Secure Composition of Untrusted Code: Box π, Wrappers, and Causality Types

Peter Sewell
Computer Laboratory
University of Cambridge
Peter.Sewell@cl.cam.ac.uk

Jan Vitek
Department of Computer Sciences
Purdue University
jv@cs.purdue.edu

March 29, 2002

Abstract

Software systems are becoming heterogeneous: instead of a small number of large programs from well-established sources, a user's desktop may now consist of many smaller components that interact in intricate ways. Some components will be downloaded from the network from sources that are only partially trusted. A user would like to know that a number of security properties hold, e.g. that personal data is not leaked to the net, but it is typically infeasible to verify that such components are well-behaved. Instead, they must be executed in a secure environment that provides fine-grain control of the allowable interactions between them, and between components and other system resources.

In this paper, we consider the problem of assembling concurrent software systems from untrusted or partially trusted off-the-shelf components, using *wrapper programs* to encapsulate components and enforce security policies. We introduce a model programming language, the *box-π calculus*, that supports composition of software components and the enforcement of information flow security policies. Several example wrappers are expressed using the calculus; we explore the delicate security properties they guarantee. We present a novel *causal type system* that statically captures the allowed flows between wrapped possibly-badly-typed components; we use it to prove that an example ordered pipeline wrapper enforces a causal flow property.

1 Introduction

Software systems are evolving. Increasingly, monolithic applications are being replaced with assemblages of software components coming from different sources. Instead of a small number of large programs from well-established suppliers, nowadays a user's desktop is made up of many smaller applications and software modules that interact in intricate ways to carry out a variety of information processing tasks. Moreover, whereas it used to be that a software base was fairly static and controlled by a system administrator, it is now easy to download code from the network and even extend application programs while they are running. These components are obtained from different untrusted or partially-trusted sources and they may be faulty or malicious, or designed with a weaker security policy that the user requires – what is legitimate marketing data to a vendor may be considered sensitive by a user. It is difficult for a user to gain assurance that the composed system is secure.

In such fluid operating environments, traditional security mechanisms and policies appear almost irrelevant. While passwords and access control mechanisms are adequate to protect the integrity of the computer system as whole, they utterly fail to address the issue of protecting the user from downloaded code being run from her account [21, 15, 27]. Approaches such as the Java sandbox that promise security by isolation are not satisfactory either, as they propose a model in which components can either interact freely or not at all [16].

While it is not feasible, in general, to analyse or modify third-party software packages, it *is* possible to intercept the communications between a package and the other parts of the system, interposing code at the boundaries of the different software components to observe and modify the data passing through [22, 44, 13, 12, 10, 4, 15]. Interposition techniques effectively encapsulate untrusted components in *wrapper programs* that have full control over the interactions between encapsulated components and the OS and over the interactions among components. The code of a wrapper can, for instance, perform access control checks, audit, attempt to detect intruders, and even monitor covert channels. Clearly, writing wrappers should not be left to the end-user. Rather we envision wrappers as reusable software components; users should then only have to pick the most appropriate wrappers, customize them with some parameters and install them. All of this process should be dynamic: wrappers must be no harder to add to a running system than new applications. Ultimately, end users will require a clear description of the properties guaranteed by their wrappers.

Practical work on wrappers underlines the difficulty of understanding exactly what these guarantees are. For example, in [12] Fraser, Badger and Feldman presented a system that splits the task of writing a wrapper into two parts. The wrapper's body is written in a variant of C called the Wrapper Definition Language. The dynamic aspects of creating wrappers and instantiating concurrently executing components are specified in the Wrapper Life Cycle framework. While quite expressive, their approach does not provide guarantees that the wrappers actually enforce the desired security policies. The powerful wrapper language, the fact that all wrappers execute in kernel mode, and the fact that components are concurrent combine to make it difficult to understand precisely what properties a wrapper enforces.

Our work in the current paper explores secure composition using wrappers, focussing on the rigorous statement and proof of their security properties. To express and reason about wrappers we require a small programming language, with a well-defined semantics, that allows the composition of concurrently-executing software components to be expressed straightforwardly and also supports the enforcement of security poli-

cies. We have therefore abstracted the essential aspects of the problem in a process calculus: the *box-π calculus*, introduced in Section 2. Box-π is a minimal extension of the π-calculus [26] with encapsulation; it is expressive enough to code non-trivial wrappers and the concurrent composition of components, but retains the simplicity and tractable semantics needed for proving properties. Moreover Pict [29] demonstrates how to build a real programming language above a π-calculus core; a similar approach could be used for box-π.

Several wrappers are expressed in box-π in Section 3. We begin with a simple example, the wrapper $\mathcal{W}_1$. It encapsulates a single component and controls its interactions with the environment, limiting them to two channels *in* and *out*. $\mathcal{W}_1$ is written as a unary context in Figure 1.

$$\begin{array}{lcll} \mathcal{W}_1(\_) & \stackrel{def}{=} & (\boldsymbol{\nu}\, a)\big(& a[\_] \\ & & & \mid \mathbf{!}\, in^{\uparrow} y.\overline{in}^{a}\, y \\ & & & \mid \mathbf{!}\, out^{a}\, y.\overline{out}^{\uparrow} y\big) \end{array}$$

Figure 1: The filtering wrapper $\mathcal{W}_1$ in box-π.

This creates a box with a new name a, installing in parallel with it two forwarders – one that receives messages from the environment on channel *in* and sends them to the wrapped program, and one that receives messages from the wrapped program on channel *out* and sends them to the environment. An arbitrary program P (possibly malicious) can be wrapped to give $\mathcal{W}_1(P)$; the design of the calculus and of $\mathcal{W}_1$ ensures that no matter how P behaves the wrapped program $\mathcal{W}_1(P)$ can only interact with its environment on the two channels *in* and *out*.

The wrapper $\mathcal{W}_1$ controls interaction between a single component and its environment – it filters messages that the component can send and receive, restricting it to a particular interface. A more interesting case occurs when the interaction between untrusted components has to be controlled. In Section 3.3 we introduce $\mathcal{W}_2$, a binary wrapper that encapsulates two components P and Q as $\mathcal{W}_2(P,Q)$, allowing each to interact with the environment in a limited way but also allowing information to flow from P to Q, but not vice versa, along a directed communication channel. The box-π program of Figure 2 is a simplified version of this example.

$$(\boldsymbol{\nu}\, a, b)\big(\quad a[\,P\,] \quad\mid\quad \mathbf{!}\, c^{a} x.\overline{c}^{b} x \quad\mid\quad b[\,Q\,] \quad\big)$$

Figure 2: A simplified pipeline wrapper $\mathcal{W}_2$, encapsulating P and Q.

Processes P and Q are arbitrary, possibly malicious, components. They are encapsulated in named boxes, with private names a and b, and placed in parallel with a forwarder process on channel c from box a to box b. The term $\overline{c}^{b} x$ is an output to channel c in box b of value x. The term $c^{a} x.\overline{c}^{b} x$ prefixes this with an input on channel c from box a; here the first x is a formal parameter that binds the second. The $\mathbf{!}$ operator indicates a replicated input, so the forwarder persists after use. The boxes restrict communication of the encapsulated processes and ensure that P and Q cannot interact with each other directly; the private names ensure that they cannot interact with their environment in any other way. This simplified forwarder sends only unordered asynchronous messages; our main example, the wrapper $\mathcal{F}$ of Section 3.4, provides

FIFO communication on an ordered pipeline (this is related to the NRL pump [23], as discussed in later).

Intuitively, the wrapper $\mathcal{W}_2$ enforces an information flow policy that prevents Q from leaking secrets to P. When one attempts to make such properties precise, however, there are many choices. A body of model-theoretic work on *non-interference* uses delicate extensional properties of the trace sets of systems. In our programming language setting a more intensional approach allows what we believe to be clearer statements. We start with a labelled transition semantics (defined in §2.3) that specifies the input/output behaviour of programs and extend it to represent and propagate causal dependencies explicitly. In terms of this, one can state the desired property as 'no visible action of P is causally dependent on any action of Q'. The causal semantics and property are defined in Section 4.

Verifying such a causal flow property directly can be laborious, requiring a characterisation of the state space of a wrapper containing arbitrary components. We therefore introduce a type system that statically captures causal flows. Since components are often provided as object code, which is impractical for the user to typecheck, our type system must admit programs with badly-typed subcomponents.

The causal type system, given in Section 5, allows us to prove information flow properties of box-π programs. For the example of Figure 2, to statically allow the flow from a to b but disallow the converse we can associate the components with principals p and q, then take a to be a box name whose contents may be affected by p, written $a : \mathbf{box}_{\{\mathsf{p}\}}$, b to be a box name whose contents may be affected by p or q, written $b : \mathbf{box}_{\{\mathsf{p},\mathsf{q}\}}$, and c to be a channel, carrying values of a top type $\top$, which can be affected only by p, so $c : \mathbf{chan}_{\{\mathsf{p}\}} \top$. The fragment is then typable, whereas the converse forwarder $c^b x.\overline{c}^a x$ is not. The type system also deals with tracking causes through computation within a wrapper, including communication of channel names, and with interaction between a wrapper and badly-typed components. All boxes are assumed to contain untyped processes; wrapper code is statically typed; run-time type checking is required only when receiving from a component.

Further discussion of related work is given in Section 6; Section 7 concludes with future work. The appendices contain outline proofs of the results; full details can be found in the technical reports [34, 36]. This paper is an extended version of [35, 37].

2 A Boxed π Calculus

The language – known as the *box-π calculus* – that we use for studying encapsulation properties must allow interacting components to be composed. The components will typically be executing concurrently, introducing nondeterminism. It is therefore natural to base the language on a process calculus. The box-π calculus lies in a large design space of distributed calculi that build on the π-calculus of Milner, Parrow and Walker [26], including among others the related calculi [2, 8, 11, 30, 32, 38, 40]. A brief overview of the design space can be found in [33]; here we highlight the main design choices for box-π.

The calculus is based on asynchronous message passing, with components interacting only by the exchange of unordered asynchronous messages. Box-π has an asynchronous π-calculus as a subcalculus – we build on a large body of work studying such calculi, notably [19, 6]. They are known to be very expressive, supporting many programming idioms including functions and objects, and are Turing-complete; a box-π process may therefore perform arbitrary internal computation. The choice of asyn-

chronous communication is important as it allows two components to interact without creating causal connections in both directions between them.

Box-π requires facilities for constraining communication – in standard π-calculi, if one process can send a message to another then the only way to prevent information flowing in the reverse direction is to impose a type system on components, which (as observed above) is not appropriate here. We therefore add a boxing primitive – boxes may be nested, giving hierarchical protection domains; communication across box boundaries is strictly limited. Underlying the calculus design is the principle that each box should be able to control all interactions of its children, both with the outside world and with each other [40]. Boxes can be viewed as protection domains, akin to operating system-enforced address spaces. All other communication, in particular that between two sibling boxes, must be mediated by code running in the parent. This code can enforce an arbitrary security policy, even supporting dynamically-changing policies and interfaces (in contrast to static restriction or blocking operators [7, 41]).

Turning to the values that may be communicated, it is convenient to allow arbitrary tuples of names (or other tuples). Note that we do *not* allow communication of process terms. Moreover, no primitives for movement of boxes are provided, in contrast to most work cited above. The calculus is therefore entirely first order, which is important for the tractable theory of behaviour (the labelled transition semantics) that we require to state and prove security properties. The calculus is also untyped – we wish to consider the wrapping of ill-understood, probably buggy and possibly malicious programs.

2.1 Syntax

The syntax of the calculus is as follows:

Names We take an infinite set $\mathcal{N}$ of *names*, ranged over by a, b, c etc. (except i, j, k, o, p, u, v). Both boxes and communication channels are named; names also play the role of variables, as in the π-calculus.

Values and Patterns Processes will interact by communicating values which are deconstructed by pattern-matching by the receiver. Values u, v can be names or tuples, with patterns p correspondingly tuple-structured.

u, v	$::=$	x	name
		$\langle v_1 \,..\, v_k \rangle$	tuple ($k \geq 0$)

p	$::=$	$\_$	wildcard
		x	name pattern
		$(p_1 \,..\, p_k)$	tuple pattern
			($k \geq 0$, no repeated names)

Processes The main syntactic category is that of *processes*, ranged over by P, Q. We introduce the primitives in three groups.

Boxes A box $n[P]$ has a name n, it can contain an arbitrary process P. Box names are not necessarily unique – the process $n[0] \mid n[0]$ consists of two distinct boxes named n,

both containing an empty process, in parallel.

$$\begin{array}{rcll} P & ::= & n[P] & \text{box named } n \text{ containing } P \\ & & P \mid P' & P \text{ and } P' \text{ in parallel} \\ & & 0 & \text{the nil process} \\ & & \ldots & \end{array}$$

Communication The standard asynchronous π-calculus communication primitives are $\overline{x}v$, indicating an output of value v on the channel named x, and $xp.P$, a process that will receive a value output on channel x, binding it to p in P. Here we refine these with a tag indicating the direction of the communication in the box hierarchy. An *input tag* ι can be either $\star$, for input within a box, $\uparrow$, for input from the parent box, or a name n, for input from a sub-box named n. An *output tag* o can be any of these, similarly. For technical reasons we must also allow an output tag to be $\overline{\uparrow}$, indicating an output received from the parent that has not yet interacted with an input, or $\overline{n}$, indicating an output received from child n that has not yet interacted. The communication primitives are then

$$\begin{array}{rcll} P & ::= & \ldots & \\ & & \overline{x}^o v & \text{output } v \text{ on channel } x \text{ to } o \\ & & x^\iota p.P & \text{input on channel } x \text{ from } \iota \\ & & !\, x^\iota p.P & \text{replicated input} \\ & & \ldots & \end{array}$$

The replicated input $!\, x^\iota p.P$ behaves essentially as infinitely many copies of $x^\iota p.P$ in parallel. This gives computational power, allowing e.g. recursion to be encoded simply, while keeping the theory simple. In $x^\iota p.P$ and $!\, x^\iota p.P$ the names occurring in the pattern p bind in P. Empty patterns and tuples will often be elided.

New name creation Both box and channel names can be created fresh, with the standard π-calculus $(\boldsymbol{\nu}\, x)P$ operator (also known as *restriction*). This declares any free instances of x within P to be instances of a globally fresh name.

$$\begin{array}{rcll} P & ::= & \ldots & \\ & & (\boldsymbol{\nu}\, x)P & \text{new name creation} \end{array}$$

In $(\boldsymbol{\nu}\, x)P$ the x binds in P. We work up to alpha conversion of bound names throughout. This means, for example, that $(\boldsymbol{\nu}\, y)\overline{x}^\uparrow y$ and $(\boldsymbol{\nu}\, z)\overline{x}^\uparrow z$ denote the same mathematical object. We write the free name function, defined in the obvious way for values, tags and processes, as $\mathrm{fn}(\_)$, so $\mathrm{fn}((\boldsymbol{\nu}\, y)\overline{x}^\uparrow y) = \{x\}$. Figure 3 summarizes the syntax of box-π.

2.2 Reduction Semantics

The simplest semantic definition of the calculus is a *reduction semantics*, a one-step reduction relation $P \to P'$ indicating that P can perform one step of internal computation to become P'. We first define the complement $\overline{\iota}$ of a tag ι in the obvious way, with $\overline{\overline{\star}} = \star$ and $\overline{\overline{\iota}} = \iota$. We define a partial function $\{\tilde{}/\_\}$, taking a pattern and a value and giving, where it is defined, a partial function from names to values.

$$\begin{array}{rcl} \{^v/_\_\} & = & \{\} \\ \{^v/_x\} & = & \{x \mapsto v\} \\ \{^{\langle v_1\, ..\, v_{k'} \rangle}/_{(p_1\, ..\, p_k)}\} & = & \{^{v_1}/_{p_1}\} \cup \ldots \cup \{^{v_k}/_{p_k}\} \text{ if } k = k' \\ & & \text{undefined, otherwise} \end{array}$$

u, v	$::=$	x	name	p	$::=$	$\_$	wildcard
		$\langle v_1 \,..\, v_k \rangle$	tuple			x	name pattern
						$(p_1 \,..\, p_k)$	tuple pattern

P	$::=$	$n[P]$	box named n containing P
		$P \mid P'$	P and P' in parallel
		0	the nil process
		$\overline{x}^o v$	output v on channel x to o
		$x^\iota p.P$	input on channel x from ι
		$!\, x^\iota p.P$	replicated input
		$(\boldsymbol{\nu}\, x)P$	new name creation

Figure 3: Box-π syntax.

The natural definition of the application of a substitution σ (from names to values) to a process term P, written σP, is also a partial operation, as the syntax does not allow arbitrary values in all the places where free names can occur. We write $\{^v/_p\}P$ for the result of applying the substitution $\{^v/_p\}$ to P. This may be undefined either because $\{^v/_p\}$ is undefined, or because $\{^v/_p\}$ is a substitution but the application of that substitution to P is undefined. For example, $\{^{\langle z\, z\rangle}/_x\}\overline{x}^\star\langle\rangle$ is not defined as $\overline{\langle z\, z\rangle}^\star\langle\rangle$ is not in the syntax. Note that the result $\{^y/_x\}P$ of applying a name-for-name substitution is always defined.

This definition of substitution leads to a lightweight notion of runtime error. A more conventional notion of runtime error would give errors only when a tuple is used as a name, e.g. for output. The substitution-based notion is forced by our choice of syntax, which disallows values in various places where names may appear. In general it will report errors sooner than the conventional notion.

The definition of reduction involves an auxiliary *structural congruence* $\equiv$, defined as the least congruence relation such that the axioms of Figure 4 hold. This allows the parts of a redex (an instance of the left-hand-side of one of the axioms in Figure 5) to be brought syntactically adjacent.

$P \mid 0$	$\equiv$	P	
$P \mid Q$	$\equiv$	$Q \mid P$	
$(P \mid Q) \mid R$	$\equiv$	$P \mid (Q \mid R)$	
$(\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P$	$\equiv$	$(\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P$	
$(\boldsymbol{\nu}\, x)(P \mid Q)$	$\equiv$	$P \mid (\boldsymbol{\nu}\, x)Q$	$x \notin \mathrm{fn}(P)$
$(\boldsymbol{\nu}\, x)n[P]$	$\equiv$	$n[(\boldsymbol{\nu}\, x)P]$	$x \neq n$

Figure 4: Structural congruence relation.

The reduction relation is now the least relation over processes satisfying the axioms and rules of Figure 5. The (Red Comm) and (Red Repl) axioms are subject to the condition that $\{^v/_p\}P$ is well-defined. The (Red Up) axiom allows an output to the parent of a box to cross the enclosing box boundary. Similarly, the (Red Down) axiom allows an output to a child box n to cross the boundary of n. The (Red Comm) axiom then allows synchronisation between a complementary output and input within the same box. The (Red Repl) axiom is similar, but preserves the replicated input in the resulting state.

$n[\overline{x}^{\uparrow} v \mid Q] \rightarrow \overline{x}^{\overline{n}} v \mid n[Q]$	(Red Up)
$\overline{x}^{n} v \mid n[Q] \rightarrow n[\overline{x}^{\overline{\uparrow}} v \mid Q]$	(Red Down)
$\overline{x}^{\overline{\iota}} v \mid x^{\iota} p.P \rightarrow \{^{v}/_{p}\}P$	(Red Comm)
$\overline{x}^{\overline{\iota}} v \mid \mathbf{!}\, x^{\iota} p.P \rightarrow \mathbf{!}\, x^{\iota} p.P \mid \{^{v}/_{p}\}P$	(Red Repl)
$P \rightarrow Q \quad \Rightarrow \quad P \mid R \rightarrow Q \mid R$	(Red Par)
$P \rightarrow Q \quad \Rightarrow \quad (\boldsymbol{\nu}\, x)P \rightarrow (\boldsymbol{\nu}\, x)Q$	(Red Res)
$P \rightarrow Q \quad \Rightarrow \quad n[P] \rightarrow n[Q]$	(Red Box)
$P \equiv P' \rightarrow Q' \equiv Q \quad \Rightarrow \quad P \rightarrow Q$	(Red Struct)

Figure 5: Box-π reduction semantics.

Communications across box boundaries take two reduction steps, as in the following upwards and downwards communications.

$$\begin{array}{lcl} n[\overline{x}^{\uparrow} v] \mid x^{n} p.P & \rightarrow & n[0] \mid \overline{x}^{\overline{n}} v \mid x^{n} p.P \\ & \rightarrow & n[0] \mid \{^{v}/_{p}\}P \end{array}$$

$$\begin{array}{lcl} \overline{x}^{n} v \mid n[x^{\uparrow} p.P] & \rightarrow & n[\overline{x}^{\overline{\uparrow}} v \mid x^{\uparrow} p.P] \\ & \rightarrow & n[\{^{v}/_{p}\}P] \end{array}$$

This removes the need for 3-way synchronisations between a box, an output and an input (as in [40]), simplifying both the semantics and the implementation model.

2.3 Labelled Transition Semantics

The reduction semantics defines only the internal computation of processes. The statements of our security properties must involve the interactions of processes with their environments, requiring more structure: a *labelled transition relation* characterising the potential inputs and outputs of a process. We give a labelled semantics for box-π in an explicitly-indexed early style, defined inductively on process structure by a structured operational semantics. The reader unfamiliar with process calculi may wish to skim to the start of Section 3 on a first reading.

