Responsive Choice in Mobile Processes*

Maxime Gamboni, António Ravara**

SQIG, Instituto de Telecomunicações and Mathematics Dept.
IST, Technical University of Lisbon

Abstract. We propose a general type notation, formal semantics and a sound, compositional, and decidable type system to characterise some liveness properties of distributed systems. In the context of mobile processes, we define two concepts, *activeness* (ability to send/receive on a channel) and *responsiveness* (ability to reliably conduct a conversation on a channel), that make the above properties precise. The type system respects the semantic definitions of the concepts, in the sense that the logical statements it outputs are, according to the semantics, correct descriptions of the analysed process. Our work is novel in two aspects. First, since mobile processes can make and communicate choices, a fundamental component of data representation (where a piece of data matches one of a set of patterns) or conversations (where the protocol may permit more than one message at each point), our types and type system use *branching* and *selection* to capture activeness and responsiveness in process constructs necessary for such usage patterns. Secondly, *conditional properties* offer *compositionality* features that permit analysing components of a system individually, and indicate, when applicable, what should be provided to the given process before the properties hold.
Keywords π-calculus, liveness properties, choice, static analysis

1 Introduction

When describing a distributed or service-oriented system using mobile processes [12, 15], it is important to provide a number of liveness guarantees, such as, from a client's point of view, "If I send a request, will it eventually be received? Will it eventually be processed, and will I eventually obtain an answer?", or, from a server's point of view, "Will I eventually receive a request? Will my clients respect my communication protocol?". The work we present herein ensures these properties statically, allowing, e.g., to guarantee reliability of actual software or distributed protocols, or to prove validity of calculus encodings. The main contribution of this work is an integration of *choice* with activeness and responsiveness, through a general type notation, formal semantics and a sound, compositional, and decidable type system. This work has three main ingredients:

* This work is partially supported by SQIG — Instituto de Telecomunicações and IST, Portugal, by Fundação para a Ciência e a Tecnologia, as well as the EU FET-GC project Sensoria (IST-2005-16004).

** CITI and Dep of Informatics, FCT, New University of Lisbon

First, *activeness* (ability to establish a connection) and *responsiveness* (ability to conduct a conversation for each connection) are liveness properties that have been studied, in more restricted forms, under the names of receptiveness [14], lock-freedom [8] or responsiveness [1]. Activeness is a generalisation of receptiveness both because communication is not required to succeed immediately but also because we may talk of output activeness, whereas receptiveness is only for inputs. Activeness of a channel end point (henceforth called *port*) is equivalent to lock-freedom of every instance of the complement port (including those in the environment). Acciai and Boreale's responsiveness is actually closer to what we call activeness than our concept of responsiveness.

Secondly, *conditional properties* are statements of the form $\Delta \triangleleft \Theta$, where Δ and Θ are logical statements on channel activeness meaning that "Δ holds *provided* Θ is made available (e.g. through parallel composition)".

Thirdly, the language of processes, as well as the language of types, support the concepts of *selection* (or "internal choice") and *branching* (or "external choice"), abstract descriptions of *choices* made and communicated by processes.

Conversations are an example where responsiveness and choice appear together. A conversation is a sequence of exchanges between a *server* and a *client*, guided by a *protocol* that describes what data type may be transmitted and in which direction, as well as choices that may be performed and by which party. The following example (in a π-calculus extended with numbers and a multiplication operator) is a *multiplication service* that receives numbers and returns their product. At every step the client *selects* to send more numbers ("*more*") or request the result ("*done*"). Input (respectively, output) responsiveness of channel *prod* in this scenario means that the server (respectively, the client) will keep *progressing* until reaching a terminal state, i.e. until t is sent over r.

$$\begin{aligned}\text{Server} = \mathbf{!}\, prod(s).\overline{p_0}\langle s, 1\rangle \quad | \quad \mathbf{!}\, p_0(s,t).\overline{s}(\boldsymbol{\nu} more, done).& \\ \big(more(s,n).\overline{p_0}\langle s, t \times n\rangle + done(r).\overline{r}\langle t\rangle\big)&\end{aligned}$$

$$\begin{aligned}\text{Client} = \overline{prod}(\boldsymbol{\nu} s).s(more, done).\overline{more}(\boldsymbol{\nu} s, 2).s(more, done).\overline{more}(\boldsymbol{\nu} s, 5).& \\ s(more, done).\overline{done}(\boldsymbol{\nu} r).r(t).\overline{print}\langle t\rangle&\end{aligned}$$

