual implementations.

In the following, we consider only DRCHAMs where *every name is defined in at most one local solution.* This condition is preserved by the chemical semantics, and simplifies the usage of rule COMM: for every message, the rule applies at most once, and delivers the message to a unique receiving location. The actual mapping from channel names to their defining locations is static; it is maintained by the implementation. (In contrast, some recent models of distributed systems detail the explicit routing of messages in the calculus [20,11]. From a language design viewpoint, we believe that the bookkeeping of routing information is a low-level activity that is best handled at the implementation level. At least in the distributed join calculus, the locality property makes routing information simple enough to be safely omitted from the language.)

Remote message passing To illustrate the use of several local solutions, we model a simplistic "print spooler" that matches available printers and job requests. The spooler can be described by the rule

$$D \stackrel{\text{def}}{=} ready\langle printer\rangle | job\langle file\rangle \triangleright printer\langle file\rangle$$

$A, B ::=$		configurations
	$D \vdash^{\varphi} P$	local solution (with path φ and contents D and P)
$\mid$	$A \parallel B$	parallel composition
$P, Q, R ::=$		processes
	$x\langle\widetilde{y}\rangle$	asynchronous message
$\mid$	$\mathtt{go}\ a; P$	migration request
$\mid$	$\mathtt{def}\ D\ \mathtt{in}\ P$	local definition
$\mid$	$P \mid Q$	parallel composition
$\mid$	$\mathbf{0}$	inert process
$D ::=$		join calculus definition
	$J \triangleright P$	reaction rule
$\mid$	$a\,[D : P]$	sub-location (named a, with contents D and P)
$\mid$	$D \wedge D'$	composition
$\mid$	$\top$	void definition
$J ::=$		join pattern
	$x\langle\widetilde{y}\rangle$	message pattern
$\mid$	$J \mid J'$	synchronization

Figure 8. Syntax for the distributed join calculus

$$\textsc{Comm} \qquad \vdash^{\alpha} x\langle\widetilde{y}\rangle \parallel D \vdash^{\beta} \rightarrow \vdash^{\alpha} \parallel D \vdash^{\beta} x\langle\widetilde{y}\rangle$$

$$\textsc{Go} \qquad \vdash^{\alpha a} \parallel \vdash^{\beta b} \mathtt{go}\ a; P \rightarrow \vdash^{\alpha a} \parallel \vdash^{\alpha a b} P$$

$$\textsc{Str-Loc} \qquad a[D : P] \vdash^{\alpha} \rightleftharpoons D \vdash^{\alpha a} P \parallel \vdash^{\alpha}$$

Side conditions:
in Comm, $x \in \mathrm{dv}(D)$;
in Go, b does not occur in any other path;
in Str-Loc, a does not occur in any other path
and $\{D\}$, $\{P\}$ is the only content of solution αa.

The local rules are unchanged (cf. figure 2)
distributed parallel composition $\parallel$ is associate-commutative.

Figure 9. The distributed RCHAM

We assume that there are three machines: a user machine u that issues some print request, a server machine s that hosts the spooler D, and a laser printer p that registers to the spooler. We let P represent the printer code. We have the series of chemical steps:

$$\begin{array}{lllll}
 & \vdash^u job\langle 1\rangle & \parallel & D \vdash^s & \parallel \quad laser\langle f\rangle \triangleright P \vdash^p ready\langle laser\rangle \\
\overset{\text{COMM}}{\longrightarrow} & \vdash^u & \parallel & D \vdash^s job\langle 1\rangle & \parallel \quad laser\langle f\rangle \triangleright P \vdash^p ready\langle laser\rangle \\
\overset{\text{COMM}}{\longrightarrow} & \vdash^u & \parallel & D \vdash^s job\langle 1\rangle, ready\langle laser\rangle & \parallel \quad laser\langle f\rangle \triangleright P \vdash^p \\
\overset{\text{REACT}}{\longrightarrow} & \vdash^u & \parallel & D \vdash^s laser\langle 1\rangle & \parallel \quad laser\langle f\rangle \triangleright P \vdash^p \\
\overset{\text{COMM}}{\longrightarrow} & \vdash^u & \parallel & D \vdash^s & \parallel \quad laser\langle f\rangle \triangleright P \vdash^p laser\langle 1\rangle
\end{array}$$

The first step forwards the message $job\langle 1\rangle$ from the user machine u to the machine that defines job, here the spooler s. Likewise, the second step forwards the message $ready\langle laser\rangle$ to the spooler. Next, synchronization occurs within the spooler between these two messages as a local reduction step. As a result, a new message on the spooler is sent to the laser printer, where it can be forwarded then processed.