The *labels* are

$$\begin{array}{lcll} \ell & ::= & \tau & \text{internal action} \\ & & \overline{x}^{o} v & \text{output action} \\ & & x^{\gamma} v & \text{input action} \end{array}$$

where o is any output tag and γ ranges over tags $\star$, n, $\uparrow$ and $\overline{n}$. The labelled transitions can be divided into those involved in moving messages across box boundaries and those involved in communications between outputs and inputs. The movement labels are

$\overline{x}^{n} v$ (sending to child n)
$x^{\overline{n}} v$ (box n receiving from its parent)
$\overline{x}^{\uparrow} v$ (sending to the parent)

We say $\mathrm{mv}(o)$ iff o is of the form n or $\uparrow$. The communication labels are

$\overline{x}^{\star}v$ (local output)
$x^{\star}v$ (local input)
$\overline{x}^{\overline{n}}v$ (output received from child n)
$x^{n}v$ (input a message received from child n)
$\overline{x}^{\overline{\uparrow}}v$ (output received from parent)
$x^{\uparrow}v$ (input a message received from parent)

Labels synchronise in the pairs $\overline{x}^{\overline{\gamma}}v$ and $x^{\gamma}v$. The labelled transition relation has the form

$$A \vdash P \xrightarrow{\ell} Q$$

where A is a finite set of names and $\mathrm{fn}(P) \subseteq A$; it should be read as 'in a state where the names A may be known to P and its environment, process P can do ℓ to become Q'. The relation is defined as the smallest relation satisfying the rules of Figure 6. We write A, x for $A \cup \{x\}$ where x is assumed not to be in A, and A, p for the union of A and the names occurring in the pattern p, where these are assumed disjoint.

The labelled semantics is similar to a standard π semantics, but must also deal with boxes and with reductions such as

$$((\boldsymbol{\nu}\, x)\overline{x}^{n}z) \mid n[0] \quad \rightarrow \quad (\boldsymbol{\nu}\, x)n[\overline{x}^{\overline{\uparrow}}z]$$

in which a new-bound name enters a box boundary.

In more detail, for the subcalculus without new-binding the labelled transition rules are straightforward — instances of the reduction rule (Red Up) correspond to uses of (Box-1), (Out), and (Par); instances of (Red Down) correspond to uses of (Comm), (Out), and (Box-2); instances of (Red Comm) correspond to uses of (Comm), (Out), and (In). The derivations of the corresponding τ-transitions can be found in the proof of Lemma 24. The addition of new-binding introduces several subtleties, some inherited from the π-calculus and some related to scope extrusion and intrusion across box boundaries. We discuss the latter briefly.

The (Red Down) rule involves synchronisation on the box name n but *not* on the channel name x — there are reductions such as that above with new-bound names entering box boundaries. To correctly match this with a τ-transition the side-condition for (Res-2) for labels with output tag n requires the bound name to occur either in channel or value position, and the (Comm) rule reintroduces the x binder on the right hand side.

Similarly, the (Red Up) rule allows new-bound names in channel position to exit a box boundary, for example in

$$n[(\boldsymbol{\nu}\, x)\overline{x}^{\uparrow}z] \quad \rightarrow \quad (\boldsymbol{\nu}\, x)(\overline{x}^{\overline{n}}z \mid n[0])$$

The (Res-2) condition for output tag $\uparrow$ again requires the bound name to occur either in channel or value position; here the (Box-1) rule reintroduces the x binder on the right hand side.

Reductions generated by (Red Comm) involve synchronisation both on the tags and on the channel name. The (Res-2) condition for output tags $\star$, $\overline{\uparrow}$ and $\overline{n}$ is analogous to the standard π-calculus (Open) rule; requiring the bound name to occur in the value but not in the tag or channel. The (Comm) rule for these output tags is analogous to the standard π rule — in particular, here it is guaranteed that $x \in A$. The two semantics coincide in the following sense.

$$\frac{}{A \vdash \overline{x}^{\circ} v \xrightarrow{\overline{x}^{\circ} v} 0}\ \text{(Out)} \qquad \frac{}{A \vdash x^{\iota} p.P \xrightarrow{x^{\iota} v} \{^{v}/_{p}\}P}\ (c)\ \text{(In)}$$

$$\frac{A \vdash P \xrightarrow{\ell} P'}{A \vdash P \mid Q \xrightarrow{\ell} P' \mid Q}\ \text{(Par)} \qquad \frac{}{A \vdash \mathbf{!}\, x^{\iota} p.P \xrightarrow{x^{\iota} v} \mathbf{!}\, x^{\iota} p.P \mid \{^{v}/_{p}\}P}\ (c)\ \text{(Repl)}$$

$$\frac{A \vdash P \xrightarrow{\overline{x}^{\gamma} v} P' \quad A \vdash Q \xrightarrow{x^{\gamma} v} Q'}{A \vdash P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu}\, \mathrm{fn}(x, v) - A)(P' \mid Q')}\ \text{(Comm)}$$

$$\frac{A \vdash P \xrightarrow{\overline{x}^{\uparrow} v} P'}{A \vdash n[P] \xrightarrow{\tau} (\boldsymbol{\nu}\, \mathrm{fn}(x, v) - A)(\overline{x}^{\overline{n}} v \mid n[P'])}\ \text{(Box-1)}$$

$$\frac{}{A \vdash n[P] \xrightarrow{x^{\overline{n}} v} n[\overline{x}^{\overline{\uparrow}} v \mid P]}\ \text{(Box-2)} \qquad \frac{A \vdash P \xrightarrow{\tau} P'}{A \vdash n[P] \xrightarrow{\tau} n[P']}\ \text{(Box-3)}$$

$$\frac{A, x \vdash P \xrightarrow{\ell} P'}{A \vdash (\boldsymbol{\nu}\, x)P \xrightarrow{\ell} (\boldsymbol{\nu}\, x)P'}\ (a)\ \text{(Res-1)} \qquad \frac{A, x \vdash P \xrightarrow{\overline{y}^{\circ} v} P'}{A \vdash (\boldsymbol{\nu}\, x)P \xrightarrow{\overline{y}^{\circ} v} P'}\ (b)\ \text{(Res-2)}$$

$$\frac{A \vdash P \xrightarrow{\ell} P' \quad P' \equiv P''}{A \vdash P \xrightarrow{\ell} P''}\ \text{(Struct)}$$

(a) The (Res-1) rule is subject to $x \notin \mathrm{fn}(\ell)$. (b) The (Res-2) rule is subject to $x \in \mathrm{fn}(v) - \mathrm{fn}(y, o)$, if o is $\star$, $\overline{\uparrow}$ or $\overline{n}$, and to $x \in \mathrm{fn}(y, v) - \mathrm{fn}(o)$ otherwise. (c) In the (In) and (Repl) axioms there is a side condition that $\{^{v}/_{p}\}P$ is well-defined. In all rules with conclusion of the form $A \vdash P \xrightarrow{\ell} Q$ there is an implicit side condition $\mathrm{fn}(P) \subseteq A$. Symmetric versions of (Par) and (Comm) are elided.

Figure 6: Box-π Labelled Transition Semantics

Theorem 1 *If* $\mathrm{fn}(P) \subseteq A$ *then* $A \vdash P \xrightarrow{\tau} Q$ *iff* $P \to Q$.

This give confidence that the labelled semantics carries enough information. The proof is somewhat delicate; it is sketched in Appendix A and given in detail in [34].

Some auxiliary notation is useful. For a sequence of labels $\ell_1 \ldots \ell_k$ we write

$$A \vdash P_1 \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_k} P_{k+1}$$

to mean $\exists P_2, \ldots, P_k \,.\, \forall i \in 1..k \,.\, A_i \vdash P_i \xrightarrow{\ell_i} P_{i+1}$, where $A_i = A \cup \bigcup_{j \in 1..i} \mathrm{fn}(\ell_j)$. If $\ell \neq \tau$ we write $A \vdash P \stackrel{\hat{\ell}}{\Longrightarrow} P'$ for $A \vdash P \xrightarrow{\tau}{}^* \xrightarrow{\ell} \xrightarrow{\tau}{}^* P'$; if $\ell = \tau$ then $A \vdash P \stackrel{\hat{\ell}}{\Longrightarrow} P'$ is defined as $A \vdash P \xrightarrow{\tau}{}^* P'$, which we also write as $A \vdash P \Longrightarrow P'$.

3 Wrappers and Components in Box-π

This section gives four example wrappers. The first wrapper, $\mathcal{W}_1$, encapsulates a single component, restricting its interactions with the outside world to communications obeying a certain protocol. The second, $\mathcal{L}$, is similar, but also writes a log of all such communications. The third wrapper, $\mathcal{W}_2$, encapsulates two components, allowing each to interact with the outside world in a limited way but also allowing information to flow from the first to the second (but not vice versa) along an unordered pipeline. The fourth and most complex wrapper, $\mathcal{F}$, is similar to $\mathcal{W}_2$, but implements an ordered pipeline between the components.

Wrappers are designed in the context of some fixed protocols for interaction between components and their environment, interaction among components, and additional interaction between the environment and the wrapper (for logging or control). These protocols can be designed so that wrappers can be nested, allowing a complex security policy to be constructed from off-the-shelf wrappers. The example wrappers below all assume rather simple fixed protocols. As a trivial example, one can compose $\mathcal{W}_1$ with itself, with

$$(\mathcal{W}_1 \circ \mathcal{W}_1)\,(\_) \quad \stackrel{def}{=} \quad \mathcal{W}_1(\mathcal{W}_1(\_))$$

as its internal and external interfaces coincide. For more interesting composition, one would have to generalise to arbitrary sets of channels instead of *in* and *out*, and allow n-ary wrappers.

3.1 $\mathcal{W}_1$: A Simple Unary Filtering Wrapper

To demonstrate the use of box-π we give the definition of a wrapper that restricts the interface for user programs. In most operating systems, programs installed and run by a user enjoy the same access rights as the user, so if the user is allowed to open a socket and send data out on the network then so can any component. We idealize this scenario with the configuration below – an idealized single-user OS in which user Alice is executing a program P. Here the box around P stands for the user protection domain enforced by the operating system.

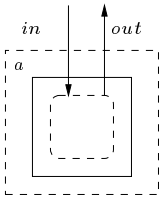
$alice[\, P \,] \mid$	
$!\ldots\overline{in}^{alice} x\ldots \mid$	OS write on Alice's *in* port
$!\, out^{alice} x\ldots \mid$	OS read from Alice's *out* port
$!\, net^{alice} x\ldots$	OS read from Alice's *net* port

The OS provides three channels in, out and net, to respectively allow the user's program to read from and write to the terminal and to send data out on a network connection. The program P is executing within a box and so interacts with the OS using the $\uparrow$ tag – for example $P = in^{\uparrow}x.\overline{out}^{\uparrow}\langle x\, x\rangle$ receives a value from the terminal and then sends a pair of copies of the value back to the terminal.

To execute some untrusted code fragment Q, Alice may run the code in parallel with her other applications, perhaps as $alice[P \mid Q]$. But, this grants too much privilege to Q. In particular, if $Q = !\, in^{\uparrow}x.\overline{net}^{\uparrow}x$ then any terminal input may be redirected to the net, or if $Q = !\, c^{\star}y.(\overline{net}^{\uparrow}c \mid \overline{c}^{\star}y)$ then Q can eavesdrop on communications on channel c between other parts of the system in P.

A wrapper is a box-π context which can enforce fine-grain control on the behaviour of Q. Our first example is the filtering wrapper $\mathcal{W}_1$, which prevents Q from accessing the network or from eavesdropping:

$$\begin{array}{rcl}\mathcal{W}_1(\_) & \stackrel{def}{=} & (\boldsymbol{\nu}\, a)\,(\ a[\ \_\ ] \\ & & \quad \mid\ !\, in^{\uparrow}x.\overline{in}^{a}x \\ & & \quad \mid\ !\, out^{a}x.\overline{out}^{\uparrow}x\ )\end{array}$$



The system becomes $alice[P \mid \mathcal{W}_1(Q)]$. The untrusted code is placed in a box with a fresh name a, so $a \notin \mathrm{fn}(Q)$. In parallel with the box are two forwarders for in and out messages. The first, $!\, in^{\uparrow}x.\overline{in}^{a}x$, is a replicated input receiving values from the OS and sending them to a; the second is dual. Only these two processes can interact with a due to the scope of the restriction, so even when put in parallel with other code the wrapper guarantees that Q will not be able to send on net.

We show a small reduction sequence where $P = 0$ and $Q = in^{\uparrow}x.\overline{net}^{\uparrow}x$. Here B is the forwarders $!\, in^{\uparrow}x.\overline{in}^{a}x \mid !\, out^{a}x.\overline{out}^{\uparrow}x$.

$$\begin{array}{rl}
 & \overline{in}^{alice}y \mid alice[P \mid \ \mathcal{W}_1(Q)\ ] \\
\equiv & \overline{in}^{alice}y \mid alice[\ \ (\boldsymbol{\nu}\, a)(\ \ a[\ \ Q] \mid B)] \\
\rightarrow & alice[\overline{in}^{\uparrow}y \mid (\boldsymbol{\nu}\, a)(\ \ a[\ \ Q] \mid B)] \\
\equiv & alice[\ \ (\boldsymbol{\nu}\, a)(\overline{in}^{\uparrow}y \mid \ a[\ \ Q] \mid B)] \\
\rightarrow & alice[\ \ (\boldsymbol{\nu}\, a)(\overline{in}^{a}y \mid \ a[\ \ Q] \mid B)] \\
\rightarrow & alice[\ \ (\boldsymbol{\nu}\, a)(\ \ a[\overline{in}^{\uparrow}y \mid Q] \mid B)] \\
\rightarrow & alice[\ \ (\boldsymbol{\nu}\, a)(\ \ a[\ \overline{net}^{\uparrow}y\ ] \mid B)] \\
\rightarrow & alice[\ \ (\boldsymbol{\nu}\, a)(\overline{net}^{a}y \mid a[\ \ 0\ \ ] \mid B)]
\end{array}$$

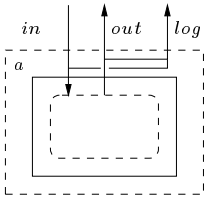
At the final state the output from Q is prevented from leaving the $alice$ box directly as B does not contain a forwarder for net. It is prevented from interaction with any P (although here P was empty) by the restriction on a.

3.2 $\mathcal{L}$: A Logging Wrapper

Wrappers can be used for monitoring as well as filtering; in operating systems auditing untrusted components is an important part of most security infrastructures. The $\mathcal{L}$ wrapper extends $\mathcal{W}_1$ to maintain a log of all communications of a process, sending

copies on a channel *log* to the environment, as follows:

$$\begin{array}{rcl}\mathcal{L}(\_) & \stackrel{def}{=} & (\boldsymbol{\nu}\, a)\big(\quad a[\_] \\ & & \quad\ \mid !\, in^{\uparrow} y.(\overline{log}^{\uparrow} y \mid \overline{in}^{a} y) \\ & & \quad\ \mid !\, out^{a} y.(\overline{log}^{\uparrow} y \mid \overline{out}^{\uparrow} y)\ \big)\end{array}$$

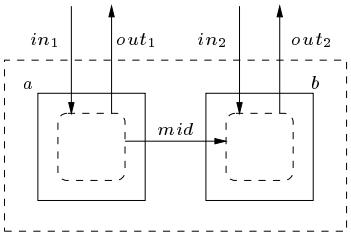


Note that $\mathcal{L}$ does not interfere with the operation of the component it encapsulates; the logging activity is transparent.

3.3 $\mathcal{W}_2$: A Pipeline Wrapper

A pipeline wrapper allows a controlled flow of information between two components. We give a binary wrapper $\mathcal{W}_2$ that encapsulates two processes. In an execution of $\mathcal{W}_2(Q_1, Q_2)$ the two wrapped processes Q_i can interact with the environment as before, on channels in_i and out_i. In addition, Q_1 can send messages to Q_2 on a channel *mid*.

$$\begin{array}{rcl}\mathcal{W}_2(\__1, \__2) & \stackrel{def}{=} & (\boldsymbol{\nu}\, a, b)\big(\quad a[\__1] \mid b[\__2] \\ & & \quad\ \mid !\, {in_1}^{\uparrow} y.\overline{in_1}^{a} y \\ & & \quad\ \mid !\, {in_2}^{\uparrow} y.\overline{in_2}^{b} y \\ & & \quad\ \mid !\, {out_1}^{a} y.\overline{out_1}^{\uparrow} y \\ & & \quad\ \mid !\, {out_2}^{b} y.\overline{out_2}^{\uparrow} y \\ & & \quad\ \mid !\, mid^{a} y.\overline{mid}^{b} y\ \big)\end{array}$$



As before $\mathcal{W}_2$ is a non-binding context – we assume, wherever we apply it to two processes Q_1, Q_2, that $\{a, b\} \cap \mathrm{fn}(Q_1, Q_2) = \emptyset$. For an example of a blocked attempt by the second process to send a value to the first, suppose $Q_2 = \overline{mid}^{\uparrow} v$. We have

$$\begin{array}{rcl}\mathcal{W}_2\Big(Q_1, \overline{mid}^{\uparrow} v\Big) & = & (\boldsymbol{\nu}\, a, b)\big(a[Q_1] \mid b[\overline{mid}^{\uparrow} v] \mid R\big) \\ & \to & (\boldsymbol{\nu}\, a, b)\big(a[Q_1] \mid b[0] \mid \overline{mid}^{b} v \mid R\big)\end{array}$$

where R is the parallel composition of forwarders. The output $\overline{mid}^{b} v$ in the final state cannot interact further – not with the environment, as b is restricted, and not with the forwarder $!\, mid^{a} y.\overline{mid}^{b} y$, as $a \neq b$.

3.4 $\mathcal{F}$: An Ordered Pipeline Wrapper

There is a tension between the strength of communication primitive supported by a wrapper and the strength of the security property it can guarantee. The previous three examples provide only asynchronous unordered communication between components, which would be awkward to use in most real systems. At the other extreme, synchronous communication introduces causal flows in both directions (the causal flow property we state in Section 4.5 would not hold in a synchronous calculus, so a more delicate property would be required – perhaps stating that there are only data-less acks from one component to another). There are two intermediate points: one can provide asynchronous ordered communication, as we do below, or use some form of

$$
\begin{array}{lll}
\mathcal{F}(\__1, \__2) & = (\boldsymbol{\nu}\, a, b)\big(& a[\__1] \mid b[\__2] \mid \\
 & & (\boldsymbol{\nu}\, \mathit{buff}, \mathit{full})\big(\\
 & & \quad (\boldsymbol{\nu}\, \mathit{front}, \mathit{back})\big(\\
\text{(create FIFO buffer)} & & \quad\quad \overline{\mathit{buff}}^{\star}\langle \mathit{front}\ \mathit{back}\rangle \mid \\
\text{(connect } \mathit{from}^a \text{ to buffer)} & & \quad\quad !\, \mathit{from}^a(v\, r).(\boldsymbol{\nu}\, r')(\overline{\mathit{front}}^{\star}\langle v\, r'\rangle \mid r'^{\star}.\overline{r}^a) \mid \\
\text{(connect buffer to } \mathit{to}^b) & & \quad\quad !\, \mathit{back}^{\star}(v\, r).(\boldsymbol{\nu}\, r')(\overline{\mathit{to}}^b\langle v\, r'\rangle \mid r'^b.\overline{r}^{\star})\big) \mid \\
\text{(buffer code)} & & !\, \mathit{buff}^{\star}(\mathit{front}\ \mathit{back}).\mathit{front}^{\star}(v\, r).(\overline{r}^{\star} \mid (\boldsymbol{\nu}\, \mathit{back}')(\overline{\mathit{buff}}^{\star}\langle \mathit{front}\ \mathit{back}'\rangle \\
 & & \qquad\qquad \mid \overline{\mathit{full}}^{\star}\langle \mathit{back}'\ \mathit{back}\ v\rangle)) \mid \\
 & & !\, \mathit{full}^{\star}(\mathit{back}'\ \mathit{back}\ v).(\boldsymbol{\nu}\, r)(\overline{\mathit{back}}^{\star}\langle v\, r\rangle \mid r^{\star}.\mathit{back}'^{\star}\langle v'\, r'\rangle.(\overline{r'}^{\star} \\
 & & \qquad\qquad \mid \overline{\mathit{full}}^{\star}\langle \mathit{back}'\ \mathit{back}\ v'\rangle))\big) \mid \\
\text{(I/O forwarders)} & & !\, \mathit{in}_1{}^{\uparrow}x.\overline{\mathit{in}_1}^a x \mid !\, \mathit{out}_1{}^a x.\overline{\mathit{out}_1}^{\uparrow} x \mid \\
 & & !\, \mathit{in}_2{}^{\uparrow}x.\overline{\mathit{in}_2}^b x \mid !\, \mathit{out}_2{}^b x.\overline{\mathit{out}_2}^b x\big)
\end{array}
$$

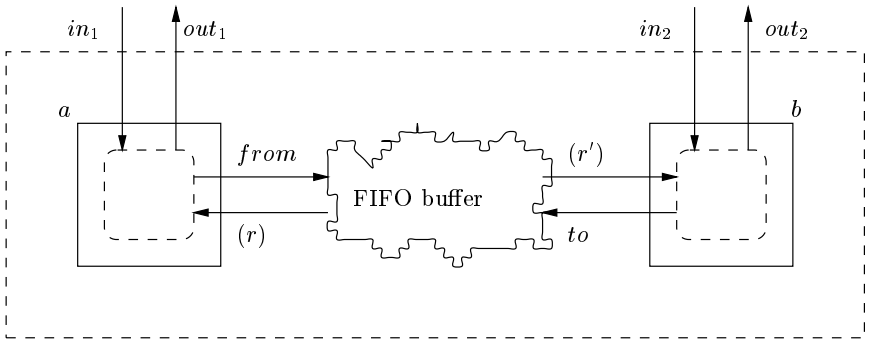


Figure 7: FIFO Pipeline Wrapper $\mathcal{F}$.

weak acknowledgments, as in the NRL pump [23]. The former still guarantees an absence of information flow (albeit at the cost of maintaining an unbounded buffer) while the latter limits bandwidth of covert channels. In both cases, it is essential to be able to guarantee that the implementation of the communication primitives does actually have the desired flow property; this is what we set out to do here.