A second application is Milner's encoding of Boolean values in the π-calculus [11], which represents them as receivers on two parameter channels: True replies to queries with a signal on the first parameter ($\mathbf{!}\, b(tf).\bar{t}$) and False on the second one ($\mathbf{!}\, b(tf).\bar{f}$). A Boolean is (input) active if it is able to receive a request, and (input) responsive if it is able to reply to all requests. Those two processes are instances of *selection* because they pick one behaviour out of a set of mutually exclusive permissions, by sending a signal to one parameter rather than to the other. A Random Boolean can be written $\mathbf{!}\, b(tf).(\boldsymbol{\nu} x)\,(\bar{x} \mid (x.\bar{t}+x.\bar{f}))$, in which the selection is performed "at run-time" by the sum ("+"). A selection made by one process may cause *branching* in another process. Branching is typically implemented with the π-calculus sum operator, as in $\overline{b}(\boldsymbol{\nu} tf).(t.P+f.Q)$, which runs P if b is True, and Q if b is False. The "$r = a$ and b" logical circuit is implemented as follows.

$$A = \mathbf{!}\, r(tf).\overline{a}(\boldsymbol{\nu} t'f').(t'.\overline{b}\langle tf\rangle + f'.\bar{f}) \tag{1}$$

Upon receiving a request on r, process A first queries a. If it returns True (t') then the process returns on b the same channels received on r. If a returns False instead (f'), the process returns False ($\bar{f}$). So, depending on a and b's behaviour, either a signal will be sent on t, or one will be sent on f (but never both). We shall use this process as a running example in the course of this paper. First by formally stating the property "r is responsive provided that both a and b are active and responsive" into a type, then we will prove that this statement is correct using semantic definitions, and finally, to illustrate our type system, we will show how to automatically infer that property from the process alone (and given that a, b and r are all Booleans).

To the best of our knowledge, no existing work is able to perform a static analysis of processes such as (1). The usual approach for deciding whether names are active is to assign a single numerical level to name occurrences. But this does not allow for conditional properties, and moreover does not deal nicely with choice (specifically, with selection). In this case, when analysing r's continuation, as $\bar{t}$ may never get triggered (in case r returns False), it would require an infinite level, and similarly for $\bar{f}$. In other words, all a level-based system is able to say is "neither $\bar{t}$ nor $\bar{f}$ is guaranteed to ever be fired". We need a typing system able to capture the fact that *exactly one* of $\bar{t}$ and $\bar{f}$ will eventually get triggered when r is queried. In contrast to level-based analysis, dependency-based systems as we have been developing naturally incorporate choice and branching operators, to express that sort of properties (a short abstract presents the approach [5]).

These three ingredients, responsiveness, choice and conditional properties, are put together into *behavioural statements*. Given a process and for every channel a *channel type* specifying its communication protocol, the type system constructs a *process type* containing a behavioural statement describing every property it was able to infer from the process (unless the process risks violating constraints such as linearity or arity of a channel, in which case it is rejected).

This extended abstract is intended as only an overview of our work, and some technical details have been deliberately left out or put in appendices. A complete technical report including proofs can be found on-line [6].

Section 2 describes our type syntax and algebra, Section 3 gives precise semantics for our types and finally Section 4 presents our type system.

2 Processes, Types and Dependencies

After a word on the process calculus used, we describe in this section our type syntax and algebra in detail.

Processes: $P \;::=\; (P\vert P) \;\;\big\vert\;\; (\boldsymbol{\nu} x : \sigma)\, P \;\;\big\vert\;\; S \;\;\big\vert\;\; \mathbf{0}$	
Components of a parallel composition: $S \;::=\; (S{+}S) \;\;\big\vert\;\; G.P$	
Guards: $G \;::=\; T \;\;\big\vert\;\; \mathbf{!}\,T$	
Non-replicated guards: $T \;::=\; (\boldsymbol{\nu} z : \sigma)\, T \;\;\big\vert\;\; a(\tilde{y}) \;\;\big\vert\;\; \overline{a}\langle\tilde{x}\rangle$	

Table 1. Process Syntax

2.1 Processes

Our target process calculus is the synchronous polyadic π-calculus with mixed guarded sums and replication, according to the grammar given in Table 1. The symbol σ (hereafter usually omitted) stands for x's *channel type*, whose definition is given later. The letters a, b, c, d, r, x, y, z denote channel names (sometimes simply called *names*), taken from a countable set. Every channel x has two *ports*, its input (x) and output ($\bar{x}$) end points. Letter p ranges over ports.