From this example, we can also illustrates *global lexical scope*. To model that $laser$ is initially private to the printer machine p, we can use a preliminary local, structural step on machine p:

$$\vdash^p \texttt{def}\ laser\langle f\rangle \triangleright P\ \texttt{in}\ ready\langle laser\rangle \overset{\text{STR-DEF}}{\rightleftharpoons} laser\langle f\rangle \triangleright P \vdash^p ready\langle laser\rangle$$

Then, the second COMM step in the series above extends the scope of $laser$ to the server, which gains the ability to send messages to the printer. In contrast, the scoping rules guarantees that no other process may send such messages at this stage.

Nested locations Assuming that every location is mapped to its host machine, agent migration is naturally represented as an update of this mapping from locations to machines. For instance, a location that contains the running code of a mobile agent may migrate to the machine that hosts another location providing a particular service.

Our model of locality is *hierarchical*, locations being attached to a parent location rather than a machine. A migration request is expressed using the process $go(a); P$ where a is the name of the target location and P is a guarded process triggered after completing the migration. The migration is subjective, inasmuch as it applies to the location that runs the process $go(a); P$ and its sublocations.

As regards distributed programming, there are many situations where several levels of moving resources are useful. For example, the server itself may sometimes move from one machine to another to continue the service while a machine goes down, and the termination of a machine and of all its locations can be modeled using the same mechanisms as a migration. Also, some agents naturally make use of sub-agents, e.g., to spawn some parts of the computation to other machines. When a mobile agent returns to its client machine, for instance,

it may contain running processes and other resources; logically, the contents of the agent should be integrated with the client: later on, if the client moves, or fails, this contents should be carried away or discarded accordingly.

From the implementor's viewpoint, the hierarchical model can be implemented as efficiently as the flat model, because each machine only has to keep track of its own local hierarchy of locations. Nonetheless, the model provides additional expressiveness to the programmer, who can assemble groups of resources that move from one machine to another as a whole. This may explain why most implementations of mobile objects provide a rich dynamic structure for controlling migration, for instance by allowing objects to be temporarily attached to one another (*cf.* [25,24]).

Consider for instance the case of concurrent migrations: a client creates an agent to go and get some information on a server; in parallel, the server goes to another machine.

- With a flat location structure, the migration from the client to the server must be dynamically resolved to a migration to a particular machine, e.g. the machine that currently hosts the server. In the case the server moves after the arrival of the agent, the agent is left behind. That is, the mapping from locations to machines depends on the scheduling of the different migrations, and the migration to the server yields no guarantee of being on the same machine as the server.
- With a hierarchical structure, the ordering of nested migrations becomes irrelevant, and the agent is guaranteed to remain with the server as long as the agent does not explicitly request another migration, even as the server moves.

Relating locations to local chemical solutions We must represent locations both as syntactic definitions (when considered as a sublocation, or in a guarded process) and as local chemical solutions (when they interact with one another). We rely on location names to relate the two structures. We assume given a countable set of location names $a, b, \ldots \in \mathcal{L}$. We also write $\alpha, \beta, ab, \alpha b, \ldots \in \mathcal{L}^*$ for finite strings of location names, or *paths*. Location names are first-class values. Much as channel names, they can be created locally, sent and received in messages, and they have a lexical scope. To introduce new locations, we extend the syntax of definitions with a new location constructor:

$$D \stackrel{\text{def}}{=} \ldots \mid a\,[D' : P]$$

where D' gathers the definitions of the location, where P is the code running in the location, and where a is a new name for the location. As regards the scopes, $a\,[D' : P]$ *defines* the name a and the names defined in D'.

Informally, the definition $a\,[D' : P]$ corresponds to the local solution $D' \vdash^{\beta a} P$, where β is the path of D's local solution. We say that $\vdash^{\alpha}$ is a sublocation of $\vdash^{\beta}$ when β is a prefix of α. In the following, DRCHAMs are multisets of solutions labeled with paths α that are all distinct, prefix-closed, and uniquely identified

by their rightmost location name, if any. These conditions ensure that solutions ordered by the sublocation relation form a tree.

The new structural rule Str-Loc relates the two representations of locations. From left to right, the rule takes a sublocation definition and creates a running location that initially contains a single definition D and a single running process P. From right to left, Str-loc has a "freezing" effect on location a and all its sublocations. The rule has a side condition that requires that there is no solution of the form $\vdash^{\psi a \phi}$ in the implicit chemical context for any ψ, ϕ in $\mathcal{L}^*$; in contrast, the definition D may contain sublocations. The side condition guarantees that D syntactically captures the whole subtree of sublocations in location a when the rule applies. Note that the rule Str-def and its side condition also apply to defined location names. This guarantees that newly-defined locations are given fresh names, and also that locations that are folded back into defining processes do not leave any running sublocation behind.