In Figure 7 we give a wrapper $\mathcal{F}$ that takes two components and allows the first to communicate with the second by a first-in, first-out buffer. The wrapper has been written with care to avoid any information leak from the second component to the first. For simplicity both components have simple unordered input and output ports in_i and out_i to the environment; it would be routine to make these FIFO also.
The interface to the wrapper is as follows. To write to the buffer a producer sends a value together with an acknowledgment channel to the wrapper (using a standard asynchronous π-calculus idiom). The wrapper inserts the value in a queue and acknowledges reception. For value v the producer may contain

$$(\boldsymbol{\nu}\, \mathit{ack})(\overline{\mathit{from}}^{\uparrow}\langle v\, \mathit{ack}\rangle \mid \mathit{ack}^{\uparrow}\ldots),$$

sending the value and a new acknowledgement channel ack to the wrapper and, in parallel, waiting for a reply before proceeding with its computation. On the receiver

side, we may have a process that waits for a pair of a value and an ack channel:

$$to^{\uparrow}(z\,r).(\overline{r}^{\uparrow} \mid \ldots)$$

The name of the receiving channel is *to*; channel r is used to send the acknowledgement back to the wrapper. Thus a configuration where B stands for the body of the wrapper could be:

$$\begin{array}{l}(\boldsymbol{\nu}\, a, b)\big(\ a[\ (\boldsymbol{\nu}\, ack)(\overline{from}^{\uparrow}\langle v\, ack\rangle \mid ack^{\uparrow}.0)\ ] \mid \\ \qquad\quad b[\ to^{\uparrow}(z\,r).\overline{r}^{\uparrow}\ ] \mid \\ \qquad\quad B\ \big)\end{array}$$

The implementation of the wrapper is somewhat tricky, as we have to be careful not to introduce covert channels between the components. Within the wrapper there is a replicated input on *buff* that creates a new empty FIFO buffer and a replicated input on *full* that creates a new buffer cell containing a value. The key point is to ensure that the acknowledgment to the first component not be dependent on any action performed by the second component. The glue process that connects the $from^a$ channel to the buffer has a subprocess, $r'^{\star}.\overline{r}^{a}$, to send the ack to a. This small process itself expects an ack from the head of the buffer saying that the message was inserted in the queue. The buffer code $front^{\star}(v\,r).(\overline{r}^{\star}\ldots$ acks on r immediately, in parallel with placing the new message in a full buffer cell at the head of the queue. The asynchrony here is essential.

4 Security Properties

So far we have been vague about the statement of the properties that we expect wrappers to enforce. For $\mathcal{W}_1$ it may be clear from examination of the code and the semantics that the wrapper is satisfactory, but it is unclear exactly what properties are guaranteed. For $\mathcal{F}$ the situation is much worse – even this simple wrapper is complex enough that a rigorous statement and proof of its security properties is essential; the user should not be required to examine the code of a wrapper in order to understand the security that it provides. We now turn to the task of formalizing these properties and developing the tools needed to prove them.

4.1 Purity

The most basic questions that one would expect a theory of wrappers to deal with are whether a component is *well-behaved* and, for certain wrappers, whether wrapping an ill-behaved component creates a well-behaved component. Statements of such properties must be with respect to a particular choice of protocol for legitimate wrapper/component interaction. For example, for the unary wrappers $\mathcal{W}_1$ and $\mathcal{L}$ a well-behaved component is expected to interact only on *in* and *out* channels with its parent. This can be easily formalised using our labelled transition semantics: we say an *interface* M is a set of pairs m of a (co)name and a tag, e.g. $M = \{in^{\uparrow}, \overline{out}^{\uparrow}\}$.

Definition 1 *A process P is* well-behaved *for an interface M iff whenever $A \vdash P \xrightarrow{l_1..l_k} Q$ then for each $j \in 1..k$ we have $l_j = \tau$ or $\exists m \in M, v\ .\ l_j = mv$.*

Recalling the examples of Section 3.1,

$$\begin{array}{rcl} P & = & in^{\uparrow}x.\overline{out}^{\uparrow}\langle x\,x\rangle \\ Q & = & !\,in^{\uparrow}x.\overline{net}^{\uparrow}x \end{array}$$

P is well-behaved for interface $\{in^{\uparrow}, \overline{out}^{\uparrow}\}$ (it has transitions only with labels of the forms τ, $in^{\uparrow}v$ or $\overline{out}^{\uparrow}v$) but Q is not.

Irrespective of the behaviour of a component R, wrapper $\mathcal{W}_1$ enforces good behavior, thus $\mathcal{W}_1(R)$ does obey the protocol – again this can be stated clearly using the LTS:

Proposition 2 *For any program R with $a \notin \mathrm{fn}(R)$, $\mathcal{W}_1(R)$ is well-behaved for $\{in^{\uparrow}, \overline{out}^{\uparrow}\}$.*

We say a unary wrapper with this property is *pure*. The proof is via an explicit characterisation of the states reachable by labelled transitions of $\mathcal{W}_1(R)$; it can be found in Appendix B.

The logging wrapper $\mathcal{L}$ is not pure in this sense, but a wrapped program $\mathcal{L}(R)$ can again interact only in limited ways.

Proposition 3 *For any program R with $a \notin \mathrm{fn}(R)$, $\mathcal{L}(R)$ is well-behaved for $\{in^{\uparrow}, \overline{out}^{\uparrow}, \overline{log}^{\uparrow}\}$.*

For an analogous notion of purity for binary wrappers with interfaces such as $\mathcal{W}_2$, say a binary wrapper $\mathcal{C}$ is pure iff for any programs R_1, R_2, (satisfying the appropriate free name condition, i.e. that with $\{a, b\} \cap \mathrm{fn}(R_1, R_2) = \emptyset$), $\mathcal{C}(R_1, R_2)$ is well-behaved for $\{in_1^{\uparrow}, \overline{out_1}^{\uparrow}, in_2^{\uparrow}, \overline{out_2}^{\uparrow}\}$.

Proposition 4 *Binary wrappers $\mathcal{W}_2$ and $\mathcal{F}$ are pure.*

Propositions 3 and 4 can be proved either via explicit characterisations similar to that of Proposition 2 or using the type system developed later.

4.2 Honesty

The properties of wrappers stated in the previous subsection are weak as they hardly constrain the behaviour of the wrapper. For example, the useless unary wrapper

$$\mathcal{C}(\_) \stackrel{def}{=} 0$$

is trivially pure as it inhibits all interactions. In [34] we introduced the class of *honest* wrappers that are guaranteed to forward legitimate messages. An initial attempt at a definition of honesty might be to take $\mathcal{W}_1$ as a specification, defining a unary wrapper $\mathcal{C}$ to be honest iff for any program P the processes $\mathcal{C}(P)$ and $\mathcal{W}_1(P)$ are operationally equivalent. This is unsatisfactory – it rules out wrappers such as $\mathcal{L}$, and it does not give a very clear statement of the properties that may be assumed of an honest wrapper.

A better attempt might be to say that a unary wrapper $\mathcal{C}$ is honest iff for any well-behaved P the processes $\mathcal{C}(P)$ and P are operationally equivalent. This would be unsatisfactory in two ways. Firstly, some intuitively sound wrappers have additional interactions with the environment – e.g. the logging outputs of $\mathcal{L}$ – and so would not be considered honest by this definition. More seriously, this definition would not constrain the behaviour of wrappers for non-well-behaved P at all – if a component P attempted, in error, a single illicit communication then $\mathcal{C}(P)$ might behave arbitrarily.

To address these points we gave an explicit definition of honesty, in the style of weak asynchronous bisimulation [3], for unary wrappers such as $\mathcal{W}_1$ and $\mathcal{L}$.

Definition 2 (Honesty) *Consider a family of relations R indexed by finite sets of names such that each R_A is a relation over $\{ P \mid \mathrm{fn}(P) \subseteq A \}$. Say R is an* h-bisimulation *if, whenever $C\ R_A\ Q$ then:*

1. *if $A \vdash C \xrightarrow{\ell} C'$ for $\ell = \overline{out}^{\uparrow} v, \tau$ then $A \vdash Q \stackrel{\hat{\ell}}{\Longrightarrow} Q' \wedge C'\ R_{A \cup \mathrm{fn}(\ell)}\ Q'$*
2. *if $A \vdash C \xrightarrow{in^{\uparrow} v} C'$ then either $A \vdash Q \stackrel{in^{\uparrow} v}{\Longrightarrow} Q'$ and $C'\ R_{A \cup \mathrm{fn}(in,v)}\ Q'$ or $A \vdash Q \Longrightarrow Q'$ and $C'\ R_{A \cup \mathrm{fn}(in,v)}\ Q' \mid \overline{in}^{\uparrow} v$*
3. *if $A \vdash C \xrightarrow{\ell} C'$ for any other label then $C'\ R_{A \cup \mathrm{fn}(\ell)}\ Q$*

together with symmetric versions of clauses 1 and 2. Say a unary wrapper $\mathcal{C}$ is honest *if for any program P (satisfying the appropriate free name condition) and any $A \supseteq \mathrm{fn}(\mathcal{C}(P))$ there is an h-bisimulation R with $\mathcal{C}(P)\ R_A\ P$.*

Loosely, clauses 1, 2 and the symmetric versions ensure that legitimate communications and internal reductions must be weakly matched. Clause 3 ensures that if the wrapper performs some additional communication then this does not affect the state as seen by the wrapped process.

Proposition 5 *The unary wrapper $\mathcal{W}_1$ is honest.*

We conjecture that $\mathcal{L}$ is also honest. We give some examples of dishonest wrappers. Take

$$\mathcal{C}(\_) \stackrel{def}{=} (\boldsymbol{\nu}\, a) a[\_]$$

This is not honest – a transition $A \vdash P \xrightarrow{\overline{out}^{\uparrow} v} P'$ cannot be matched by $\mathcal{C}(P)$, violating the symmetric version of clause 1. Now consider

$$\mathcal{C}(\_) \stackrel{def}{=} \_$$

This wrapper is also dishonest as $\mathcal{C}(P)$ can perform actions not in the protocol that essentially affect the state of P. For example, take $P = x^{\star} y.\overline{out}^{\uparrow}\langle\rangle$. Suppose $\mathcal{C}(P)\ R_A\ P$ for an h-bisimulation R. We have $A \vdash \mathcal{C}(P) \xrightarrow{x^{\star}\langle\rangle} \overline{out}^{\uparrow}\langle\rangle$ so by clause 3 $\overline{out}^{\uparrow}\langle\rangle\ R_A\ P$, but then clause 1 cannot hold – the left hand side can perform an $\overline{out}^{\uparrow}\langle\rangle$ transition that cannot be matched be the right hand side.

A definition of honesty for binary wrappers must take into account the legitimate interactions between the two components. In [34] we gave a tentative definition, in terms of *binary h-bisimulations*, but it was rather complex – dealing with the combination of the $\mathcal{W}_2$ protocol and the asynchrony of the calculus. We regard it as an open problem to give satisfactory definitions of honesty for complex wrappers and of an operational equivalence $\approx$, concluding this subsection with some desirable relationships between them.

The protocol for communication between a component and a unary wrapper is designed so that wrappers may be nested. We conjecture that the composition of any honest unary wrappers is honest.

Conjecture 6 *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are honest unary wrappers then $\mathcal{C}_1 \circ \mathcal{C}_2$ is honest.*

Analogous results for non-unary wrappers would require wrappers with more complex interfaces so that the input, output and mid channels could be connected correctly.

A desirable property of a pure wrapper is that it should not affect the behaviour of any well-behaved component — one might expect for any pure and honest $\mathcal{C}$ and well-behaved P that $P \approx \mathcal{C}(P)$ for any reasonable operational equivalence $\approx$. Unfortunately this does not hold for the obvious naive adaptation of weak asynchronous bisimulation to box-π, even for $\mathcal{W}_1$, as the wrapper can make input transitions that cannot be matched – a more refined equivalence is required.

A simpler property would be that multiple wrappings have no effect. We would expect that $\mathcal{W}_1$ is idempotent, i.e. that $\mathcal{W}_1(\mathcal{W}_1(P)) \approx \mathcal{W}_1(P)$, for any reasonable equivalence $\approx$.

4.3 Unidirectional Information Flow: First Attempts

Honesty and Purity are desirable properties, but they do not address our central problem: that of understanding in what sense a multi-hole wrapper such as $\mathcal{W}_2$ or $\mathcal{F}$ allows fine-grain control over the information flows between components – enforcing the unidirectional flow property that the second wrapped component should not be able to affect the first. By examining the code for $\mathcal{W}_2$ it is intuitively clear that information can not flow from Q to P within $\mathcal{W}_2(P,Q)$. For $\mathcal{F}$ it is much less obvious, however, and when one comes to make the intuition precise it becomes far from clear exactly what property is desired. Moreover, the user should not have to examine the wrapper code in order to get sufficient guarantees about its behaviour.

In this subsection we define two information flow properties expressed using the LTS: *new name directionality* and *permutation*. Neither is satisfactory; we argue that a more intensional semantics is required. The following two subsections develop a *coloured labelled transition semantics* and state a *causal flow property* in terms of it. Together, these definitions illustrate the wide range of precise properties which the intuitive statement might be thought to mean. We hope to provoke discussion of exactly what guarantees should be desired by users and by component designers.

For simplicity, only pure binary wrappers $\mathcal{C}$ are considered.

New-name directionality As we are using a calculus with creation of new names, we can test a wrapper by supplying a new name to the second component, on in_2, and observing whether it can ever be output by the first component on out_1. Say $\mathcal{C}$ is *directional for new names* if whenever

$$A \vdash \mathcal{C}(P_1,P_2) \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_j} \xrightarrow{in_2{}^{\uparrow}u} \xrightarrow{\ell'_1} \ldots \xrightarrow{\ell'_k} \xrightarrow{\overline{out_1}{}^{\uparrow}u'} P$$

with $x \in \mathrm{fn}(u)$, but x is new, i.e. $x \notin A \cup \mathrm{fn}(\ell_1 \ldots \ell_j)$, and x is not subsequently input to the first component, i.e.

$$x \notin \bigcup_{i \in 1..k \wedge \ell'_i = in_1{}^{\uparrow}v} \mathrm{fn}(v)$$

then x is not output by the first component, i.e. $x \notin \mathrm{fn}(u')$. This property does not prevent all information flow, however – a variant of $\mathcal{W}_2$ containing a reverse-forwarder that only forwards particular values, such as

$$!\,mid^{a_2}y.\mathbf{if}\ y \in \{0,1\}\ \mathbf{then}\ \overline{mid}^{a_1}y$$

could still satisfy it. (Here 0 and 1 are free names, which must therefore be in A.)

Permutation Our second property formalises the intuition that if no observable behaviour due to P_1 depends on the behaviour of P_2 then in any trace it should be possible to move the actions associated with P_1 before all actions associated with P_2. Say $\mathcal{C}$ has the *permutation property* if whenever

$$A \vdash \mathcal{C}(P_1, P_2) \stackrel{\ell_1}{\Longrightarrow} \ldots \stackrel{\ell_k}{\Longrightarrow} P$$

with $\ell_i \neq \tau$ there exists a permutation π of $\{1, \ldots, k\}$ such that

$$A \vdash \mathcal{C}(P_1, P_2) \stackrel{\ell_{\pi(1)}}{\Longrightarrow} \ldots \stackrel{\ell_{\pi(k)}}{\Longrightarrow} P$$

and no in_1 or out_1 transition occurs after any in_2 or out_2 transition in $\ell_{\pi(1)} \ldots \ell_{\pi(k)}$. Permutation ensures that actions of P_1 do not depend on inputs of P_2 but it does not prevent initial interactions between the components.

For an example wrapper without this property, consider a wrapper $\mathcal{C}$ which forces inputs of P_1 to be causally dependent on inputs of P_2.

$$\begin{aligned}\mathcal{C}(\__1, \__2) \stackrel{def}{=} \quad & (\boldsymbol{\nu}\, a_1, a_2)\big(a_1[\__1] \mid a_2[\__2] \\ & \mid\, !\, in_2{}^{\uparrow} y.(\overline{in_2}^{a_2} y \mid !\, in_1{}^{\uparrow} y.\overline{in_1}^{a_1} y\big) \\ & \mid\, !\, out_1{}^{a_1} y.\overline{out_1}^{\uparrow} y \\ & \mid\, !\, out_2{}^{a_2} y.\overline{out_2}^{\uparrow} y \\ & \mid\, !\, mid^{a_1} y.\overline{mid}^{a_2} y\big)\end{aligned}$$

Here the in_1 messages are not forwarded until at least one in_2 input is received from the environment. Nonetheless, in some sense there is still no information flow from the second component to the first.

The new-name directionality and permutation properties are expressed purely in terms of the externally observable behaviour of $\mathcal{C}(P, Q)$ (in fact, they are properties of its trace set, a very extensional semantics). Note, however, that the intuitive statement that information does not flow from Q to P depends on an understanding of the internal computation of P and Q that is not present in the reduction or labelled transition relations (given only that $\mathcal{C}(P, Q) \rightarrow^* R$ there is no way to associate subterms of R with an 'origin' in $\mathcal{C}$, P or Q). We therefore develop a more intensional semantics in which output and input processes are tagged with sets of colours. The semantics propagates colours in interaction steps, thereby tracking the causal dependencies between interactions.

4.4 Colouring the Box-π Calculus

We introduce two semantics for capturing the intuitive property that one wrapped component does not causally affect another. First, we define a simple *coloured reduction semantics* for box-π which annotates output processes with sets of colours that record their causal histories – essentially the sets of principals that have affected them in the past – and the reduction semantics propagate this causal history data. Secondly, we introduce a coloured labelled transition semantics, allowing more direct statements of security properties of wrappers that interact with their environment. The coloured calculus is a trade-off – it captures less detailed causality information than the non-interleaving models studied in concurrency theory [45, 5, 9] but is much simpler; it captures enough information to express interesting security properties.

The coloured syntax. We take a set col of *colours* or *principals* (we use the terms interchangeably) disjoint from $\mathcal{N}$. Let $\mathsf{k}, \mathsf{p}, \mathsf{q}$ range over elements of col and $\mathsf{C}, \mathsf{D}, \mathsf{K}$ range over subsets of col. We define a coloured box-π calculus by annotating all outputs with sets of colours:

$$P \quad ::= \quad \mathsf{C}{:}\overline{x}^{o}v \;\mid\; x^{\iota}p.P \;\mid\; !\,x^{\iota}p.P \;\mid\; n[P] \;\mid\; 0 \;\mid\; P \mid P' \;\mid\; (\boldsymbol{\nu}\, x)P$$

If P is a coloured term we write $|P|$ for the term of the original syntax obtained by erasing all annotations. Conversely, for a term P of the original syntax $\mathsf{C} \circ P$ denotes the term with every particle coloured by C. For a coloured P we write $\mathsf{C} \bullet P$ for the coloured term which is as P but with C unioned to every set of colours occurring in it. We sometimes confuse p and the set $\{\mathsf{p}\}$. Let $\mathrm{pn}(P)$ be the set of colours that occur in P. We write CD for the union $\mathsf{C} \cup \mathsf{D}$.