Free names $\mathrm{fn}(P)$ of a process P are defined as usual, binders being $(\boldsymbol{\nu} x)\, P$ (binding x in P) and $a(\tilde{y}).P$ (binding $\tilde{y}$ in P). A guard G has a *subject port* $\mathsf{sub}(G)$, defined by the axioms $\mathsf{sub}(\mathbf{!}\,T) \overset{\mathrm{def}}{=} \mathsf{sub}((\boldsymbol{\nu} x : \sigma)\, T) \overset{\mathrm{def}}{=} \mathsf{sub}(T)$, $\mathsf{sub}(a(\tilde{y})) \overset{\mathrm{def}}{=} a$ and $\mathsf{sub}(\overline{a}\langle\tilde{x}\rangle) \overset{\mathrm{def}}{=} \bar{a}$, and a set of *object names* $\mathsf{obj}(G)$, defined by $\mathsf{obj}(\mathbf{!}\,T) \overset{\mathrm{def}}{=} \mathsf{obj}((\boldsymbol{\nu} x : \sigma)\, T) \overset{\mathrm{def}}{=} \mathsf{obj}(T)$, $\mathsf{obj}(a(\tilde{y})) \overset{\mathrm{def}}{=} \{\tilde{y}\}$ and $\mathsf{obj}(\overline{a}\langle\tilde{x}\rangle) \overset{\mathrm{def}}{=} \{\tilde{x}\}$, of which the *bound names* $\mathrm{bn}(G)$ are a subset: $\mathrm{bn}(a(\tilde{y})) \overset{\mathrm{def}}{=} \{\tilde{y}\}$ and $\mathrm{bn}((\boldsymbol{\nu}\tilde{z})\, \overline{a}\langle\tilde{x}\rangle) \overset{\mathrm{def}}{=} \{\tilde{z}\}$. Finally, the *multiplicity* $\#(G)$ of a guard G is ω if it is replicated, or 1 otherwise. The operational semantics of the calculus is given, as usual, by a labelled transition system (Appendix A).

2.2 Syntax of types

Types contain annotations on channels to record the liveness properties they enjoy (activeness and/or responsiveness), as well as the number of times they may be used: *Activeness* and *multiplicities* specify, respectively, lower and upper bounds on the number of times a port is going to be used. We write p^m, where p is a port and m is a multiplicity that can be 0, 1, ω (one replicated occurrence) or $\star$ (unbounded), to specify an upper bound on the use of p. We write $p_{\mathbf{A}}$ to specify a non-zero lower bound on the use of p. Note that multiplicity is a *safety* property (broken by using a channel too often), while activeness is a *liveness* property, satisfied once a message is ready to be sent or received. We focus on liveness properties, and use multiplicities merely as a tool for establishing them.

Behavioural statements. Just like $p_{\mathbf{A}}$, activeness of a port p, tells that a p-guarded process eventually comes to top-level[1], activeness of a branching $s_{\mathbf{A}}$ where $s = \sum_i p_i$ requires a sum to eventually come to top-level, with one p_i-guarded branch for each i.

[1] Q is *at top-level* in P if $P \equiv (\boldsymbol{\nu}\tilde{z})\, (P \mid Q)$

Behavioural statements Δ	$::=$	$\Delta \vee \Delta \;\mid\; \Delta \wedge \Delta \;\mid\; \Delta \lhd \Delta \;\mid\; \gamma \;\mid\; p^m \;\mid\; \bot \;\mid\; \top$
Resources γ	$::=$	$s_{\mathbf{A}} \;\mid\; p_{\mathbf{R}}$
Sums s	$::=$	$s + s \;\mid\; p$

Table 2. Behavioural Statement Syntax

A port a or $\bar{a}$ is *responsive* in a process (written $a_{\mathbf{R}}$ or $\bar{a}_{\mathbf{R}}$) if a-receivers (or $\bar{a}$-senders) respect the channel *protocol*. Protocols, expressed using *channel types*, will be described later on.