In well-formed drchams, we have required that all reaction rules defining a given channel name belong to a single local solution, and that all local solutions have distinct paths. With the addition of frozen locations in solution, we also require that locations in solution all have distinct location names that do not appear in the path of any local solution. We constrain the syntax of definitions accordingly: in a well-formed definition, for all conjunctions $D \wedge D'$, we require that $\mathrm{dv}(D) \cap \mathrm{dv}(D')$ contain only port names that are not defined in a sublocation of D or D'. For instance, the definitions $a\,[\top : \mathbf{0}] \wedge a\,[\top : \mathbf{0}]$ and $a\,[x\langle\rangle \triangleright P \wedge b\,[x\langle\rangle \triangleright Q : \mathbf{0}] : \mathbf{0}]$ are ruled out.

As an example of nested locations, we describe a series of structural rearrangements that enable some actual reduction steps. We assume that a does not occur in P_c or Q.

$$
\begin{array}{rl}
 & \vdash \mathtt{def}\ c\big[x\langle u\rangle \triangleright Q \wedge a\,[D_a : P_a] : P_c\big]\ \mathtt{in}\ y\langle c, x\rangle | x\langle a\rangle \\
\overset{\text{Str-Def}}{\rightleftharpoons} & c\big[x\langle u\rangle \triangleright Q \wedge a\,[D_a : P_a] : P_c\big] \vdash y\langle c, x\rangle | x\langle a\rangle \\
\overset{\text{Str-Loc}}{\rightleftharpoons} & x\langle u\rangle \triangleright Q \wedge a\,[D_a : P_a] \vdash^c P_c \quad \| \quad \vdash y\langle c, x\rangle | x\langle a\rangle \\
\overset{\text{Str-Def,Par}}{\rightleftharpoons} & x\langle u\rangle \triangleright Q,\ a\,[D_a : P_a] \vdash^c P_c \quad \| \quad \vdash y\langle c, x\rangle,\ x\langle a\rangle \\
\overset{\text{Str-loc}}{\rightleftharpoons} & D_a \vdash^{ca} P_a \quad \| \quad x\langle u\rangle \triangleright Q \vdash^c P_c \quad \| \quad \vdash y\langle c, x\rangle, x\langle a\rangle \\
\overset{\text{Comm}}{\rightarrow} & D_a \vdash^{ca} P_a \quad \| \quad x\langle u\rangle \triangleright Q \vdash^c P_c, x\langle a\rangle \quad \| \quad \vdash y\langle c, x\rangle \\
\overset{\text{React}}{\rightarrow} & D_a \vdash^{ca} P_a \quad \| \quad x\langle u\rangle \triangleright Q \vdash^c P_c, Q\{^a/_u\} \quad \| \quad \vdash y\langle c, x\rangle \\
\rightleftharpoons & \vdash \mathtt{def}\ c\big[x\langle u\rangle \triangleright Q : P_c | \mathtt{def}\ a\,[D_a : P_a]\ \mathtt{in}\ Q\{^a/_u\}\big]\ \mathtt{in}\ y\langle c, x\rangle
\end{array}
$$

Now that the bookkeeping of the location tree is handled as a special case of structural rearrangement, we can express migration as the relocation of a branch in the location tree. We extend distributed chemical semantics with a second reduction rule that operates on two chemical solutions (rule Go). Informally, location b moves from its current position βb in the tree to a new position $\alpha a b$ just under the location name a passed to the migration request. The target solution $\vdash^{\alpha a}$ is identified by its name a. Once b arrives, the guarded process P is

triggered. The side condition forces the preliminary application of rule Str-Loc to fold any sublocation of a, and thus guarantees that the branch moves as a whole.

5.3 Attaching some meaning to locations

In the machine above, the locality structure is mostly descriptive: the distributed semantics keeps track of locality information, but the location of a particular process or definition does not affect the result of the computation, at least for the observables studied in Section 2. Formally, we can erase any locality information and relate the simple semantics to the distributed semantics. (Some care is required to rule out migration attempts towards one's own sublocation, which may delay or block some processes.)

To conclude, we briefly present two refined models where locality has an impact on the computation and can be partially observed.

Partial failure and failure-detection Our calculus can be extended with a simple "fail-stop" model of failure, where the crash of a physical site causes the permanent failure of its processes and definitions. In the extended model, a location can *halt* with all its sublocations. The failure of a location can also be asynchronously detected at any other running location, allowing programmable error recovery.