In the coloured output $\mathsf{C} : \overline{x}^{o}v$ think of C as recording the causal history of the output particle – C is the set (possibly empty) of principals $\mathsf{p} \in \mathsf{C}$ that have affected the particle in the past. In an initial state all outputs might typically be coloured by singleton sets giving their actual principals, for example colouring the code of wrapper $\mathcal{F}$ and two wrapped components with different colours $\mathsf{w}, \mathsf{p}, \mathsf{q}$:

$$(\mathsf{w} \circ \mathcal{F})\,(\mathsf{p} \circ P \mid \mathsf{q} \circ Q)$$

The coloured reduction semantics is obtained by replacing the first four axioms of the uncoloured semantics by the rules

$$\begin{array}{ll}
n[\mathsf{C}{:}\overline{x}^{\uparrow}v \mid Q] \longrightarrow \mathsf{C}{:}\overline{x}^{\overline{n}}v \mid n[Q] & \text{(C Red Up)} \\
\mathsf{C}{:}\overline{x}^{n}v \mid n[Q] \longrightarrow n[\mathsf{C}{:}\overline{x}^{\uparrow}v \mid Q] & \text{(C Red Down)} \\
\mathsf{C}{:}\overline{x}^{\overline{\iota}}v \mid x^{\iota}p.P \longrightarrow \mathsf{C} \bullet (\{^{v}/_{p}\}P) & \text{(C Red Comm)} \\
\mathsf{C}{:}\overline{x}^{\overline{\iota}}v \mid !\,x^{\iota}p.P \longrightarrow !\,x^{\iota}p.P \mid \mathsf{C} \bullet (\{^{v}/_{p}\}P) & \text{(C Red Repl)}
\end{array}$$

that propagate colour sets. The coloured calculus has essentially the same reduction behaviour as the original calculus:

Proposition 7 *For any coloured P we have $|P| \to Q$ iff $\exists P' \,.\, P \longrightarrow P' \wedge |P'| = Q$.*

The proof is by straightforward induction on the derivation of transitions.

The coloured labelled transitions have labels ℓ exactly as before. The coloured labelled transition relation has the form

$$A \vdash P \xrightarrow{\ell}_{\mathsf{C}} Q$$

where A is a finite set of names and $\mathrm{fn}(P) \subseteq A$; it should be read as 'in a state where the names A may be known to P and its environment, process P can do ℓ, *coloured* C, to become Q'. Again C records causal history, giving all the principals which have directly or indirectly contributed to this action. The relation is defined as the smallest relation satisfying the rules in Figure 8. It coincides with the previous LTS and with the coloured reduction semantics in the following senses.

Proposition 8 *For any coloured P we have $A \vdash |P| \xrightarrow{\ell} Q$ iff $\exists \mathsf{C}, P' \,.\, A \vdash P \xrightarrow{\ell}_{\mathsf{C}} P' \wedge |P'| = Q$.*

$$\frac{}{A \vdash \mathsf{C}{:}\overline{x}^{o}v \xrightarrow{\overline{x}^{o}v}_{\mathsf{C}} 0}\ \text{(Out)} \qquad \frac{}{A \vdash x^{\iota}p.P \xrightarrow{x^{\iota}v}_{\mathsf{C}} \mathsf{C}\bullet\{^{v}/_{p}\}P}\ (c)\ \text{(In)}$$

$$\frac{A \vdash P \xrightarrow{\ell}_{\mathsf{C}} P'}{A \vdash P \mid Q \xrightarrow{\ell}_{\mathsf{C}} P' \mid Q}\ \text{(Par)} \qquad \frac{}{A \vdash \mathbf{!}\,x^{\iota}p.P \xrightarrow{x^{\iota}v}_{\mathsf{C}} \mathbf{!}\,x^{\iota}p.P \mid \mathsf{C}\bullet\{^{v}/_{p}\}P}\ (c)\ \text{(Repl)}$$

$$\frac{A \vdash P \xrightarrow{\overline{x}^{\gamma}v}_{\mathsf{C}} P' \quad A \vdash Q \xrightarrow{x^{\gamma}v}_{\mathsf{C}} Q'}{A \vdash P \mid Q \xrightarrow{\tau}_{\emptyset} (\boldsymbol{\nu}\,\mathrm{fn}(x,v) - A)(P' \mid Q')}\ \text{(Comm)}$$

$$\frac{A \vdash P \xrightarrow{\overline{x}^{\uparrow}v}_{\mathsf{C}} P'}{A \vdash n[P] \xrightarrow{\tau}_{\emptyset} (\boldsymbol{\nu}\,\mathrm{fn}(x,v) - A)(\mathsf{C}{:}\overline{x}^{\overline{n}}v \mid n[P'])}\ \text{(Box-1)}$$

$$\frac{}{A \vdash n[P] \xrightarrow{x^{\overline{n}}v}_{\mathsf{C}} n[\mathsf{C}{:}\overline{x}^{\overline{\uparrow}}v \mid P]}\ \text{(Box-2)} \qquad \frac{A \vdash P \xrightarrow{\tau}_{\mathsf{C}} P'}{A \vdash n[P] \xrightarrow{\tau}_{\mathsf{C}} n[P']}\ \text{(Box-3)}$$

$$\frac{A, x \vdash P \xrightarrow{\ell}_{\mathsf{C}} P'}{A \vdash (\boldsymbol{\nu}\,x)P \xrightarrow{\ell}_{\mathsf{C}} (\boldsymbol{\nu}\,x)P'}\ (a)\ \text{(Res-1)} \qquad \frac{A, x \vdash P \xrightarrow{\overline{y}^{o}v}_{\mathsf{C}} P'}{A \vdash (\boldsymbol{\nu}\,x)P \xrightarrow{\overline{y}^{o}v}_{\mathsf{C}} P'}\ (b)\ \text{(Res-2)}$$

$$\frac{A \vdash P \xrightarrow{\ell}_{\mathsf{C}} P' \quad P' \equiv P''}{A \vdash P \xrightarrow{\ell}_{\mathsf{C}} P''}\ \text{(Struct)}$$

(a) The (Res-1) rule is subject to $x \notin \mathrm{fn}(\ell)$. (b) The (Res-2) rule is subject to $x \in \mathrm{fn}(v) - \mathrm{fn}(y, o)$, if o is $\star$, $\overline{\uparrow}$ or $\overline{n}$, and to $x \in \mathrm{fn}(y, v) - \mathrm{fn}(o)$ otherwise. (c) In the (In) and (Repl) axioms there is a side condition that $\{^{v}/_{p}\}P$ is well-defined. In all rules with conclusion of the form $A \vdash P \xrightarrow{\ell}_{\mathsf{C}} Q$ there is an implicit side condition $\mathrm{fn}(P) \subseteq A$. Symmetric versions of (Par) and (Comm) are elided.

Figure 8: Coloured Box-π Labelled Transition Semantics

The proof is by straightforward induction on the derivation of labelled transitions.

Proposition 9 *For coloured P and Q, if* $\mathrm{fn}(P) \subseteq A$ *then* $A \vdash P \xrightarrow{\tau}_{\emptyset} Q$ *iff* $P \to Q$.

The proof is a minor adaptation of that of Theorem 1.

4.5 Unidirectional Flow: The Causal Flow Property

A more convincing property can now be stated. Say an *instantiation* of some binary wrapper $\mathcal{W}$ is an uncoloured process $\mathcal{W}(P,Q)$ where P and Q are uncoloured processes not containing the new-bound names scoping the holes of $\mathcal{W}$. As before, say $\mathcal{W}$ is a *pure binary wrapper* if for any instantiation and any transition sequence

$$A \vdash \mathcal{W}(P,Q) \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_k} R$$

the labels ℓ_j have the form τ, ${in_i}^{\uparrow} v$, or $\overline{out_i}^{\uparrow} v$, for $i \in \{1,2\}$. (purity simply means that the wrapper has a fixed interface and thus simplifies the statement of the causal flow property).

Definition 3 (Causal flow property) *A pure binary wrapper $\mathcal{W}$ has the* causal flow property *if for any instantiation $\mathcal{W}(P,Q)$ and any coloured trace*

$$A \vdash \emptyset \circ \mathcal{W}(P,Q) \xrightarrow{\ell_1}_{\mathsf{C}_1} \ldots \xrightarrow{\ell_k}_{\mathsf{C}_k},$$

such that all input transitions ${in_1}^{\uparrow} v$ and ${in_2}^{\uparrow} v$ in $\ell_1..\ell_k$ are coloured with principal sets $\{\mathsf{p}\}$ and $\{\mathsf{q}\}$ respectively, we have $\ell_j = \overline{out_1}^{\uparrow} v$ implies that $\mathsf{q} \notin \mathsf{C}_j$.

This property forbids any causal flow from an input on in_2 to an output on out_1.

Different variants of the flow property, with different characteristics, can also be stated. For example, to prevent information in the initial state of Q affecting outputs on out_1 we could consider coloured traces

$$A \vdash (\emptyset \circ \mathcal{W})(\mathsf{p} \circ P, \mathsf{q} \circ Q) \xrightarrow{\ell_1}_{\mathsf{C}_1} \ldots \xrightarrow{\ell_k}_{\mathsf{C}_k}$$

This still allows the Q to communicate with P but only on the condition that P does not perform any further output dependent on the communicated values. Forbidding Q affecting P at all (even if there are no inputs or outputs of either component) can be done with a slightly more intricate coloured semantics. There is no clear cut 'best' solution, yet the use of causal semantics allows succinct statement of the alternatives and eases the comparison of these different properties.

5 Causality Types

Verifying a causal flow property directly can be laborious, requiring a characterisation of the state space of a wrapper containing arbitrary components. We therefore introduce a type system that statically captures causal flows; a wrapper can be shown to satisfy the causal flow property simply by checking that it is well-typed. Often (though not always) one might start with uncoloured terms; colours are propagated into the terms duing labelled transitions. The type system captures invariants about

how colours can propagate – the causal flow property is a corollary of the subject reduction theorem for the type system. This section introduces the type system, gives its soundness theorems, and applies it to $\mathcal{F}$.

A simple type system for Box-π would have types

$$T ::= \mathbf{chan}\, T \mid \mathbf{box} \mid \langle T \,..\, T \rangle$$

for the types of channel names carrying T, box names, and tuples. We annotate the first two by sets K of principals and add a type **name**, of arbitrary names, and $\top$, of arbitrary values, giving the *value types*

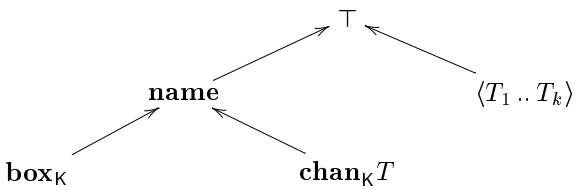
$$T ::= \mathbf{chan}_{\mathsf{K}} T \mid \mathbf{box}_{\mathsf{K}} \mid \langle T \,..\, T \rangle \mid \mathbf{name} \mid \top$$

If $x : \mathbf{chan}_{\mathsf{K}} T$ then x is the name of a channel carrying T; moreover, in an output process $\mathsf{C} : \overline{x}^{\star} v$ on x the typing rules will require $\mathsf{C} \subseteq \mathsf{K}$ – intuitively, such an output may have been causally affected only by the principals $\mathsf{k} \in \mathsf{K}$. In an input $x' p.P$ on x the continuation P must therefore be allowed to be affected by any $\mathsf{k} \in \mathsf{K}$, so any output within P must be on a channel of type $\mathbf{chan}_{\mathsf{K}'} T$ with $\mathsf{K} \subseteq \mathsf{K}'$.

We are concerned with the encapsulation of possibly badly-typed components, so must allow a box $a[P]$ in a well-typed term to contain an untyped process P. The type system cannot be sensitive to the causal flows within such a box; it can only enforce an upper bound on the set of principals that can affect any part of the contents. If $a : \mathbf{box}_{\mathsf{K}}$ then a is a box name; the contents may have been causally affected only by $\mathsf{k} \in \mathsf{K}$.

We take *type environments* Γ to be finite partial functions from names to value types. The type system has two main judgments, $\Gamma \vdash v : T$ for values and $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}$ for processes. The typing for processes records just enough information to determine when prefixing a process with an input is legitimate – if $P : \mathbf{proc}_{\mathsf{K}}$ then P can be prefixed by an input on a channel $x : \mathbf{chan}_{\mathsf{K}'} \langle\rangle$, to give $x^{\star}.P$, iff $\mathsf{K}' \subseteq \mathsf{K}$. Note, however, that a $P : \mathbf{proc}_{\mathsf{K}}$ may have been affected by (and so syntactically contain) $\mathsf{k} \notin \mathsf{K}$.

To type interactions between well-typed wrapper code and a badly-typed boxed component some simple subtyping is useful. We take the subtype order $T \leq T'$ as below, and write $\bigwedge \{ T_i \mid i \in 1..k \}$ for the greatest lower bound of $T_1, .., T_k$, where this exists.



The complete type system is given in Figure 9. It uses judgements $\vdash p : T \rhd \Delta$, meaning pattern p matches values of type T and gives bindings Δ; $\Gamma \vdash v : T$, meaning value v has type T in environment Γ; and $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}$, meaning process P is well-formed in environment Γ and can be prefixed by anything that affects at most K. We now explain the key aspects by giving some admissible typing rules.

Basic Flow Typing Consider the type environment $x : \mathbf{chan}_{\mathsf{K}} \langle\rangle$, $y : \mathbf{chan}_{\mathsf{L}} \langle\rangle$ and the reduction

$$\mathsf{C} : \overline{x}^{\star} \mid x^{\star}.\mathsf{D} : \overline{y}^{\star} \rightarrow (\mathsf{C} \cup \mathsf{D}) : \overline{y}^{\star}$$

During the reduction the output $\overline{y}^\star$ on y is causally affected by the output on x – the right-hand process term $(\mathsf{C} \cup \mathsf{D}) : \overline{y}^\star$ records that the output on y has been (indirectly) affected by all the principals that had affected the output on x. For the left process to be well-typed we must clearly require $\mathsf{C} \subseteq \mathsf{K}$ and $\mathsf{D} \subseteq \mathsf{L}$; for the right process to be well-typed we need also $\mathsf{C} \subseteq \mathsf{K}$, to guarantee this the typing rules require $\mathsf{K} \subseteq \mathsf{L}$. The relevant admissible rules are below.

$$\frac{\begin{array}{l}\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} T \\ \Gamma \vdash v : T \\ \mathsf{C} \subseteq \mathsf{K}\end{array}}{\Gamma \vdash \mathsf{C} : \overline{x}^\star v \,:\, \mathbf{proc}_{\mathsf{K}}} \qquad \frac{\begin{array}{l}\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} T \\ \Gamma, y : T \vdash P : \mathbf{proc}_{\mathsf{K}''} \\ \mathsf{K} \subseteq \mathsf{K}''\end{array}}{\Gamma \vdash x^\star y.P : \mathbf{proc}_{\mathsf{K}}}$$

Now consider also $y : \mathbf{chan}_{\mathsf{L}'} \langle\rangle$ and the process

$$\mathsf{C} : \overline{x}^\star \mid x^\star.\big(\mathsf{D} : \overline{y}^\star \mid \mathsf{D}' : \overline{y'}^\star\big)$$

Here both the output on y and that on y' must be affectable by C, so the typing rule for parallel must take the intersection of allowed-cause sets:

$$\frac{\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}} \quad \Gamma \vdash Q : \mathbf{proc}_{\mathsf{K}'}}{\Gamma \vdash P \mid Q : \mathbf{proc}_{\mathsf{K} \cap \mathsf{K}'}}$$

The examples above involve only communication within a wrapper, with tag $\star$. Communication between a wrapper and its parent, with tag $\uparrow$, has the same typing rules, as the parent is presumed well-typed.

Channel Passing Channel passing involves no additional complication. Consider the type environment $\Gamma = z : \mathbf{chan}_{\mathsf{K}''} \langle\rangle$, $x : \mathbf{chan}_{\mathsf{K}} \mathbf{chan}_{\mathsf{K}''} \langle\rangle$, and the reduction

$$\mathsf{C} : \overline{x}^\star z \mid x^\star y.\mathsf{D} : \overline{y}^\star \to (\mathsf{C} \cup \mathsf{D}) : \overline{z}^\star$$

The left-hand process is typable using the rules above if $\mathsf{C} \subseteq \mathsf{K}$ for the x output, $\mathsf{D} \subseteq \mathsf{K}''$ for the y output, and $\mathsf{K} \subseteq \mathsf{K}''$ for the input, using $\Gamma, y : \mathbf{chan}_{\mathsf{K}''} \langle\rangle \vdash \mathsf{D} : \overline{y}^\star : \mathbf{proc}_{\mathsf{K}''}$. Together these imply $(\mathsf{C} \cup \mathsf{D}) \subseteq \mathsf{K}''$, so the right-hand process is well-typed.

Interacting with a box (at $\top$) As discussed above, the contents of a box may be badly-typed, yet a wrapper must still be able to interact with them. The simplest case is that in which a wrapper sends and receives values that it considers to be of type $\top$; we consider more general communication in the next paragraph. The typing rule for boxes requires only that the principals $\mathrm{pn}(P)$ syntactically occurring within the contents P of a box are contained in the permitted set and that P's free names are all declared in the type environment.

$$\frac{\begin{array}{l}\Gamma \vdash a : \mathbf{box}_{\mathsf{K}} \\ \mathrm{pn}(P) \subseteq \mathsf{K} \\ \mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)\end{array}}{\Gamma \vdash a[P] : \mathbf{proc}_{\mathsf{K}}}$$

Consider sending to and receiving from a box $a : \mathbf{box}_{\mathsf{K}}$.

$$\mathsf{C} : \overline{x}^a v \mid a[P] \mid z^a y.Q$$

For the output to be well-typed we must insist only that $\mathsf{C} \subseteq \mathsf{K}$; for the input to be well-typed Q must be allowed to be affected by any principal that might have affected the contents P.

$$\frac{\Gamma \vdash a : \mathbf{box}_{\mathsf{K}} \quad \Gamma \vdash x : \mathbf{name} \quad \Gamma \vdash v : \top \quad \mathsf{C} \subseteq \mathsf{K}}{\Gamma \vdash \mathsf{C} : \overline{x}^a v \,:\, \mathbf{proc}_{\mathsf{K}}} \qquad \frac{\Gamma \vdash a : \mathbf{box}_{\mathsf{K}} \quad \Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} \top \quad \Gamma, y : \top \vdash P : \mathbf{proc}_{\mathsf{K}''} \quad \mathsf{K} \subseteq \mathsf{K}' \subseteq \mathsf{K}''}{\Gamma \vdash x^a p.P : \mathbf{proc}_{\mathsf{K}'}}$$

Interacting with a box (at any transmissible S) More generally, a wrapper may receive from a box tuples containing names which are to be used *for communicating with the box* as channel names, for example

$$x^a(v\, r).\big(\mathsf{C} : \overline{r}^a \mid \ldots\big)$$

receives a value v and name r from box a and uses r to send an ack back into a. This necessarily involves some run-time typechecking, as the box may send a tuple instead of a name for r. There is a design choice here: how strong should this typechecking be? Requiring an implementation to maintain a run-time record of the types of all names would be costly, so we check only the structure of values received from boxes. We suppose the run-time representations of values allow names (bit-patterns of some fixed length) and tuples to be distinguished, and the number of items in a tuple to be determined, but no more (so e.g. $x : \mathbf{chan}_{\mathsf{K}} T$ and $y : \mathbf{box}_{\mathsf{L}}$ will both be represented as bit patterns of the same length). We introduce the supertype **name** of $\mathbf{chan}_{\mathsf{K}} T$ and $\mathbf{box}_{\mathsf{L}}$, and allow a wrapper to receive only values of the *transmissible types*

$$S ::= \top \mid \mathbf{name} \mid \langle S \,..\, S \rangle$$

To send a value to a box by $\mathsf{C} : \overline{x}^a v$ it is necessary only for x to be of type **name**.