These three expressions — p^m, $s_{\mathbf{A}}$ and $p_{\mathbf{R}}$ — are the fundamental building blocks of *behavioural statements*, logical expressions describing the behaviour of a process. The *dependency statement* $\Delta \lhd \Theta$ (read "Δ if Θ" and also called *rely-guarantee* construct in the literature), says that whenever Θ holds in a process's environment, Δ will hold in that process. For instance $a_{\mathbf{A}} \lhd \bar{b}_{\mathbf{A}}$ holds for the process $b.a$ because, should a third-party process provide an output at b ("$\bar{b}_{\mathbf{A}}$"), this process will provide an input at a ("$a_{\mathbf{A}}$"). Dependency "$\Delta \lhd \Theta$" can be understood as an implication "$\Delta \Leftarrow \Theta$", and indeed shares many properties with logical implication.

The usual logical connectives $\vee$ (disjunction), $\wedge$ (conjunction), $\top$ (truth) and $\bot$ (falsity) are used to build complex behavioural statements (ranged over by ε, Δ, Θ or Ξ and given by the grammar in Table 2) about a process. In this work, multiplicities p^m may appear neither on the left nor on the right of a $\lhd$ connective, and in $\Delta \lhd \Theta$, Δ and Θ may not themselves use the $\lhd$ connective. By convention ε denotes the dependencies of a particular resource. We often group statements about a particular port into a single abbreviated expression: $p^m_{\mathbf{A}} \stackrel{\text{def}}{=} p^m \wedge p_{\mathbf{A}}$ ("p is used at least once and at most m times") and $p_{\mathbf{AR}} \stackrel{\text{def}}{=} p_{\mathbf{A}} \wedge p_{\mathbf{R}}$ ("p is active and responsive"). For instance $p^1_{\mathbf{A}}$ is a linear port (used precisely once), $p^\star_{\mathbf{A}}$ is a port used *at least* once, and p^1 is a port used *at most* once.

Channel Types give, separately for the input and output ports of a channel, behavioural statements that *must* hold for every receiver, respectively sender, at the corresponding channel, using natural numbers (starting from 1) to refer to the parameter channels. Specifically, multiplicities indicate which capabilities (input or output) of the parameters may be used, activeness resources tell which parameter must be active, selection "$\vee$" tells what choices may be performed, and branching "+" tells what branching they must offer. Note how the type of some channel a only talks about the parameters carried on a — it does not include a's multiplicities or activeness which are given by the *process type*.

The input port of a Boolean channel (such as r, a and b in (1)) has type

$$\bar{1}^1_{\mathbf{A}} \;\vee\; \bar{2}^1_{\mathbf{A}} \tag{2}$$

that says that either the first parameter ("1") must be output ("$\bar{1}$") active ("$_{\mathbf{A}}$"), and the second parameter unused, or ("$\vee$") the opposite ("$\bar{2}^1_{\mathbf{A}}$") — by convention we don't mention ports with multiplicity zero. The output port has type

$$\left(1^1 \;\vee\; 2^1\right) \wedge (1+2)_{\mathbf{A}}, \tag{3}$$

which has a similar meaning, but where one of its parameters (t and f in the example) should be *input* rather than output. Additionally ("$\wedge$"), inputs at the parameters ("1" and "2") must be the guards of a sum ("+"). A Boolean channel is now said input (resp., output) *responsive* if its input port (resp., output port) respects this protocol. A *channel type* σ is a triple $(\tilde{\sigma}; \xi_{\mathrm{I}}; \xi_{\mathrm{O}})$ where $\tilde{\sigma}$ are the types of the parameters, ξ_{I} and ξ_{O} are behavioural statements (only using *numbers* for channels) standing for the behaviour required respectively of inputs and outputs at that channel. For instance, abbreviating the parameter-less channel type $(\varnothing; \top; \top)$ as $()$, the Boolean type gathers (2) and (3) as

$$\mathsf{Bool} \stackrel{\text{def}}{=} \left(()() \;;\; \bar{1}^1_{\mathbf{A}} \;\vee\; \bar{2}^1_{\mathbf{A}} \;;\; (1^1 \vee 2^1) \wedge (1+2)_{\mathbf{A}}\right)$$