We supplement the syntax of distributed processes with constructs for failure and asynchronous failure detection:

$P, Q, R ::=$		processes
	$\ldots$	*(as in Figure 8)*
	$\mid$ `halt`	local failure
	$\mid$ `fail` $a; P$	remote failure detection

We also put an optional "has failed" marker Ω in front of every location name in paths, and add a side condition "the path does not contain Ω" in front of every chemical reduction rule (that is, a failed location and its sub-locations cannot migrate, communicate, or perform local steps). In addition, we provide two chemical rules for the new constructs:

$$\textsc{Halt} \qquad \vdash^{\alpha a} \texttt{halt} \;\rightarrow\; \vdash^{\alpha \Omega a}$$

$$\textsc{Detect} \qquad \vdash^{\alpha a} \;\|\; \vdash^{\beta b} \texttt{fail}\ a; P \;\rightarrow\; \vdash^{\alpha a} \;\|\; \vdash^{\beta b} P$$

with side conditions: in Halt, a does not occur in any other path and the marker Ω does not occur in α; in Go, the marker Ω occurs in α but not in β.

Inasmuch as the only reported failures are permanent location failures, the programming model remains relatively simple[4]. For instance, the single delivery

[4] In addition, timeouts are pragmatically useful, but they are trivially modeled in an asynchronous setting—they are ordinary non-deterministic reduction steps—and they do not provide any negative guarantee.

of every message is guaranteed unless the sender fails, and thus the programmer can consider larger units of failure. Besides, failure-detection provides useful *negative information*: once a location failure is detected, no other location may interact with it, hence its task may be taken over by another location without interfering with the failed location. The model of failure is only partially supported in JoCaml (some failures may never be reported).

Authentication and Secrecy Interpreting locations as security boundaries, or "principals", we have also developed a model of authentication for a variant of the join calculus [4]. In this variants, we consider only a flat and static parallel composition of locations.

As in the distributed join calculus, every name "belongs" to the single location that defines it, hence only that location can receive messages sent on that name, while other locations cannot even detect those messages. In addition, every message carries a mandatory first parameter that contains a location name, meant to be the name of the sending location, and used by the receiver to authenticate the sender. These properties are both enforced by a modified COMM rule:

$$\text{SECURE-COMM} \qquad \vdash^a x\langle c, \widetilde{y}\rangle \;\; \| \;\; D_x \vdash^b \;\rightarrow\; \vdash^a \;\; \| \;\; D_x \vdash^b x\langle a, \widetilde{y}\rangle$$

where x is (uniquely) defined in D_x. Remark that the first parameter c is overwritten with the correct sender identity a, much as in Modula 3's secure network objects [42].

These secrecy and authenticity properties provide a strong, abstract basis for reasoning on the security of distributed programs, but their semantics depends on global chemical conditions that are not necessarily met in practical implementations—when communicating with an untrusted remote machine, there is no way to check that it is running a correct implementation. This delicate implementation is actually the main issue in [4], where we show that those strong properties can be faithfully implemented using standard cryptographic primitives, with much weaker assumptions on the network. This result is expressed in terms of bisimilarity equivalences.

Acknowledgements

These lecture notes are partly derived from previous works in collaboration with Martín Abadi, Sylvain Conchon, Cosimo Laneve, Fabrice Le Fessant, Jean-Jacques Lévy, Luc Maranget, Didier Rémy, and Alan Schmitt.

References

1. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. Manuscript, on the Web at `http://research.microsoft.com/~fournet`; subsumes [2] and [3].

2. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *Proceedings of LICS '98*, pages 105–116. IEEE, June 1998.
3. M. Abadi, C. Fournet, and G. Gonthier. Secure communications processing for distributed languages. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 74–88, May 1999.
4. M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *Proceedings of POPL'00*, pages 302–315. ACM, Jan. 2000.
5. G. Agha, I. Mason, S. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.
6. R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.
7. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
8. M. Boreale and D. Sangiorgi. Some congruence properties for π-calculus bisimilarities. *Theoretical Computer Science*, 198(1–2):159–176, 1998.
9. E. Brinksma, A. Rensink, and W. Vogler. Fair testing. In I. Lee and S. A. Smolka, editors, *6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 313–327. Springer-Verlag, 1995.
10. E. Brinksma, A. Rensink, and W. Vogler. Applications of fair testing. In R. Gotzhein and J. Bredereke, editors, *Formal Description Techniques IX: Theory, Applications and Tools*, volume IX. Chapman and Hall, 1996.
11. L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.
12. S. Conchon and F. Le Fessant. Jocaml: Mobile agents for objective-caml. In *ASA/MA'99*, pages 22–29. IEEE Computer Society, Oct. 1999.
13. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Palaiseau, Nov. 1998. INRIA TU-0556. Also available from `http://research.microsoft.com/~fournet`.
14. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pages 372–385. ACM, Jan. 1996.
15. C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi (extended abstract). In K. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP '98)*, volume 1443 of *LNCS*, pages 844–855, Aalborg, Denmark, July 1998. Springer-Verlag.
16. C. Fournet and C. Laneve. Bisimulations in the join-calculus. To appear in TCS, available from `http://research.microsoft.com/~fournet`, Oct. 2000.
17. C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Proceedings of IFIP TCS 2000*, volume 1872 of *LNCS*, Sendai, Japan, 17–19 Aug. 2000. IFIP TC1, Springer-Verlag. An extended report is available from `http://research.microsoft.com/~fournet`.
18. C. Fournet and L. Maranget. The join-calculus language (version 1.03 beta). Source distribution and documentation available from `http://join.inria.fr/`, June 1997.
19. R. Glabbeek. The linear time—branching time spectrum II; the semantics of sequential systems with silent moves (extended abstract). In E. Best, editor, *4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *LNCS*, pages 66–81. Springer-Verlag, 1993. Also Manuscript, preliminary version available at `ftp://Boole.stanford.edu/pub/spectrum.ps.gz`.

20. M. Hennessy and J. Riely. A typed language for distributed mobile processes. In *Proceedings of POPL '98*, pages 378–390. ACM, Jan. 1998.
21. K. Honda and M. Tokoro. On asynchronous communication semantics. In P. Wegner, M. Tokoro, and O. Nierstrasz, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, volume 612 of *LNCS*, pages 21–51. Springer-Verlag, 1992.
22. K. Honda and N. Yoshida. Combinatory representation of mobile processes. In *Proceedings of POPL '94*, pages 348–360, 1994.
23. K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.
24. E. Jul. Migration of light-weight processes in emerald. *IEEE Operating Sys. Technical Committee Newsletter, Special Issue on Process Migration*, 3(1):20, 1989.
25. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 62–74, November 1987.
26. C. Laneve. May and must testing in the join-calculus. Technical Report UBLCS 96-04, University of Bologna, Mar. 1996. Revised: May 1996.
27. F. Le Fessant. The JoCAML system prototype (beta). Software and documentation available from `http://pauillac.inria.fr/jocaml`, 1998.
28. F. Le Fessant and L. Maranget. Compiling join-patterns. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*, Nice, France, Sept. 1998. Elsevier Science Publishers. To appear.
29. X. Leroy and al. The Objective CAML system 3.01. Software and documentation available from `http://caml.inria.fr`.
30. R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
31. R. Milner. *Communication and Mobile Systems: the π-Calculus*. Cambridge University Press, Cambridge, 1999.
32. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, Sept. 1992.
33. R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP'92*, volume 623 of *LNCS*, pages 685–695. Springer-Verlag, 1992.
34. V. Natarajan and R. Cleaveland. Divergence and fair testing. In *Proceedings of ICALP '95*, volume 944 of *LNCS*. Springer-Verlag, 1995.
35. U. Nestmann and B. C. Pierce. Decoding choice encodings. In U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 179–194. Springer-Verlag, Aug. 1996. Revised full version as report ERCIM-10/97-R051, 1997.
36. J. Parrow and P. Sjödin. Multiway synchronization verified with coupled simulation. In R. Cleaveland, editor, *Third International Conference on Concurrency Theory (CONCUR'92)*, volume 630 of *LNCS*, pages 518–533. Springer-Verlag, 1992.
37. J. Parrow and P. Sjödin. The complete axiomatization of cs-congruence. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings of STACS'94*, volume 775 of *LNCS*, pages 557–568. Springer-Verlag, 1994.
38. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. Ph.D. thesis, University of Edinburgh, May 1993.
39. D. Sangiorgi. On the bisimulation proof method. Revised version of Technical Report ECS–LFCS–94–299, University of Edinburgh, 1994. An extended abstract appears in Proc. of MFCS'95, LNCS 969, 1994.

40. D. Sangiorgi and R. Milner. The problem of "weak bisimulation up to". In W. R. Cleaveland, editor, *Proceedings of CONCUR'92*, volume 630 of *LNCS*, pages 32–46. Springer-Verlag, 1992.
41. D. Sangiorgi and D. Walker. *The Pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, July 2001. ISBN 0521781779.
42. L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 211–221, May 1996.