The operational semantics expresses this run-time typechecking by means of the condition that $\{^v/_p\}P$ is well-defined in the reduction communication rule and the labelled-transition input rules – for example, $\{^{\langle z\, z\rangle}/_x\}\mathsf{C} : \overline{x}^\star$ is not well-defined, as the syntax does not allow a tuple to occur in channel-name position of an output. We would like to ensure that run-time typechecking is only required when receiving values from a box, i.e. that for communication within a wrapper or between a wrapper and its parent such a substitution is always well-defined. This is guaranteed by requiring a box input prefix to immediately test all parts of a received value that are assumed of type **name** – in typing an input $x^a p.P$ the type environment Δ derived from the pattern p must contain no tuples, and all $x : \mathbf{name}$ in Δ must be used within P as a channel or box. For example, if $a : \mathbf{box}_{\mathsf{K}}$ and $x : \mathbf{chan}_{\mathsf{K}} \langle \mathbf{name}\, \mathbf{name} \rangle$ then

$$x^a(y\, z).\big(\mathsf{K} : \overline{y}^a \mid \mathsf{K} : \overline{z}^a\big)$$

is well-typed as the pattern $(y\, z)$ completely decomposes values of type $\langle \mathbf{name}\, \mathbf{name} \rangle$ and both y and z are used as channels in $\mathsf{K} : \overline{y}^a \mid \mathsf{K} : \overline{z}^a$. On the other hand

$$x^a w.\overline{x}^\star w$$

is not, as it may receive (for example) a triple from the box, leading to a later run-time error within the wrapper. The type system is conservative in also excluding

Patterns:

$$\frac{}{\vdash \_ : T \rhd \emptyset} \qquad \frac{}{\vdash x : T \rhd x : T} \qquad \frac{\vdash p_1 : T_1 \rhd \Delta_1 \;..\; \vdash p_k : T_k \rhd \Delta_k}{\vdash (p_1 \,..\, p_k) : \langle T_1 \,..\, T_k \rangle \rhd \Delta_1, .., \Delta_k}$$

Values:

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{\Gamma \vdash v_1 : T_1 \;..\; \Gamma \vdash v_k : T_k}{\Gamma \vdash \langle v_1 \,..\, v_k \rangle : \langle T_1 \,..\, T_k \rangle} \qquad \frac{\mathrm{fn}(v) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash v : \top} \qquad \frac{T \text{ atomic}}{\Gamma, x : T \vdash x : \mathbf{name}}$$

Processes:

$$\frac{\begin{array}{l} o \in \{\star, \uparrow, \overline{\uparrow}\} \\ \Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} T \\ \Gamma \vdash v : T \\ \mathsf{C} \subseteq \mathsf{K} \end{array}}{\Gamma \vdash \mathsf{C} : \overline{x}^{o} v \,:\, \mathbf{proc}_{\mathsf{K}}} \text{ (Out-}\star,\uparrow,\overline{\uparrow}\text{)} \qquad \frac{\begin{array}{l} \iota \in \{\star, \uparrow\} \\ \Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} T \\ \vdash p : T \rhd \Delta \\ \Gamma, \Delta \vdash P : \mathbf{proc}_{\mathsf{K}} \end{array}}{\Gamma \vdash x^{\iota} p.P : \mathbf{proc}_{\mathsf{K}}} \text{ (In-}\star,\uparrow\text{)}$$

$$\frac{\begin{array}{l} o \in \{a, \overline{a}\} \\ \Gamma \vdash a : \mathbf{box}_{\mathsf{K}} \\ \Gamma \vdash x : \mathbf{name} \\ \Gamma \vdash v : \top \\ \mathsf{C} \subseteq \mathsf{K} \end{array}}{\Gamma \vdash \mathsf{C} : \overline{x}^{o} v \,:\, \mathbf{proc}_{\mathsf{K}}} \text{ (Out-}a,\overline{a}\text{)} \qquad \frac{\begin{array}{l} \Gamma \vdash a : \mathbf{box}_{\mathsf{K}'} \\ \Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} S \\ \vdash p : S \rhd \Delta \\ \Gamma, \Delta \vdash P : \mathbf{proc}_{\mathsf{K}} \\ \mathsf{K}' \subseteq \mathsf{K} \\ \Delta \text{ flat} \\ P \text{ tests all names of type } \mathbf{name} \text{ in } \Delta \\ p \text{ contains no wildcards} \end{array}}{\Gamma \vdash x^{a} p.P : \mathbf{proc}_{\mathsf{K}}} \text{ (In-}a\text{)}$$

$$\frac{\begin{array}{l} \Gamma \vdash P : \mathbf{proc}_{\mathsf{K}} \\ \Gamma \vdash Q : \mathbf{proc}_{\mathsf{K}'} \end{array}}{\Gamma \vdash P \mid Q : \mathbf{proc}_{\mathsf{K} \cap \mathsf{K}'}} \text{ (Par)} \qquad \frac{\begin{array}{l} \Gamma \vdash n : \mathbf{box}_{\mathsf{K}} \\ \mathrm{pn}(P) \subseteq \mathsf{K} \\ \mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma) \end{array}}{\Gamma \vdash n[P] : \mathbf{proc}_{\mathsf{K}}} \text{ (Box)}$$

$$\frac{}{\Gamma \vdash 0 : \mathbf{proc}_{\mathsf{K}}} \text{ (Nil)} \qquad \frac{\begin{array}{l} \Gamma, x : T \vdash P : \mathbf{proc}_{\mathsf{K}} \\ T \text{ atomic} \end{array}}{\Gamma \vdash (\boldsymbol{\nu}\, x) P : \mathbf{proc}_{\mathsf{K}}} \text{ (Res)}$$

$$\frac{\begin{array}{l} \Gamma \vdash P : \mathbf{proc}_{\mathsf{K}'} \\ \mathsf{K} \subseteq \mathsf{K}' \end{array}}{\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}} \text{ (Spec)}$$

The replicated input rules are similar to the input rules. The predicate 'P tests all names of type **name** in Δ' is defined to be true iff for all $y : \mathbf{name}$ in Δ, y occurs free in channel or box position within P.

Figure 9: Coloured Box-π Typing

$x^a(y\,z).\big(\mathsf{K}:\overline{y}^a\big)$. Say a type is *atomic* if it is of the form **name**, $\mathbf{chan}_{\mathsf{K}}T$ or $\mathbf{box}_{\mathsf{K}}$ and *flat* if it is of the form $\top$, **name**, $\mathbf{chan}_{\mathsf{K}}T$, or $\mathbf{box}_{\mathsf{K}}$. Say Γ is atomic or flat if all types in $\mathrm{ran}(\Gamma)$ are. The atomic types are those which can be dynamically extended using restriction. We consider dynamics (reductions and labelled transitions) only for processes with respect to atomic typing contexts; the definitions ensure that an extruded name can always be taken to be of an atomic type. The calculus has no basic data types, e.g. a type of integers, that are not dynamically extensible. This makes the type system a little degenerate.

Nil and Restriction The typing rules for nil and restriction are straightforward; there is also a specialisation rule allowing some permitted affectees of a process to be forgotten.

$$\frac{}{\Gamma \vdash 0 : \mathbf{proc}_{\mathsf{K}}} \qquad \frac{\Gamma, x:T \vdash P : \mathbf{proc}_{\mathsf{K}} \quad T \text{ atomic}}{\Gamma \vdash (\boldsymbol{\nu}\, x)P : \mathbf{proc}_{\mathsf{K}}} \qquad \frac{\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}'} \quad \mathsf{K} \subseteq \mathsf{K}'}{\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}}$$

5.1 Soundness

We wish to infer properties of the coloured input/output behaviour of wrappers from the soundness of the type system, and therefore need a subject reduction result which refers not only to reductions (equivalently, τ transitions) but also to input/output transitions. Define typed labelled transitions by

$$\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\ell}_{\mathsf{C}} Q \quad \text{iff} \quad \big(\Gamma \text{ atomic} \wedge \Gamma \vdash P : \mathbf{proc}_{\mathsf{K}} \wedge \mathrm{dom}(\Gamma) \vdash P \xrightarrow{\ell}_{\mathsf{C}} Q\big)$$

The subject reduction theorem for ℓ an output $\overline{x}^o v$ should state that x, o, v and Q have suitable types; the theorem for ℓ an input should state that if ℓ can be typed then Q can. The result is complicated by the fact that box-π is a calculus with new name generation, so new names can be extruded and intruded. Type environments for these names are calculated as follows. For a type environment Γ, with Γ atomic, and a value v extruded at type T define the type environment $tc(\Gamma, v, T)$ for new names in v as follows.

$$\begin{array}{lcll}
tc(\Gamma, x, T) & = & x:T & \text{if } x \notin \mathrm{dom}(\Gamma) \text{ and } T \text{ atomic} \\
tc(\Gamma, x, \top) & = & x:\mathbf{name} & \text{if } x \notin \mathrm{dom}(\Gamma) \\
tc(\Gamma, x, T) & = & \emptyset & \text{if } \Gamma \vdash x:T \\
tc(\Gamma, \langle v_1 \,..\, v_k\rangle, \top) & = & \bigwedge_{1..n} tc(\Gamma, v_i, \top) & \\
tc(\Gamma, \langle v_1 \,..\, v_k\rangle, \langle T_1 \,..\, T_k\rangle) & = & \bigwedge_{1..n} tc(\Gamma, v_i, T_i) & \\
tc(\Gamma, v, T) & \text{undefined elsewhere} & &
\end{array}$$

Here $\bigwedge_{i\in 1..k}\Gamma_i$ is the type environment that maps each x in some $\mathrm{dom}(\Gamma_i)$ to $\bigwedge\{\, T \mid \exists i \,.\, x:T \in \Gamma_i \,\}$, where all of these are defined. $\bigwedge_{i\in 1..k}\Gamma_i$ is undefined otherwise. Note that in the $\top$ case the $tc(\Gamma, v_i, \top)$ will necessarily all be well-defined and will be consistent. To see the need for $\bigwedge$, consider $\Gamma = c:\mathbf{chan}_{\mathsf{K}}\langle\mathbf{box}_{\mathsf{K}}\,\mathbf{name}\rangle$ and $P = (\boldsymbol{\nu}\, x)\overline{c}^{\star}\langle x\, x\rangle$. P has an extrusion transition with value $\langle x\, x\rangle$; the type context $tc(\Gamma, \langle x\, x\rangle, \langle\mathbf{box}_{\mathsf{K}}\,\mathbf{name}\rangle)$ should be well-defined and equal to $x:\mathbf{box}_{\mathsf{K}}$.

Further, the type system involves subtyping, so $tc(\Gamma, v, T)$ can only be used as a bound on the extruded/intruded type environments. Say $\Gamma \leq \Gamma'$ iff $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\forall x \in \mathrm{dom}(\Gamma)\,.\,\Gamma(x) \leq \Gamma'(x)$.

We can now state the subject reduction result. For output tags $\{\star, \overline{\uparrow}\}$ and $\uparrow$ the name x is guaranteed to have a channel type and v the type carried; for a and $\overline{a}$ they are only guaranteed to be a **name** and a value of type $\top$. $\{\star, \overline{\uparrow}\}$ and $\overline{a}$ are communication tags, so x cannot be extruded, whereas $\uparrow$ and a are movement tags, so x may be extruded. By convention we elide a conjunct that $tc(...)$ is defined wherever it is mentioned.

Theorem 10 (Subject Reduction) *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\overline{x}^{o} v}_{\mathsf{C}} Q$ *then*

case $o \in \{\star, \overline{\uparrow}\}$**:** *for some* K', T *we have* $\mathsf{C} \subseteq \mathsf{K}'$, $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T$, *and there exists* $\Theta \leq tc(\Gamma, v, T)$ *such that* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $o = \uparrow$**:** *for some* K', T *we have* $\mathsf{C} \subseteq \mathsf{K}'$ *and there exists* $\Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{chan}_{\mathsf{K}'} T\, T\rangle)$ *such that* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $o = a$**:** *for some* K' *we have* $\mathsf{C} \subseteq \mathsf{K}'$, $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$, *and there exists a type environment* $\Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{name}, \top\rangle)$ *such that* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $o = \overline{a}$**:** *for some* K' *we have* $\mathsf{C} \subseteq \mathsf{K}'$, $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$, $\Gamma \vdash x : \mathbf{name}$, *and there exists* $\Theta \leq tc(\Gamma, v, \top)$ *such that* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

If $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{x^{\gamma} v}_{\mathsf{C}} Q$ *then*

case $\gamma \in \{\star, \uparrow\}$**:** *for some* K', T *we have* $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T$. *If moreover* $\mathsf{C} \subseteq \mathsf{K}'$ *and* $\Theta \leq tc(\Gamma, v, T)$ *then* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $\gamma = a$**:** *for some* $\mathsf{K}' \subseteq \mathsf{K}''$, *and* S *we have* $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$, $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}''} S$, $tc(\Gamma, v, S)$ *well-defined, and* $\mathrm{ran}(tc(\Gamma, v, S)) \subseteq \{\mathbf{name}\}$. *If moreover* $\mathsf{C} \subseteq \mathsf{K}''$ *and* $\Theta \leq tc(\Gamma, v, S)$ *then* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $\gamma = \overline{a}$**:** *for some* K' *we have* $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$. *If moreover* $\mathsf{C} \subseteq \mathsf{K}'$ *and we have* $\Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{name}\, \top\rangle)$ *then* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

If $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\tau}_{\mathsf{C}} Q$ *then* $\mathsf{C} = \emptyset$ *and* $\Gamma \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

A run-time error for box-π is a process in which a potential communication fails because the associated substitution is not defined. More precisely, P contains a run-time error if it contains subterms $\overline{x}^{\overline{\gamma}} v$ and $x^{\gamma} p.P$ in parallel (and not under an input prefix) and $\{^{v}/_{p}\}P$ is not defined. In a well-typed process run-time errors can only occur within boxes (whose contents are untyped) or at communications from a box to the wrapper. Internal transitions of the wrapper and communications between the wrapper and its parent therefore do not require dynamic typechecking.

Theorem 11 (Limited Runtime Errors)
If $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}$, $P \equiv (\boldsymbol{\nu}\, x_1 .. x_n)\big(\overline{x}^{\overline{\gamma}} v \mid x^{\gamma} p.P' \mid Q\big)$, Γ *atomic,* P' *does not contain a box and* $\gamma \in \{\star, \uparrow\}$ *then* $\{^{v}/_{p}\}P$ *is well-defined. Similarly for replicated input.*

5.2 Typing the Ordered Pipeline Wrapper

Finally, we can show that instantiations of $\mathcal{F}$ are well-typed and use the subject reduction theorem to conclude that $\mathcal{F}$ has the causal flow property.

Theorem 12 ($\mathcal{F}$ typing) *If*

$$\begin{array}{rcl}\Gamma & = & \Gamma_1,\ in_1 : \mathbf{chan}_{\{\mathsf{p}\}}\top,\ out_1 : \mathbf{chan}_{\{\mathsf{p}\}}\top,\quad from : \mathbf{chan}_{\{\mathsf{p}\}}\langle\top\ \mathbf{name}\rangle, \\ & & in_2 : \mathbf{chan}_{\{\mathsf{q}\}}\top,\ out_2 : \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\top,\ to : \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle\top\ \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle\rangle\rangle\end{array}$$

and also $\mathrm{fn}(P,Q) \subseteq \mathrm{dom}(\Gamma) - \{a,b\}$
then $\Gamma \vdash \emptyset \circ \mathcal{F}(P,Q) : \mathbf{proc}_{\mathsf{p}}$.

The proof of this involves type assumptions for the new-bound names of $\mathcal{F}$ as follows.

$$\begin{array}{rl} a{:}\mathbf{box}_{\{\mathsf{p}\}} & \\ b{:}\mathbf{box}_{\{\mathsf{p},\mathsf{q}\}} & \\ \mathit{buff}{:}\mathbf{chan}_{\{\mathsf{p}\}}\langle & \mathbf{chan}_{\{\mathsf{p}\}}\langle\top\ \mathbf{chan}_{\{\mathsf{p}\}}\langle\rangle\rangle \\ & \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle\top\ \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle\rangle\rangle\rangle \\ \mathit{full}{:}\mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle & \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle\top\ \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle\rangle\rangle \\ & \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle\top\ \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}}\langle\rangle\rangle \\ & \top\rangle \end{array}$$

A straightforward induction on trace lengths using the Subject Reduction theorem then proves the desired causal flow result:

Theorem 13 *Wrapper $\mathcal{F}$ has the causal flow property.*

6 Discussion

Policy enforcement mechanisms: Wrappers impose security policies on components for which it is impractical to analyze the internal structure, e.g. where only untyped object code is available.

Several alternative approaches are possible, differing in the level of trust required, the flexibility of the security policy enforced, and their costs to component producers and users. Code signing and Java-style sandboxing have low cost but cannot enforce flexible policies – signed components may behave in arbitrary ways whereas sandboxed components should not be able to interact with each other at all. Code signing requires the user to have total trust in the component producers – not just in their intent, but also in their ability to produce bug-free components. Sandboxing requires no trust, but the lack of any interaction is often too restrictive. More delicate policies can be enforced by shipping code together with data allowing the user to type-check it in a security-sensitive type system [43, 17], or to check a proof of a security-relevant behavioural property [27]. In the long term these seem likely to be the best approaches, but they require component producers to invest effort and to conform to a common standard for types or proofs – in the short term this is prohibitive. Shifting the burden of proof to the user, by performing type inference or static analysis of downloaded code, seems impractical given only the object code, which may not have been written with security in mind and so may not conform to any reasonable type system. In contrast, wrappers have been shown to have low-cost – none to the producer and only a small run-time cost to the user [12]. They allow more flexible interaction than sandboxing, albeit coarser-grain policies than proof-carrying components or security-type-checked components.

Information flow properties: The causal flow property is related to the property, studied in many contexts, that there is no information flow from a high to a low security level (though most work addresses components, which may have the property, rather than wrappers, which may enforce it on subcomponents). The literature contains a range of definitions that aim to capture this intuition in some particular setting; the formalisations vary widely. A basic choice is whether the property is stated purely extensionally, in terms of a semantics that describes only the input/output behaviour of a system, or using a more intensional semantics. A line of work on Non-Interference, summarised in [25], takes an extensional approach, stating properties in terms of the traces of input and output events of a system. Related definitions, adapted to a programming language setting, are used in [43, 17]. In the presence of nondeterminism, however, non-interference becomes problematic – as discussed in [42], the property may only be meaningful given probabilistic scheduling, which has a high run-time cost.

We believe that the basic difficultly is that the intuitive property is an *intensional* one – the notion of one component affecting another depends on some understanding of how components interact; a precise statement requires a semantics that captures some aspects of internal execution, not just input/output behaviours. This might be denotational or operational. Intensional denotational semantics have been used in the proofs (and, in the last, statements) of non-interference properties in [17, 1, 31], which use a logical relations proof and PER-based models. [42] and [31] go on to consider probabilistic properties.

For wrappers, it is important that the end-user be able to understand the security that they provide as clearly as possible. We therefore wish to use as lightweight a semantics as possible, as this must be understood before any security property stated using it, and so adopt an annotated operational semantics (developing a satisfactory denotational semantics of box-π, dealing with name creation, boxes, and untyped components, would be a challenging research problem in its own right). In a sequential setting annotated operational semantics have been used by [46]; see also [24]. The definition of the coloured semantics for box-π seems unproblematic, but in general one might validate an annotated semantics by relating it to a lower-level execution model (as mentioned below).

Neglecting boxing and wrappers for the moment, considering simply π-processes, we believe that intensional properties stated in terms of causal flow will generally imply properties stated purely in terms of trace-sets. As a starting point, we show that our type system implies a non-interference property (similar to the permutation property of [35], but for processes rather than wrappers) in a particular case. We prove that an output on a 'low' channel can always be permuted before an input on a 'higher' channel (with respect to the lattice of sets of colours).

Proposition 14 *If* $\mathsf{L} \subsetneq \mathsf{H}$ *and* $\{h : \mathbf{chan}_{\mathsf{H}} U,\ l : \mathbf{chan}_{\mathsf{L}} V\} \vdash P : \mathbf{proc}_{\emptyset}$ *then*

$$\{h, l\} \vdash P \xrightarrow{h^\star u} \xrightarrow{\overline{l}^\star v} Q \quad \textit{implies} \quad \{h, l\} \vdash P \xrightarrow{\overline{l}^\star v} \xrightarrow{h^\star u} Q.$$

Proof Sketch One can first show that $\emptyset \circ P$ has coloured transitions with the input coloured H and the output by some C. By subject reduction $\mathsf{C} \subseteq \mathsf{L}$. Analysing the form of P with Lemmas 21,20 from [35], and using $\mathsf{L} \subsetneq \mathsf{H}$, shows that the output term in P is not prefixed by the input, so the transitions can be permuted. □

Information flow type systems: The type system differs from previous work [43,

42, 28] primarily in handling badly typed components. Necessarily, it does not provide fine-grain tracking of information flow through these components. It also handles nondeterminism, new name creation and channel passing. Precise comparisons with related type systems are difficult as the languages involved differ widely. One can, however, embed fragments of these languages into box-π (noting that this only exploits the fully-typed part of our calculus). For example, in the work of Smith and Volpano [39] an assignment to a low security variable can follow an assignment to a high variable – the program `h:=3;l:=1` is well-typed. The natural translation of this program in box-π would be

$$(\overline{h}^{\star}0 \mid \overline{l}^{\star}0) \quad | \quad h^{\star}y.(\overline{h}^{\star}3 \mid l^{\star}y.\overline{l}^{\star}1)$$

where the left subterm models an initial store assigning 0 to h and l. This would not be well-typed in the system of this paper, taking $h : \mathbf{chan}_{\{\mathsf{H},\mathsf{L}\}}\mathbf{Int}$, $l : \mathbf{chan}_{\{\mathsf{L}\}}\mathbf{Int}$ and a new base type **Int**. Here the low assignment is causally dependent on the high, even though no high information can leak. On the other hand a box-π encoding of branches would not forbid high variable guards. In recent papers, type systems for capturing information flow in the π-calculus have been proposed by Honda, Vasconcelos and Yoshida [20], and by Hennessy and Riely [18]. We leave detailed comparison of the expressiveness of these systems and the π fragment of the causal type system presented here to future work.

Causal flow is a robust and straightforward property; it can be enforced by a remarkably simple type system. But, as the example above shows, it is sometimes over-constraining. We envisage that in a large system the bulk of the code will be typeable in a secure type system, a small portion will be in clearly-identified unsafe modules that are subject only to conventional typechecking, and a small portion (any untrusted code) will be encapsulated in wrappers. Automatic type inference would be required to relieve the burden of adding security annotations to all declarations.

7 Conclusion

The issue of securely composing untrusted or partially trusted components has great practical relevance. In this paper we have studied techniques for formally proving that software wrappers – the glue between components – actually enforce user-specified information flow constraints. We have defined a coloured operational semantics for a concurrent wrapper language. By keeping track of all the principals that have affected a process in the semantics it becomes easy to formulate clear statements of information flow properties. To prove that particular wrappers are secure, we defined a causal type system and so only need show that the wrappers are well typed.

Throughout the paper we focussed on wrapper properties – the calculus, statement of security properties and type system are all designed specifically for wrappers – but we believe similar techniques are applicable to other situations in which interaction must be controlled but not completely excluded, for example in isolating a security-critical kernel of a single application, or in controlling interactions between packets in an active network. Allowing untyped code fragments in otherwise typed programs gives a way to loosen security restrictions when necessary.

To make the theoretical work of this paper tractable we made the simplifying assumption that all components are expressed in box-π. It is important to relax this assumption, looking at more realistic models. In future work it would be worth integrating the causal type system with a lower-level semantics for object code, such as the typed assembly language of [14]. As we note above, one would expect real applications

to contain some (small) non-causally-typable parts, and perhaps also to require special OS support for asynchronous interaction. The issue of type inference of security levels should be addressed, and the proper statement of properties involving dynamic changes in information flow policy is also open.

Acknowledgements We would like to thank J. Leifer, J. Palsberg and the anonymous referees for comments. The first author was supported by a Royal Society University Research Fellowship and by EPSRC grant GR/L 62290 *Calculi for Interactive Systems: Theory and Experiment*. The second author did part of this work in the Object System Group at the University of Geneva.

A Coincidence of the Labelled Transition and Reduction Semantics

This appendix contains the proof of equivalence of the labelled transition semantics and the reduction semantics. It is divided into three parts, the first giving basic properties of the labelled transition system, the second showing that any reduction can be matched by a τ-transition and the third showing the converse.

A.1 Basic Properties of the LTS

The first lemmas are all proved by induction on derivations of transitions.

Lemma 15 *If* $P \equiv Q$ *then* $\mathrm{fn}(P) = \mathrm{fn}(Q)$.

Lemma 16 *If* $A \vdash P \xrightarrow{\ell} Q$ *then*

1. $\mathrm{fn}(P) \subseteq A$
2. $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P, \ell)$
3. *if* $\ell = \overline{x}^{o} v$ *then* $\mathrm{fn}(\ell) \cap A \subseteq \mathrm{fn}(P)$
4. *if* $\ell = \overline{x}^{o} v$ *then* $\mathrm{fn}(o) \subseteq \mathrm{fn}(P)$
5. *if* $\ell = \overline{x}^{o} v$ *and* $\neg \mathrm{mv}(o)$ *then* $x \in \mathrm{fn}(P)$
6. *if* $\ell = x^{\gamma} v$ *then* $\mathrm{fn}(\gamma) \subseteq \mathrm{fn}(P)$.
7. *if* $\ell = x^{\gamma} v$ *and* $\gamma \neq \overline{n}$ *then* $x \in \mathrm{fn}(P)$.

Lemma 17 (Strengthening) *If* $A, B \vdash P \xrightarrow{\ell} P'$ *and* $B \cap \mathrm{fn}(P, \ell) = \emptyset$ *then* $A \vdash P \xrightarrow{\ell} P'$.

Lemma 18 (Injective Substitution) *If* $A \vdash P \xrightarrow{\ell} P'$, *and* $f : A \to B$ *and* $g : (\mathrm{fn}(\ell) - A) \to (\mathcal{N} - B)$ *are injective, then* $B \vdash fP \xrightarrow{(f+g)\ell} (f + g)P'$.

Lemma 19 (Weakening and Strengthening) $(A \vdash P \xrightarrow{\ell} P' \wedge x \notin A \cup \mathrm{fn}(\ell))$ *iff* $(A, x \vdash P \xrightarrow{\ell} P' \wedge x \notin \mathrm{fn}(P, \ell))$.

PROOF The right-to-left implication follows from the well-formedness of A, x and from Lemma 17. The left-to-right implication follows from the condition $\mathrm{fn}(P) \subseteq A$ in the definition of the transition rules and from Lemma 18, taking f to be the inclusion from A to A, x and g the identity on $\mathrm{fn}(\ell) - A$. □

Lemma 20 (Shifting) *1.* $(A \vdash P \xrightarrow{z^{\iota} v} P' \wedge x \in \mathrm{fn}(v) - A)$ *iff* $(A, x \vdash P \xrightarrow{z^{\iota} v} P' \wedge x \in \mathrm{fn}(v) - \mathrm{fn}(P))$.

2. $(A \vdash P \xrightarrow{z^{\overline{n}} v} P' \wedge x \in \mathrm{fn}(z, v) - A)$ *iff* $(A, x \vdash P \xrightarrow{z^{\overline{n}} v} P' \wedge x \in \mathrm{fn}(z, v) - \mathrm{fn}(P))$

PROOF SKETCH Each part is by two inductions on derivations of transitions. □

As we are working up to alpha conversion a little care is required when analysing transitions. We need the following lemma (of which only the input and restriction cases are at all interesting).

Lemma 21 *1.* $A \vdash \overline{x}^{o} v \xrightarrow{\ell} Q$ *iff* $\mathrm{fn}(\overline{x}^{o} v) \subseteq A$, $\ell = \overline{x}^{o} v$ *and* $Q \equiv 0$.

2. $A \vdash x^{\iota} p.P \xrightarrow{\ell} Q$ *iff there exists* v *such that* $\mathrm{fn}(x^{\iota} p.P) \subseteq A$, $\ell = x^{\iota} v$, $\{^{v}/_{p}\}P$ *is defined and* $Q \equiv \{^{v}/_{p}\}P$.

3. $A \vdash \mathbf{!}\, x^{\iota} p.P \xrightarrow{\ell} Q$ *iff there exists* v *such that* $\mathrm{fn}(\mathbf{!}\, x^{\iota} p.P) \subseteq A$, $\ell = x^{\iota} v$, $\{^{v}/_{p}\}P$ *is defined and* $Q \equiv \mathbf{!}\, x^{\iota} p.P \mid \{^{v}/_{p}\}P$.

4. $A \vdash n[P] \xrightarrow{\ell} Q$ *iff one of the following hold.*

- *(a) there exist* x, v, *and* $\hat{P}$ *such that* $n \in A$, $\ell = \tau$, $A \vdash P \xrightarrow{\overline{x}^{\uparrow} v} \hat{P}$, *and* $Q \equiv (\boldsymbol{\nu}\, \mathrm{fn}(x, v) - A)(\overline{x}^{\overline{n}} v \mid n[\hat{P}])$.
- *(b) there exist* x *and* v *such that* $\mathrm{fn}(n[P]) \subseteq A$, $\ell = x^{\overline{n}} v$ *and* $Q \equiv n[\overline{x}^{\overline{\uparrow}} v \mid P]$.
- *(c) there exists* $\hat{P}$ *such that* $n \in A$, $\ell = \tau$, $A \vdash P \xrightarrow{\tau} \hat{P}$, *and* $Q \equiv n[\hat{P}]$.

5. $A \vdash P \mid Q \xrightarrow{\ell} R$ *iff either*

- *(a) there exists* $\hat{P}$ *such that* $\mathrm{fn}(Q) \subseteq A$, $A \vdash P \xrightarrow{\ell} \hat{P}$ *and* $R \equiv \hat{P} \mid Q$.
- *(b) there exists* x, γ, v, $\hat{P}$ *and* $\hat{Q}$ *such that* $\ell = \tau$, $A \vdash P \xrightarrow{\overline{x}^{\gamma} v} \hat{P}$, $A \vdash Q \xrightarrow{x^{\gamma} v} \hat{Q}$, *and* $R \equiv (\boldsymbol{\nu}\, \mathrm{fn}(x, v) - A)(\hat{P} \mid \hat{Q})$.

or symmetric cases.

6. $A \vdash (\boldsymbol{\nu}\, x)P \xrightarrow{\ell} Q$ *iff either*

- *(a) there exists* $\hat{x} \notin A \cup \mathrm{fn}(\ell) \cup (\mathrm{fn}(P) - x)$ *and* $\hat{Q}$ *such that* $A, \hat{x} \vdash \{^{\hat{x}}/_{x}\}P \xrightarrow{\ell} \hat{Q}$ *and* $Q \equiv (\boldsymbol{\nu}\, \hat{x})\hat{Q}$.
- *(b) there exists* y, o, v, $\hat{Q}$ *and* $\hat{x} \notin A \cup \mathrm{fn}(y, o) \cup (\mathrm{fn}(P) - x)$ *such that* $\ell = \overline{y}^{o} v$, $A, \hat{x} \vdash \{^{\hat{x}}/_{x}\}P \xrightarrow{\overline{y}^{o} v} \hat{Q}$, $\hat{x} \in \mathrm{fn}(v)$, $\neg \mathrm{mv}(o)$ *and* $Q \equiv \hat{Q}$.

(c) there exists y, o, v, $\hat{Q}$ and $\hat{x} \notin A \cup \mathrm{fn}(o) \cup (\mathrm{fn}(P) - x)$ such that $\ell = \overline{y}^o v$, $A, \hat{x} \vdash \{^{\hat{x}}/_x\}P \xrightarrow{\overline{y}^o v} \hat{Q}$, $\hat{x} \in \mathrm{fn}(y, v)$, $\mathrm{mv}(o)$ and $Q \equiv \hat{Q}$.

PROOF SKETCH The right-to-left implications are all shown using a single transition rule together with (Trans Struct Right). The left-to-right implications are shown by induction on derivations of transitions. □

A.2 Reductions Imply Transitions

Take the *size* of a derivation of a structural congruence to be number of instances of inference rules contained in it.

Lemma 22 *If $P' \equiv P$ and $\{^v/_p\}P$ is defined then $\{^v/_p\}P'$ is defined and $\{^v/_p\}P' \equiv \{^v/_p\}P$. Moreover, for any derivation of $P' \equiv P$ there is a derivation of the same size of $\{^v/_p\}P' \equiv \{^v/_p\}P$.*

PROOF SKETCH Immediate. □

Proposition 23 *If $P' \equiv P$ then $A \vdash P' \xrightarrow{\ell} Q$ iff $A \vdash P \xrightarrow{\ell} Q$.*

PROOF SKETCH A lengthy induction on the size of derivation of $P' \equiv P$. The most interesting cases are for the structural congruence axioms for scope extrusion across parallel composition and boxes. □

Lemma 24 *If $\mathrm{fn}(P) \subseteq A$ and $P \to Q$ then $A \vdash P \xrightarrow{\tau} Q$.*

PROOF SKETCH Induction on derivations of $P \to Q$, constructing derivations of τ-transitions for the reduction axioms (Red Up), (Red Down), (Red Comm) and (Red Repl), and using Proposition 23 for the (Red Struct) case. We give the first three cases in detail.

(Red Up)

$$\dfrac{\dfrac{\dfrac{}{A \vdash \overline{x}^{\uparrow} v \xrightarrow{\overline{x}^{\uparrow} v} 0}\ \text{(Trans Out)}}{A \vdash \overline{x}^{\uparrow} v \mid Q \xrightarrow{\overline{x}^{\uparrow} v} 0 \mid Q}\ \text{(Trans Par)}}{A \vdash n[\overline{x}^{\uparrow} v \mid Q] \xrightarrow{\tau} (\boldsymbol{\nu}\, \mathrm{fn}(x, v) - A)(\overline{x}^{\overline{n}} v \mid n[0 \mid Q])}\ \text{(Trans Box-1)}$$

By the premise $\mathrm{fn}(n[\overline{x}^{\uparrow} v \mid Q]) \subseteq A$ we have $\mathrm{fn}(x, v) \subseteq A$, so using (Trans Struct Right) we have $A \vdash n[\overline{x}^{\uparrow} v \mid Q] \xrightarrow{\tau} \overline{x}^{\overline{n}} v \mid n[Q]$, the right hand side of which is exactly the right hand side of (Red Up).

(Red Down)

$$\dfrac{\dfrac{}{A \vdash \overline{x}^n v \xrightarrow{\overline{x}^n v} 0}\ \text{(Trans Out)} \qquad \dfrac{x \in A}{A \vdash n[Q] \xrightarrow{x^{\overline{n}} v} n[\overline{x}^{\overline{\uparrow}} v \mid Q]}\ \text{(Trans Box-2)}}{A \vdash \overline{x}^n v \mid n[Q] \xrightarrow{\tau} (\boldsymbol{\nu}\, \mathrm{fn}(v) - A)(0 \mid n[\overline{x}^{\overline{\uparrow}} v \mid Q])}\ \text{(Trans Comm)}$$

By the premise $\mathrm{fn}(\overline{x}^n v \mid n[Q]) \subseteq A$ we have $x \in A$ and also $\mathrm{fn}(v) \subseteq A$, so using (Trans Struct Right) we have $A \vdash \overline{x}^n v \mid n[Q] \xrightarrow{\tau} n[\overline{x}^{\uparrow} v \mid Q]$, the right hand side of which is exactly the right hand side of (Red Down).

(Red Comm)

$$\dfrac{\dfrac{}{A \vdash \overline{x}^{\iota} v \xrightarrow{\overline{x}^{\iota} v} 0}\ (\text{Trans Out}) \qquad \dfrac{}{A \vdash x^{\iota} p.P \xrightarrow{x^{\iota} v} \{^{v}/_{p}\}P}\ (\text{Trans In})}{A \vdash \overline{x}^{\iota} v \mid x^{\iota} p.P \xrightarrow{\tau} (\boldsymbol{\nu}\, \mathrm{fn}(v) - A)(0 \mid \{^{v}/_{p}\}P)}\ (\text{Trans Comm})$$

The side condition $\{^{v}/_{p}\}P$ defined for (Trans In) is ensured by the same condition for (Red Comm). By the premise $\mathrm{fn}(\overline{x}^{\iota} v \mid x^{\iota} p.P) \subseteq A$ we have $\mathrm{fn}(v) \subseteq A$, so using (Trans Struct Right) we have $A \vdash \overline{x}^{\iota} v \mid x^{\iota} p.P \xrightarrow{\tau} \{^{v}/_{p}\}P$, the right hand side of which is exactly the right hand side of (Red Comm).

□

A.3 Transitions Imply Reductions

For the converse direction we first show that if a process has an output or input transition then it contains a corresponding output, input or box subterm.

Lemma 25 *If* $A \vdash P \xrightarrow{\overline{z}^{o} v} P'$ *then* $P \equiv (\boldsymbol{\nu}\, \mathrm{fn}(z, v) - A)(\overline{z}^{o} v \mid P')$

Proof Sketch Induction on derivation of $A \vdash P \xrightarrow{\overline{z}^{o} v} P'$. □

Lemma 26 *If* $A \vdash Q \xrightarrow{x^{\iota} v} Q'$ *then there exist* B, p, Q_1 *and* Q_2 *such that* $B \cap (A \cup \mathrm{fn}(x^{\iota} v)) = \{\}$ *and either* $Q \equiv (\boldsymbol{\nu}\, B)(x^{\iota} p.Q_1 \mid Q_2)$ *and* $Q' \equiv (\boldsymbol{\nu}\, B)(\{^{v}/_{p}\}Q_1 \mid Q_2)$ *or* $Q \equiv (\boldsymbol{\nu}\, B)(!\, x^{\iota} p.Q_1 \mid Q_2)$ *and* $Q' \equiv (\boldsymbol{\nu}\, B)(\{^{v}/_{p}\}Q_1 \mid !\, x^{\iota} p.Q_1 \mid Q_2)$.

Proof Sketch Induction on derivation of $A \vdash Q \xrightarrow{x^{\iota} v} Q'$. □

Lemma 27 *If* $A \vdash Q \xrightarrow{x^{\overline{n}} v} Q'$ *then there exist* B, Q_1 *and* Q_2 *such that* $B \cap (A \cup \mathrm{fn}(x^{\overline{n}} v)) = \{\}$, $Q \equiv (\boldsymbol{\nu}\, B)(n[Q_1] \mid Q_2)$ *and* $Q' \equiv (\boldsymbol{\nu}\, B)(n[(\overline{x}^{\uparrow} v \mid Q_1)] \mid Q_2)$.

Proof Sketch Induction on derivation of $A \vdash Q \xrightarrow{x^{\overline{n}} v} Q'$. □

Lemma 28 *If* $A \vdash P \xrightarrow{\tau} Q$ *then* $P \to Q$.

Proof Sketch Induction on derivations of $A \vdash P \xrightarrow{\tau} Q$, using the preceding three lemmas for the (Trans Box-1) and (Trans Comm) rules. □

The proof of Theorem 1, i.e. that if $\mathrm{fn}(P) \subseteq A$ then $A \vdash P \xrightarrow{\tau} Q$ iff $P \to Q$, is now immediate from Lemmas 24 and 28.

B Purity and Honesty

This appendix sketches the proofs of purity and honesty results. We first give another transition-analysis lemma. This allows us to rename extruded names in a label instead of in the source process term.

Lemma 29 *If $A \vdash (\boldsymbol{\nu}\, N)P \xrightarrow{\ell} Q$, $\ell = \overline{y}^{\uparrow}v$, and A, N and M are pairwise disjoint finite sets of names then there exists a partition N_1, N_2 of N, a process P', and*

$$h : (\mathrm{fn}(\ell) - A) \to (\mathcal{N} - (A, N_2, M))$$

injective such that

$$\begin{array}{l} A, N \vdash P \stackrel{(1_A + h)\ell}{\longrightarrow} P' \\ A \vdash (\boldsymbol{\nu}\, N)P \stackrel{(1_A + h)\ell}{\longrightarrow} (\boldsymbol{\nu}\, N_2)P' \equiv (1_A + h)Q \\ N_2 = N - \mathrm{fn}((1_A + h)\ell) \end{array}$$

Proof Sketch Induction on N, using Lemmas 18 and Lemma 21.6. □

The simple security properties are proved using an explicit characterisation of the states and labelled transitions of $\mathcal{W}_1(P)$. If N is a finite set of names, a is a name and $\mathcal{A}$ and Q are processes define

$$\begin{array}{lcll} [\![a; N; \mathcal{A}; Q]\!] & \stackrel{def}{=} & (\boldsymbol{\nu}\, N \cup \{a\})(& \mathcal{A} \\ & & & \mid a[Q] \\ & & & \mid\, !\, in^{\uparrow} y.\overline{in}^{a} y \\ & & & \mid\, !\, out^{a} y.\overline{out}^{\uparrow} y\ ) \end{array}$$

Say the 4-tuple a, N, $\mathcal{A}$, Q is *good* if N, $\{a\}$, and $\{in, out\}$ are pairwise disjoint, $\mathcal{A}$ is a parallel composition of outputs of the forms

$$\overline{out}^{a} v,\ \overline{out}^{\uparrow} v,\ \overline{in}^{a} v,\ \overline{x}^{a} v \text{ where } x \notin \{out, a\}$$

with $a \notin \mathrm{fn}(v)$ in each case, and Q is a process with $a \notin \mathrm{fn}(Q)$. Say a process P is good if $P \equiv [\![a; N; \mathcal{A}; Q]\!]$ for some good a, N, $\mathcal{A}$, Q.

Lemma 30 *If $a \notin \mathrm{fn}(P)$ then $\mathcal{W}_1(P) \equiv [\![a; \emptyset; 0; P]\!]$, hence $\mathcal{W}_1(P)$ is good.*

Proof Sketch Straightforward. □

We define a transition relation $A \vdash P \stackrel{\ell}{\rightharpoonup} Q$ as the least satisfying the following rules.

$$\begin{array}{lll} t_1 & & A \vdash [\![a; N; \mathcal{A}; Q]\!] \stackrel{in^{\uparrow} v}{\rightharpoonup} [\![a; N; \mathcal{A} \mid \overline{in}^{a} v; Q]\!] \quad \mathrm{fn}(v) \cap (N \cup \{a\}) = \emptyset \\ t_2 & & A \vdash [\![a; N; \mathcal{A} \mid \overline{in}^{a} v; Q]\!] \stackrel{\tau}{\rightharpoonup} [\![a; N; \mathcal{A}; Q \mid \overline{in}^{\uparrow} v]\!] \\ t_4 & A, N, a \vdash Q \stackrel{\overline{out}^{\uparrow} v}{\longrightarrow} Q' & A \vdash [\![a; N; \mathcal{A}; Q]\!] \stackrel{\tau}{\rightharpoonup} [\![a; N, \mathrm{fn}(v) - (A, N, a); \mathcal{A} \mid \overline{out}^{a} v; Q']\!] \\ t_5 & A, N, a \vdash Q \stackrel{\overline{x}^{\uparrow} v}{\longrightarrow} Q' & A \vdash [\![a; N; \mathcal{A}; Q]\!] \stackrel{\tau}{\rightharpoonup} [\![a; N, \mathrm{fn}(x, v) - (A, N, a); \mathcal{A} \mid \overline{x}^{a} v; Q']\!] \\ t_6 & & A \vdash [\![a; N; \mathcal{A} \mid \overline{out}^{a} v; Q]\!] \stackrel{\tau}{\rightharpoonup} [\![a; N; \mathcal{A} \mid \overline{out}^{\uparrow} v; Q]\!] \\ t_7 & & A \vdash [\![a; N; \mathcal{A} \mid \overline{out}^{\uparrow} v; Q]\!] \stackrel{\overline{out}^{\uparrow} v}{\rightharpoonup} [\![a; N - \mathrm{fn}(v); \mathcal{A}; Q]\!] \\ t_8 & A, N, a \vdash Q \xrightarrow{\tau} Q' & A \vdash [\![a; N; \mathcal{A}; Q]\!] \stackrel{\tau}{\rightharpoonup} [\![a; N; \mathcal{A}; Q']\!] \end{array}$$

$$\frac{A \vdash P \stackrel{\ell}{\rightharpoonup} P' \quad P' \equiv P''}{A \vdash P \stackrel{\ell}{\rightharpoonup} P''}$$

For rule t_5, we have a side condition that $x \neq out$. For all rules we have a sidecondition that the 4-tuple in the left hand side of the conclusion is good. For all rules we have a sidecondition that the free names of the process on the left hand side of the conclusion are contained in A.

Lemma 31 *If $A \vdash P \stackrel{\ell}{\rightharpoonup} P'$ then P' is good.*

PROOF By inspection of the transition axioms, checking that the 4-tuple on the right hand side is good in each case, and noting that the definition of P good is preserved by structural congruence. For t_4 by the condition $\mathrm{fn}([\![a; N; \mathcal{A}; Q]\!]) \subseteq A$ we have $\{in, out\} \subseteq A$ so $\{in, out\} \cap (\mathrm{fn}(v) - (A, N, a)) = \emptyset$. By Lemma 16.3 $a \notin \mathrm{fn}(v)$ By Lemma 16.2 $a \notin \mathrm{fn}(Q')$. For t_5 by the condition $\mathrm{fn}([\![a; N; \mathcal{A}; Q]\!]) \subseteq A$ we have $\{in, out\} \subseteq A$ so $\{in, out\} \cap (\mathrm{fn}(x, v) - (A, N, a)) = \emptyset$. By Lemma 16.3 $a \notin \mathrm{fn}(x, v)$ By Lemma 16.2 $a \notin \mathrm{fn}(Q')$. For t_8 by Lemma 16.2 $a \notin \mathrm{fn}(Q')$. The other cases are straightforward. □

Lemma 32 *For all good P we have $A \vdash P \stackrel{\ell}{\longrightarrow} P'$ iff $A \vdash P \stackrel{\ell}{\rightharpoonup} P'$.*

PROOF SKETCH We first show that $A \vdash P \stackrel{\ell}{\rightharpoonup} P'$ implies $A \vdash P \stackrel{\ell}{\longrightarrow} P'$, by induction on derivations of the former. The converse direction is by a case analysis of the possible transition derivations. □

PROOF SKETCH (of Proposition 2 (Purity)) We show by induction on k that Q is good and that the conclusion holds. The $k = 0$ case is by Lemma 30. The inductive step uses Lemmas 31 and 32. □

PROOF SKETCH (of Proposition 5 (Honesty)) To check that the unary wrapper $\mathcal{W}_1$ is honest, if N is a finite set of names, a is a name and $\mathcal{A}$ and Q are processes define

$$\begin{array}{rcl}
\langle\!\langle\!\langle a; N; \mathcal{A}; Q \rangle\!\rangle\!\rangle & \stackrel{def}{=} & Q \\
& & \mid \{\!| \ \overline{out}^{\uparrow} v \mid \overline{out}^{\overline{a}} v \in \mathcal{A} \ |\!\} \\
& & \mid \{\!| \ \overline{out}^{\uparrow} v \mid \overline{out}^{\uparrow} v \in \mathcal{A} \ |\!\} \\
& & \mid \{\!| \ \overline{x}^{\uparrow} v \mid \overline{x}^{\overline{a}} v \in \mathcal{A} \wedge x \neq out \ |\!\} \\
& & \mid \{\!| \ \overline{in}^{\uparrow} v \mid \overline{in}^{a} v \in \mathcal{A} \ |\!\} \\
\langle\!\langle a; N; \mathcal{A}; Q \rangle\!\rangle & \stackrel{def}{=} & (\boldsymbol{\nu}\, N) \langle\!\langle\!\langle a; N; \mathcal{A}; Q \rangle\!\rangle\!\rangle
\end{array}$$

Note that if $a; N; \mathcal{A}; Q$ is good then $a \notin \mathrm{fn}(\langle\!\langle a; N; \mathcal{A}; Q \rangle\!\rangle)$. Now take the family of relations below.

$$R_A \quad = \quad \equiv \circ \{\, [\![a; N; \mathcal{A}; Q]\!], \langle\!\langle a; N; \mathcal{A}; Q \rangle\!\rangle \mid a; N; \mathcal{A}; Q \text{ good and } \mathrm{fn}([\![a; N; \mathcal{A}; Q]\!]) \subseteq A \,\} \circ \equiv$$

We must check that for any P with $a \notin \mathrm{fn}(P)$ and $A \supseteq \mathrm{fn}(\mathcal{W}_1(P))$ we have $\mathcal{W}_1(P)\ R_A\ P$ and that R is an h-bisimulation. The former follows from Lemma 30 and the fact $\langle\!\langle a; \emptyset; 0; P \rangle\!\rangle \equiv P$. For the latter there are a number of cases to check; we omit the details. □

C Causality Typing: Soundness and Application

This appendix gives the soundness proofs for the type system (of the Subject Reduction and Limited Runtime Error theorems) and the proof that $\mathcal{F}$ has the causal flow property.

C.1 Soundness

The proof of Subject Reduction is divided into three main parts. First we require lemmas giving conditions under which a substitution is well-defined and well-typed (here 'good'). We then prove substitution lemmas for values and processes by induction on typing derivations, and finally the Subject Reduction result by induction on pairs of transition and typing derivations. The Limited Runtime Error result is almost an immediate consequence of these lemmas.

Say $\Gamma; \Delta \vdash \{^u/_p\}$ good iff $\{^u/_p\}$ is well-defined, $\mathrm{dom}(\{^u/_p\}) = \mathrm{dom}(\Delta)$, and $\forall x\!:\!T \in \Delta \;.\; \Gamma \vdash \{^u/_p\}x\!:\!T$. We adopt the convention below that wherever $tc(\Gamma, v, T)$ is mentioned it is also assumed well-defined.

Lemma 33 *If* $\Gamma = \bigwedge_{i\in 1..n} \Gamma_i$ *and for some* $j \in 1..n$ $\Gamma_j \vdash v\!:\!T$ *then* $\Gamma \vdash v\!:\!T$.

Proof Sketch Induction on derivation of $\Gamma_j \vdash v\!:\!T$, using the fact that atomic types are down-closed in the **(Name)** case. □

Lemma 34 (tc) *If* $\Theta \stackrel{def}{=} tc(\Gamma, v, T)$ *then* Θ atomic *and* $\Gamma, \Theta \vdash v\!:\!T$.

Proof Sketch The first part is by induction on v, noting that the set of atomic types is closed under defined glbs. The second part is also by induction on v. □

Lemma 35 *If* Γ atomic *then* $tc(\Gamma, v, \top)$ *is well-defined and is equal to the type context mapping each* $x \in \mathrm{fn}(v) - \mathrm{dom}(\Gamma)$ *to* **name**.

Proof Sketch Induction on v. □

Lemma 36 (Goodness - Standard - Preliminary) *If*

$$\begin{array}{l} \Gamma \text{ atomic} \\ \Gamma \vdash u\!:\!U \\ \vdash p\!:\!U \rhd \Delta \end{array}$$

then $\Gamma; \Delta \vdash \{^u/_p\}$ good.

Proof Sketch By induction on the two typing derivations, with case analysis on the last rule of the pattern judgement. □
Note that this result requires that the range of Γ contains no tuple types. Consider $u = x$, $U = \langle \mathbf{box}_\mathsf{K}\, \mathbf{box}_\mathsf{K} \rangle$, $\Gamma = x\!:\!U$ and $p = (y\ z)$. We have $\vdash p\!:\!U \rhd y\!:\!\mathbf{box}_\mathsf{K}, z\!:\!\mathbf{box}_\mathsf{K}$ but $\{^x/_{(y\ z)}\}$ is not well-defined.

Lemma 37 (Goodness - Dynamic) *If*

$$\begin{array}{l}\Gamma \text{ atomic}\\ \vdash p : S \rhd \Delta\\ \mathrm{dom}(\Gamma) \textit{ and } \mathrm{dom}(\Delta) \textit{ disjoint}\\ \{^u/_p\} \textit{ well defined}\\ \Delta \text{ flat } \textit{(so } \mathrm{ran}(\Delta) \subseteq \{\top, \mathbf{name}\}\textit{)}\\ \forall y : \mathbf{name} \in \Delta \,.\, \{^u/_p\}y \textit{ is a name}\\ p \textit{ contains no wildcards}\end{array}$$

then $\Theta \stackrel{def}{=} tc(\Gamma, u, S)$ *is well-defined,* $\mathrm{ran}(\Theta) \subseteq \{\mathbf{name}\}$*, and* $\Gamma, \Theta; \Delta \vdash \{^u/_p\}$ good.

Proof Sketch By induction on the pattern p, using Lemmas 35 and 33 in the variable and tuple cases. □

Lemma 38 (Substitution - values) *If*

$$\begin{array}{l}\Gamma, \Delta \vdash v : T\\ \Gamma, \Theta; \Delta \vdash \{^u/_p\} \text{ good}\end{array}$$

then $\{^u/_p\}v$ *is well-defined and* $\Gamma, \Theta \vdash \{^u/_p\}v : T$

Proof Sketch By $\Gamma, \Theta; \Delta \vdash \{^u/_p\}$ good we have that $\{^u/_p\}$ is well-defined, so $\{^u/_p\}v$ is well-defined. The second part is proved by induction on the value typing derivation. □

Note that for this to hold the typing rules must ensure that names of tuple types do not have type **name**. Note also that this lemma does not require any atomicity, and that that is important in the first input clause of the process substitution lemma.

Lemma 39 (Substitution - processes) *If*

$$\begin{array}{l}\Gamma, \Delta \vdash P : \mathbf{proc}_{\mathsf{K}}\\ \Gamma, \Theta; \Delta \vdash \{^u/_p\} \text{ good}\end{array}$$

then (1) if $\{^u/_p\}P$ *is well-defined then* $\Gamma, \Theta \vdash \{^u/_p\}P : \mathbf{proc}_{\mathsf{K}}$ *and (2) if* P *contains no subterm* $n[Q]$ *then* $\{^u/_p\}P$ *is well-defined.*

Proof Sketch We prove both parts simultaneously by induction on the size of type derivation for P. For (1) we give two instances of each typing rule; in each case showing that the premises of the right-hand instance follow from those of the left-hand instance. This uses Lemma 38. □

To see the need for the condition that P is box-free, consider $\Gamma = \emptyset$, $\Delta = x : \langle \mathbf{name}\, \mathbf{name} \rangle$, $\Theta = z : \mathbf{name}$, $P = (\boldsymbol{\nu}\, n) n[\overline{x}^{\star}\langle\rangle]$, and $\{^u/_p\} = \{^{\langle z\, z\rangle}/_x\}$. The premises of the Lemma hold, but $\{^u/_p\}P$ is not well-defined.

Lemma 40 (Painting – Jackson Pollack style) *If*

$$\begin{array}{l}\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}\\ \mathsf{C} \subseteq \mathsf{K}\end{array}$$

then $\Gamma \vdash \mathsf{C} \circ P : \mathbf{proc}_{\mathsf{K}}$

PROOF SKETCH Routine induction on typing derivations. □

Lemma 41 *If* $\Gamma \vdash v : T$ *then* $tc(\Gamma, v, T) = \emptyset$.

PROOF SKETCH Routine induction on v. □

Lemma 42 *If* $\mathrm{dom}(\Delta)$ *is disjoint from* $\mathrm{dom}(\Gamma)$, $\mathrm{fn}(v)$ *and* $\mathrm{fn}(P)$ *then*

1. $\Gamma \vdash v : T \iff \Gamma, \Delta \vdash v : T$.
2. $tc(\Gamma, v, T) = tc((\Gamma, \Delta), v, T)$.
3. $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}} \iff \Gamma, \Delta \vdash P : \mathbf{proc}_{\mathsf{K}}$

PROOF SKETCH Routine inductions. □

Lemma 43 *If* $tc((\Gamma, y : U), v, T)$ *well-defined and* $y \in \mathrm{fn}(v)$ *then there exists some* V *with* $U \leq V$ *and* $tc(\Gamma, v, T) = tc((\Gamma, y : U), v, T), y : V$.

PROOF SKETCH Induction on v, using Lemma 42.2. □

Lemma 44 *If* $\Gamma, y : U$ atomic, $\Theta \leq tc((\Gamma, y : U), v, T)$ *and* $y \in \mathrm{fn}(v)$ *then* $\Theta, y : U \leq tc(\Gamma, v, T)$.

PROOF An immediate corollary of Lemma 43, which gives that there exists some V with $U \leq V$ and $tc(\Gamma, v, T) = tc((\Gamma, y : U), v, T), y : V$. □

Lemma 45 (Structural Congruence) *If* $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

PROOF SKETCH Induction on derivations of $P \equiv Q$. □

Say $\Gamma \leq \Gamma'$ iff $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\forall x \in \mathrm{dom}(\Gamma)\ .\ \Gamma(x) \leq \Gamma'(x)$.

Lemma 46 *If* $\Gamma \leq \Gamma'$ *and* $\Gamma' \vdash v : T$ *then* $\Gamma \vdash v : T$.

PROOF SKETCH Induction on typing derivation of v. □

Lemma 47 *If* $\Gamma \leq \Gamma'$ *and* $\Gamma'; \Delta \vdash \{^u/_p\}$ good *then* $\Gamma; \Delta \vdash \{^u/_p\}$ good.

PROOF By the definition of good and Lemma 46. □

Lemma 48 *If* $A \vdash P \xrightarrow{\overline{x}^o v}_{\mathsf{C}} Q$ *then* $\mathsf{C} \subseteq \mathrm{pn}(P)$ *and* $\mathrm{pn}(Q) \subseteq \mathrm{pn}(P)$.

PROOF SKETCH Routine induction on transition derivations. □

We can now restate and prove Theorem 10.

Theorem 49 (subject reduction)

1. *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\overline{x}^{o} v}_{\mathsf{C}} Q$ *and* $o \in \{\star, \overline{\uparrow}\}$ *then for some* K', T

$$\begin{array}{l}\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T \\ \textit{there exists } \Theta \leq tc(\Gamma, v, T) \textit{ such that } \Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}} \\ \mathsf{C} \subseteq \mathsf{K}'\end{array}$$

2. *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\overline{x}^{\uparrow} v}_{\mathsf{C}} Q$ *then for some* K', T

$$\begin{array}{l}\textit{there exists } \Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{chan}_{\mathsf{K}'} T\, T\rangle) \textit{ such that } \Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}} \\ \mathsf{C} \subseteq \mathsf{K}'\end{array}$$

3. *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\overline{x}^{a} v}_{\mathsf{C}} Q$ *then for some* K'

$$\begin{array}{l}\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'} \\ \textit{there exists } \Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{name}, \top\rangle) \textit{ such that } \Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}} \\ \mathsf{C} \subseteq \mathsf{K}'\end{array}$$

4. *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\overline{x^{a}}\, v}_{\mathsf{C}} Q$ *then for some* K'

$$\begin{array}{l}\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'} \\ \Gamma \vdash x : \mathbf{name} \\ \textit{there exists } \Theta \leq tc(\Gamma, v, \top) \textit{ such that } \Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}} \\ \mathsf{C} \subseteq \mathsf{K}'\end{array}$$

5. *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{x^{\gamma} v}_{\mathsf{C}} Q$ *and* $\gamma \in \{\star, \uparrow\}$ *then for some* K', T $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T$. *If moreover*

$$\begin{array}{l}\Theta \leq tc(\Gamma, v, T) \\ \mathsf{C} \subseteq \mathsf{K}'\end{array}$$

then $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

6. *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{x^{a} v}_{\mathsf{C}} Q$ *then for some* $\mathsf{K}' \subseteq \mathsf{K}''$, *and* S *we have* $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$, $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}''} S$, $tc(\Gamma, v, S)$ *well-defined, and* $\mathrm{ran}(tc(\Gamma, v, S)) \subseteq \{\mathbf{name}\}$. *If moreover*

$$\begin{array}{l}\Theta \leq tc(\Gamma, v, S) \\ \mathsf{C} \subseteq \mathsf{K}''\end{array}$$

then $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

7. *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{x^{\overline{a}} v}_{\mathsf{C}} Q$ *then for some* K' *we have* $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$. *If moreover*

$$\begin{array}{l}\Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{name}\, \top\rangle) \\ \mathsf{C} \subseteq \mathsf{K}'\end{array}$$

then $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

8. *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\tau}_{\mathsf{C}} Q$ *then* $\mathsf{C} = \emptyset$ *and* $\Gamma \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

PROOF We give first the output part, for clauses 1–4, then the input part, for 5–7, then the tau part, for 8. Each is by induction on pairs of transition and typing derivations.

Output Consider the last pair of rules used:

(Out),(Out-$\star,\uparrow,\overline{\uparrow}$) Take $\mathsf{K}' = \mathsf{K}$ and consider cases of o:

1: $\star,\overline{\uparrow}$ By Lemma 41 we can take $\Theta = tc(\Gamma, v, T) = \emptyset$.

2: $\uparrow$ By Lemma 41 we can take $\Theta = tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{chan}_{\mathsf{K}'}\, T\, T\rangle) = \emptyset$.

(Out),(Out-$a,\overline{a}$) Take $\mathsf{K}' = \mathsf{K}$ and consider cases of o:

3: a By Lemma 41 we can take $\Theta = tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{name}, \top\rangle) = \emptyset$.

4: $\overline{a}$ By Lemma 41 we can take $\Theta = tc(\Gamma, v, \top) = \emptyset$.

(Struct),(*) 1–4 follow from the same clauses of the induction hypothesis and Lemma 45.

(*),(Spec) 1–4 follow from the same clauses of the induction hypothesis and a use of (Spec) for Q.

(Par),(Par) Consider $\Gamma \vdash_{\mathsf{K}} P \mid P' \xrightarrow{\ell}_{\mathsf{C}} Q \mid P'$ with $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\ell}_{\mathsf{C}} Q$. 1–4 follow from the same clauses of the induction hypothesis and a use of Lemma 42.3 for P'.

(Res-1),(Res) Consider $\Gamma \vdash_{\mathsf{K}} (\boldsymbol{\nu}\, y)P \xrightarrow{\ell}_{\mathsf{C}} (\boldsymbol{\nu}\, y)Q$ with $\Gamma, y : U \vdash_{\mathsf{K}} P \xrightarrow{\ell}_{\mathsf{C}} Q$, ℓ an output $\overline{x}^{o}v$ and $y \notin \mathrm{fn}(\ell)$. Suppose $o \in \{\star, \overline{\uparrow}\}$. By clause 1 of the induction hypothesis for some K', T

$\Gamma, y : U \vdash x : \mathbf{chan}_{\mathsf{K}'} T$
there exists $\Theta \leq tc((\Gamma, y : U), v, T)$ such that $\Gamma, y : U, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$
$\mathsf{C} \subseteq \mathsf{K}'$

By Lemma 42.1 $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T$. By Lemma 42.2 $tc((\Gamma, y : U), v, T) = tc(\Gamma, v, T)$, so taking the same Θ and using the (Res) typing rule we have $\Gamma, \Theta \vdash (\boldsymbol{\nu}\, y)Q : \mathbf{proc}_{\mathsf{K}}$ as required. The other cases of o are similar.

(Res-2),(Res) Consider $\Gamma \vdash_{\mathsf{K}} (\boldsymbol{\nu}\, y)P \xrightarrow{\ell}_{\mathsf{C}} Q$ with $\Gamma, y : U \vdash_{\mathsf{K}} P \xrightarrow{\ell}_{\mathsf{C}} Q$, ℓ an output $\overline{x}^{o}v$ and $y \in \mathrm{fn}(\ell)$.

Case $\neg\mathrm{mv}(o)$**.** We have $o \in \{\star, \overline{\uparrow}, \overline{a}\}$ and $y \in \mathrm{fn}(v) - \mathrm{fn}(x, o)$.
Suppose $o \in \{\star, \overline{\uparrow}\}$. By clause 1 of the induction hypothesis for some K', T

$\Gamma, y : U \vdash x : \mathbf{chan}_{\mathsf{K}'} T$
there exists $\Theta \leq tc((\Gamma, y : U), v, T)$ such that $\Gamma, y : U, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$
$\mathsf{C} \subseteq \mathsf{K}'$

By Lemma 42.1 $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T$. By Lemma 44 $\Theta, y : U \leq tc(\Gamma, v, T)$. The case $o = \overline{a}$, for clause 4, is similar.

Case $\mathrm{mv}(o)$**.** We have $o \in \{\uparrow, a\}$ and $y \in \mathrm{fn}(x, v) - \mathrm{fn}(o)$.
Suppose $o = \uparrow$. By clause 2 of the induction hypothesis for some K', T

there exists $\Theta \leq tc((\Gamma, y : U), \langle x\, v\rangle, \langle \mathbf{chan}_{\mathsf{K}'} T\, T\rangle)$ such that $\Gamma, y : U, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$
$\mathsf{C} \subseteq \mathsf{K}'$

By Lemma 44 $\Theta, y : U \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{chan}_{\mathsf{K}'} T\, T\rangle)$. The case $o = a$, for clause 3, is similar.

Input Consider the last pair of rules used:

(In)(In-$\star,\uparrow$) Clause 5. Take $\mathsf{K}' = \mathsf{K}$. By $tc(\Gamma, v, T)$ defined and Lemma 34 we have $tc(\Gamma, v, T)$ atomic and $\Gamma, tc(\Gamma, v, T) \vdash v : T$. It follows that Θ atomic and by Lemma 46 $\Gamma, \Theta \vdash v : T$. By Lemma 36 $\Gamma, \Theta; \Delta \vdash \{^v/_p\}$ good. By the definition of labelled transitions $\{^v/_p\}P$ is well-defined so by Lemma 39 $\Gamma, \Theta \vdash \{^v/_p\}P : \mathbf{proc}_{\mathsf{K}}$. By Lemma 40 $\Gamma, \Theta \vdash \mathsf{C} \circ \{^v/_p\}P : \mathbf{proc}_{\mathsf{K}}$.

(In)(In-a) Clause 6. Take $\mathsf{K}'' = \mathsf{K}$. By the definition of labelled transitions $\{^v/_p\}P$ is well-defined so $\{^v/_p\}$ is well-defined. As P tests all $y : \mathbf{name} \in \Delta$ and $\{^v/_p\}P$ is defined $\forall y : \mathbf{name} \in \Delta \,.\, \{^v/_p\}y$ is a name. By Lemma 37 $tc(\Gamma, v, S)$ is well-defined, $\mathrm{ran}(tc(\Gamma, v, S)) \subseteq \{\mathbf{name}\}$, and $\Gamma, tc(\Gamma, v, S); \Delta \vdash \{^v/_p\}$ good. By Lemma 47 $\Gamma, \Theta; \Delta \vdash \{^v/_p\}$ good. By Lemma 39 $\Gamma, \Theta \vdash \{^v/_p\}P : \mathbf{proc}_{\mathsf{K}}$. By Lemma 40 $\Gamma, \Theta \vdash \mathsf{C} \circ \{^v/_p\}P : \mathbf{proc}_{\mathsf{K}}$.

(Repl)(Repl-$\star,\uparrow$) and (Repl)(Repl-a) Similar to the two cases above.

(Box-2)(Box) Clause 7. Take $\mathsf{K}' = \mathsf{K}$. To check $\Gamma, \Theta \vdash n[\mathsf{C} : \overline{x}^{\uparrow} v \mid P] : \mathbf{proc}_{\mathsf{K}}$ observe that $\Gamma, \Theta \vdash n : \mathbf{box}_{\mathsf{K}}$ by weakening, $\mathrm{pn}(\mathsf{C} : \overline{x}^{\uparrow} v \mid P) \subseteq \mathsf{C} \cup \mathrm{pn}(P) \subseteq \mathsf{K}$, and $\mathrm{fn}(\mathsf{C} : \overline{x}^{\uparrow} v \mid P) \subseteq \mathrm{fn}(x, v) \cup \mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma, \Theta)$.

(Struct),(*) 5–7 follow from the same clauses of the induction hypothesis and Lemma 45.

(*),(Spec) 5–7 follow from the same clauses of the induction hypothesis and a use of (Spec) for Q.

(Par),(Par) Consider $\Gamma \vdash_{\mathsf{K}} P \mid P' \xrightarrow{\ell}_{\mathsf{C}} Q \mid P'$ with $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\ell}_{\mathsf{C}} Q$. 5–7 follow from the same clauses of the induction hypothesis and a use of Lemma 42.3 for P'.

(Res-1)(Res) Consider $\Gamma \vdash_{\mathsf{K}} (\boldsymbol{\nu}\, y)P \xrightarrow{\ell}_{\mathsf{C}} (\boldsymbol{\nu}\, y)Q$ with $\Gamma, y : U \vdash_{\mathsf{K}} P \xrightarrow{\ell}_{\mathsf{C}} Q$, ℓ an input $x^{\gamma} v$ and $y \notin \mathrm{fn}(\ell)$. Suppose $\gamma \in \{\star, \uparrow\}$. By clause 5 of the induction hypothesis for some K', T $\Gamma, y : U \vdash x : \mathbf{chan}_{\mathsf{K}'} T$ and

$$\begin{array}{l} \Theta \leq tc((\Gamma, y : U), v, T) \\ \mathsf{C} \subseteq \mathsf{K}' \end{array}$$

implies $\Gamma, y : U, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$. By $y \notin \mathrm{fn}(\ell)$ we have $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T$. Now suppose

$$\begin{array}{l} \Theta \leq tc(\Gamma, v, T) \\ \mathsf{C} \subseteq \mathsf{K}' \end{array}$$

By Lemma 42.2 $tc(\Gamma, v, T) = tc((\Gamma, y : U), v, T)$ so by the implication in the induction hypothesis $\Gamma, y : U, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$, hence $\Gamma, \Theta \vdash (\boldsymbol{\nu}\, y)Q : \mathbf{proc}_{\mathsf{K}}$. The case of $\gamma = \overline{a}$, for 7, is similar.

Now suppose $\gamma = a$. By clause 6 of the induction hypothesis for some $\mathsf{K}' \subseteq \mathsf{K}''$, and S we have $\Gamma, y : U \vdash a : \mathbf{box}_{\mathsf{K}'}$, $\Gamma, y : U \vdash x : \mathbf{chan}_{\mathsf{K}''} S$, $tc((\Gamma, y : U), v, S)$ well-defined, and $\mathrm{ran}(tc((\Gamma, y : U), v, S)) \subseteq \{\mathbf{name}\}$. Moreover

$$\begin{array}{l} \Theta \leq tc((\Gamma, y : U), v, S) \\ \mathsf{C} \subseteq \mathsf{K}'' \end{array}$$

implies $\Gamma, y : U, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

As $y \notin \mathrm{fn}(\ell)$ the various strengthening results suffice to show clause 6.

Tau Consider the last pair of rules used:

(Comm)(Par) We have

$$\frac{\Gamma \vdash_{\mathsf{K}_1} P_1 \xrightarrow{\overline{x}^{\gamma} v}_{\mathsf{C}} P_1' \quad \Gamma \vdash_{\mathsf{K}_2} P_2 \xrightarrow{x^{\gamma} v}_{\mathsf{C}} P_2'}{\Gamma \vdash_{\mathsf{K}_1 \cap \mathsf{K}_2} P_1 \mid P_2 \xrightarrow{\tau}_{\emptyset} (\boldsymbol{\nu}\, \mathrm{fn}(x,v) - \mathrm{dom}(\Gamma))(P_1' \mid P_2')} \text{ (Comm)}$$

Consider cases of γ and the corresponding output and input clauses:

Case $\star, \uparrow$ 1,5. By the induction hypotheses there exists $\Theta \leq tc(\Gamma, v, T)$ such that $\Gamma, \Theta \vdash P_1' : \mathbf{proc}_{\mathsf{K}_1}$ and $\Gamma, \Theta \vdash P_2' : \mathbf{proc}_{\mathsf{K}_2}$. By the (Par) and (Res) typing rules $\Gamma \vdash (\boldsymbol{\nu}\, \mathrm{fn}(x,v) - \mathrm{dom}(\Gamma))(P_1' \mid P_2') : \mathbf{proc}_{\mathsf{K}_1 \cap \mathsf{K}_2}$.

Case a 4,6 By clause 4 of the induction hypothesis there exists $\Theta \leq tc(\Gamma, v, \top)$ such that $\Gamma, \Theta \vdash P_1' : \mathbf{proc}_{\mathsf{K}_1}$. By Lemma 35 $tc(\Gamma, v, \top)$ is the type context mapping each $x \in \mathrm{fn}(v) - \mathrm{dom}(\Gamma)$ to **name**.

By clause 4 $\mathsf{C} \subseteq \mathsf{K}'$ and by clause 6 $\mathsf{K}' \subseteq \mathsf{K}''$ so $\mathsf{C} \subseteq \mathsf{K}''$.

By clause 6 of the induction hypothesis $tc(\Gamma, v, S)$ is well-defined and has range contained in $\{\mathbf{name}\}$, so $tc(\Gamma, v, S) = tc(\Gamma, v, \top)$, so $\Theta \leq tc(\Gamma, v, S)$, so $\Gamma, \Theta \vdash P_2' : \mathbf{proc}_{\mathsf{K}_2}$.

By the (Par) and (Res) typing rules $\Gamma \vdash (\boldsymbol{\nu}\, \mathrm{fn}(x,v) - \mathrm{dom}(\Gamma))(P_1' \mid P_2') : \mathbf{proc}_{\mathsf{K}_1 \cap \mathsf{K}_2}$.

Case $\overline{a}$ 3,7 Similar to case $\star, \uparrow$ above.

(Par)(Par) By the induction hypothesis.

(Box-1)(Box) We have

$$\frac{\mathrm{dom}(\Gamma) \vdash P \xrightarrow{\overline{x}^{\uparrow} v}_{\mathsf{C}} Q}{\mathrm{dom}(\Gamma) \vdash n[P] \xrightarrow{\tau}_{\emptyset} (\boldsymbol{\nu}\, \mathrm{fn}(x,v) - \mathrm{dom}(\Gamma))(\mathsf{C} : \overline{x}^{n} v \mid n[Q])} \text{ (Box-1)}$$

and

$$\frac{\begin{array}{l}\Gamma \vdash n : \mathbf{box}_{\mathsf{K}} \\ \mathrm{pn}(P) \subseteq \mathsf{K} \\ \mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)\end{array}}{\Gamma \vdash n[P] : \mathbf{proc}_{\mathsf{K}}} \text{ (Box)}$$

Note that we do not have $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}$, so the induction hypothesis is not applicable.

Take $\Theta = tc(\Gamma, \langle x\, v \rangle, \top)$.

By weakening $\Gamma, \Theta \vdash n : \mathbf{box}_{\mathsf{K}}$. By Lemma 48 $\mathrm{pn}(Q) \subseteq \mathsf{K}$. In addition we have $\mathrm{fn}(Q) \subseteq \mathrm{dom}(\Gamma, \Theta)$, so $\Gamma, \Theta \vdash n[Q] : \mathbf{proc}_{\mathsf{K}}$.

We have also $\Gamma, \Theta \vdash x : \mathbf{name}$, $\Gamma, \Theta \vdash v : \top$ and (again by Lemma 48) $\mathsf{C} \subseteq \mathsf{K}$, so $\Gamma, \Theta \vdash \mathsf{C} : \overline{x}^{n} v : \mathbf{proc}_{\mathsf{K}}$.

By the (Par) and (Res) typing rules $\Gamma \vdash (\boldsymbol{\nu}\, \mathrm{fn}(x,v) - \mathrm{dom}(\Gamma))(\mathsf{C} : \overline{x}^{n} v \mid n[Q]) : \mathbf{proc}_{\mathsf{K}}$.

(Box-3)(Box) As a τ transitions cannot increase the principal set or free name set of a process.

(Res-1)(Res) By the induction hypothesis.

(Struct)(*) Follows from the induction hypothesis and Lemma 45.

(*)(Spec) Follows from the induction hypothesis and a use of (Spec) for Q.

□

PROOF (of Theorem 11) By $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}$ and for some T and K, we get $x : \mathbf{chan}_{\mathsf{K}} T \in \Gamma$. Furthermore, we have $\Gamma \vdash v : T$ and since $\gamma \in \{\star, \uparrow\}$ we also have $\Gamma \vdash p : T$. By Lemma 36 and the facts that Γ atomic, $\Gamma \vdash v : T$ and $\vdash p : T \rhd \Delta$, we have $\Gamma; \Delta \vdash \{^{v}/_{p}\}$ good. By Lemma 39 and the facts that P does not contain a box, $\Gamma, \Delta \vdash P' : \mathbf{proc}'_{\mathsf{K}}$ and $\Gamma; \Delta \vdash \{^{v}/_{p}\}$ good, we have $\{^{v}/_{p}\}P$ is well-defined. □

C.2 Proving Causal Flow for $\mathcal{F}$

The proof that $\mathcal{F}$ has the causal flow property is a straightforward induction on the traces of $\mathcal{F}(P, Q)$ using the Subject Reduction theorem.

PROOF (of Theorem 13) Consider an instantiation $\mathcal{F}(P, Q)$ and coloured trace

$$A \vdash \emptyset \circ \mathcal{F}(P, Q) \xrightarrow{\ell_1}_{\mathsf{C}_1} R_1 \ldots \xrightarrow{\ell_k}_{\mathsf{C}_k} R_k,$$

such that all inputs on in_1 in $\ell_1..\ell_k$ are coloured with p and all inputs on in_2 are coloured with q.

By the definition of transitions (if $k \geq 1$) we have $\mathrm{fn}(\mathcal{F}(P, Q)) \subseteq A$.

Let Γ_0 be the type environment for in_1, in_2, out_1, out_2, *from* and *to*, as in the statement of Theorem 12.

Let Γ_1 be the type environment mapping $\mathrm{fn}(P, Q) - \mathrm{dom}(\Gamma_0)$ to **name** and $\Gamma = \Gamma_0, \Gamma_1$. Clearly Γ atomic.

By the definition of instantiation we have $\mathrm{fn}(P, Q) \subseteq \mathrm{dom}(\Gamma) - \{a, b\}$.

By Theorem 12 $\Gamma \vdash \emptyset \circ \mathcal{F}(P, Q) : \mathbf{proc}_{\mathsf{p}}$.

By $\mathcal{F}$ pure we know the ℓ_j have the form τ, ${in_i}^{\uparrow} v$, or $\overline{out_i}^{\uparrow} v$, for $i \in \{1, 2\}$.

Take $R_0 = \emptyset \circ \mathcal{F}(P, Q)$ and $\Theta_0 = \emptyset$. We now show by induction on k that for all $j \in 1..k$ $\ell_j = \overline{out_1}^{\uparrow} v \implies \mathsf{q} \notin \mathsf{C}_j$ and there exists Θ_j atomic such that $\Gamma, \Theta_j \vdash R_j : \mathbf{proc}_{\mathsf{p}}$.

Consider the transition $R_{k-1} \xrightarrow{\ell_k}_{\mathsf{C}_k} R_k$. We have Γ, Θ_{k-1} atomic, $\Gamma, \Theta_{k-1} \vdash R_{k-1} : \mathbf{proc}_{\mathsf{p}}$, and $\mathrm{dom}(\Gamma, \Theta_{k-1}) \vdash R_{k-1} \xrightarrow{\ell_k}_{\mathsf{C}_k} R_k$, so

$$\Gamma, \Theta_{k-1} \vdash_{\mathsf{p}} R_{k-1} \xrightarrow{\ell_k}_{\mathsf{C}_k} R_k$$

Consider cases of ℓ_j.

Case $\overline{out_1}^{\uparrow} v$. By Theorem 10 for some K', T we have $\mathsf{C}_k \subseteq \mathsf{K}'$ and there exists $\Theta \leq tc(\Gamma, \Theta_{k-1}, \langle out_1\ v \rangle, \langle \mathbf{chan}_{\mathsf{K}'} T\ T \rangle)$ such that $\Gamma, \Theta_{k-1}, \Theta \vdash R_k : \mathbf{proc}_{\mathsf{p}}$.

As $tc(...)$ is defined and $out_1 : \mathbf{chan}_{\mathsf{p}} \top \in \Gamma$ we have $\mathsf{K}' = \{\mathsf{p}\}$ and $T = \top$, so $\mathsf{C}_k \subseteq \{\mathsf{p}\}$, so $\mathsf{q} \notin \mathsf{C}_k$.

Take $\Theta_k = \Theta_{k-1}, \Theta$; it is clearly atomic.

Case $\overline{in_1}^{\uparrow} v$. By Theorem 10 for some K', T we have $\Gamma, \Theta_{k-1} \vdash in_1 : \mathbf{chan}_{\mathsf{K}'} T$. If moreover $\mathsf{C}_k \subseteq \mathsf{K}'$ and $\Theta \leq tc(\Gamma, \Theta_{k-1}, v, T)$ then $\Gamma, \Theta_{k-1}, \Theta \vdash R_k : \mathbf{proc}_{\mathsf{p}}$.

As $in_1 : \mathbf{chan}_{\mathsf{p}} \top \in \Gamma$ we have $\mathsf{K}' = \{\mathsf{p}\}$ and $T = \top$. By the premises $\mathsf{C}_k \subseteq \{\mathsf{p}\}$.

As $T = \top$ we have $tc(\Gamma, \Theta_{k-1}, v, T)$ defined and atomic; take Θ equal to this and $\Theta_k = \Theta_{k-1}, \Theta$.

The other cases are similar. □

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *LICS 98 (Indiana)*, pages 105–116. IEEE, Computer Society Press, July 1998.

[3] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In U. Montanari and V. Sassone, editors, *Proceedings CONCUR 96,* Pisa, Italy, volume 1119 of *LNCS*, pages 147–162, 1996.

[4] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, Aug. 6, 1998.

[5] M. Boreale and D. Sangiorgi. A fully abstract semantics for causality in the pi-calculus. In E. W. Mayr and C. Puech, editors, *Proceedings of STACS'95*, volume 900 of *Lecture Notes in Computer Science*, pages 243–254. Springer-Verlag, 1995.

[6] G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[7] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[8] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), ETAPS'98, LNCS 1378*, pages 140–155, Mar. 1998.

[9] P. Degano and C. Priami. Causality for mobile processes. In Z. Fülöp and F. Gécseg, editors, *Proceedings of ICALP '95*, volume 944 of *Lecture Notes in Computer Science*, pages 660–671. Springer-Verlag, 1995.

[10] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 137–151, Berkeley, CA, USA, Oct. 1996. USENIX.

[11] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, Aug. 1996.

[12] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, Berkeley, California, May 1999.

[13] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. Interposition as an operating system extension mechanism. Technical Report CSD-96-920, University of California, Berkeley, Apr. 9, 1997.

[14] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 250–261, New York, NY, USA, 1999. ACM Press.

[15] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[16] L. Gong. Java security architecture (JDK 1.2). Technical report, JavaSoft, July 1997. Revision 0.5.

[17] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th POPL*, Jan. 1998.

[18] M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous pi-calculus (extended abstract). In *Proceedings of the 27th ICALP, LNCS 1853*, pages 415–427, July 2000.

[19] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of ECOOP '91, LNCS 512*, pages 133–147, July 1991.

[20] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proceedings of ESOP 2000*, 2000.

[21] N. Islam, R. Anand, T. Jaeger, and J. R. Rao. A flexible security system for using Internet content. *IEEE Software*, 14(5):52–59, Sept./Oct. 1997.

[22] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In J. Vitek and C. Jensen, editors, *Secure Internet Programing: Security Issues for Mobile and Distributed Objects*. Springer Verlag, 1999.

[23] M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network pump. *IEEE Transactions on Software Engineering*, 22(5):329–338, May 1996.

[24] X. Leroy and F. Rouaix. Security properties of typed applets. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, San Diego, California, 19–21 Jan. 1998.

[25] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[26] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[27] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 61–91. SV, 1998.

[28] J. Palsberg and P. Ørbæk. Trust in the lambda-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

[29] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1999.

[30] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th POPL*, Jan. 1998.

[31] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. In *Proceedings of European Symposium on Programming*, Amsterdam, Netherlands, March 1999.

[32] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of ICALP '98, LNCS 1443*, pages 695–706, 1998.

[33] P. Sewell. Applied π – a brief tutorial. Technical Report 498, Computer Laboratory, University of Cambridge, Aug. 2000.

[34] P. Sewell and J. Vitek. Secure composition of insecure components. Technical Report 463, Computer Laboratory, University of Cambridge, Apr. 1999.

[35] P. Sewell and J. Vitek. Secure composition of insecure components. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW-12)*, Mordano, Italy, June 1999.

[36] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. Technical Report 478, Computer Laboratory, University of Cambridge, Nov. 1999.

[37] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Proceedings of CSFW 00: The 13th IEEE Computer Security Foundations Workshop.*, pages 269–284. IEEE Computer Society, July 2000.

[38] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*. Springer-Verlag, Oct. 1999.

[39] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, 19–21 Jan. 1998.

[40] J. Vitek and G. Castagna. Towards a calculus of mobile computations. In *Workshop on Internet Programming Languages, Chicago*, May 1998.

[41] J.-L. Vivas and M. Dam. From higher-order pi-calculus to pi-calculus in the presence of static operators. In D. Sangiorgi and R. de Simone, editors, *CONCUR '98: Concurrency Theory (9th International Conference, Nice, France)*, volume 1466 of *lncs*, pages 115–130. sv, Sept. 1998.

[42] D. Volpano and G. Smith. Confinement properties for programming languages. *SIGACT News*, 29(3):33–42, Sept. 1998.

[43] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[44] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proceedings of the 16th Symposium on Operating System Principles*, 1997.

[45] G. Winskel and M. Nielsen. Models for concurrency. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume IV, pages 1–148. Oxford University Press, 1995.

[46] S. Zdancewic, D. Grossman, and G. Morrisett. Principals in programming languages: A syntactic proof technique. In *International Conference on Functional Programming*, Paris, France, September 1999.