The type σ_p of channel *prod* in the conversation example from the introduction nicely illustrates how a channel type describes the protocol used at a channel:
1. Connection: $\sigma_p = (\sigma_s; \bar{1}_{\mathbf{AR}}; 1_{\mathbf{AR}})$,
2. Client selects m or d: $\sigma_s = (\sigma_m, \sigma_d; \bar{1}_{\mathbf{AR}} \vee \bar{2}_{\mathbf{AR}}; (1+2)_{\mathbf{A}} \wedge (1_{\mathbf{R}} \vee 2_{\mathbf{R}}))$,
3. If m, client sends a number: $\sigma_m = (\sigma_s, \mathsf{Int}; \bar{1}_{\mathbf{AR}}; 1_{\mathbf{AR}})$,
4. If d, client requests result: $\sigma_d = (\sigma_r; \bar{1}_{\mathbf{AR}}; 1_{\mathbf{AR}})$,
5. Server returns result: $\sigma_r = (\mathsf{Int}; \top; \top)$.

Process Types are similar to channel types, but refer to channels by names rather than parameter numbers. A process type Γ is a structure $(\Sigma \;;\; \Xi_{\mathrm{L}} \;\blacktriangleleft\; \Xi_{\mathrm{E}})$ where $\Sigma = \tilde{a} : \tilde{\sigma}$ is the *channel type mapping* giving the channel types of free names used by the process, while Ξ_{L} and Ξ_{E} are behavioural statements using names in $\tilde{a}$, respectively the *local component* (constraints what the process does) and the *environment component* (constraints what any third-party process may do). Unless specified otherwise, Ξ_{E} contains no activeness or responsiveness statements.

Typing the running example. The process (1) can be given the following type, where the local component says that r is active with multiplicity ω (i.e. has precisely one occurrence and it is replicated), and its responsiveness depends on both a and b being active and responsive. The environment component specifies that a and b must both have at most one replicated instance, and there are no additional input on r.

$$\Gamma_A = \left(a : \mathsf{Bool}, b : \mathsf{Bool}, r : \mathsf{Bool}; r^{\omega}_{\mathbf{A}} \wedge \left(r_{\mathbf{R}} \triangleleft (a_{\mathbf{AR}} \wedge b_{\mathbf{AR}})\right) \blacktriangleleft a^{\omega} \wedge b^{\omega} \wedge r^0\right) \tag{4}$$

Apart from some informal descriptions, behavioural statements have so far been purely syntactical constructs. Some operators and relations we present ahead clarify their semantics: (1) equivalence and weakening relations highlight their *logical* aspect (a statement may *imply* another); (2) composition, restriction and prefixing operators highlight their *spatial* aspect by mirroring process constructs; and (3) the transition operator and the typed transition relation highlight their *dynamical* aspect (types, like processes, may evolve over time).

2.3 Logical Aspects

We define weakening and reduction relations on behavioural statements.

A weakening relation on behavioural statements (and, by extension, on process types) builds on the idea that a statement A can be said *weaker* than a statement B (written $A \succeq B$) if all worlds (processes) satisfying B also satisfy A. Similarly, statements are *equivalent* (written $A \cong B$) if they hold in the same set of worlds (i.e., if $A \succeq B$ and $B \succeq A$).

The weakening relation is inductively defined by the rules in Appendix B. We present now the most significant rules, useful to analyse the running example.

- $\Delta_1 \wedge \Delta_2 \preceq \Delta_1 \preceq \Delta_1 \vee \Delta_2$, and $\bot \preceq \Delta \preceq \top$. $\Delta \wedge (\Delta_1 \vee \Delta_2) \cong (\Delta \wedge \Delta_1) \vee (\Delta \wedge \Delta_2)$.
- $\wedge$ and $\vee$ are commutative, associative and idempotent, up to $\cong$.
- On multiplicities, $p^{m_1} \preceq p^{m_2}$ if $m_1 = 0$ or $m_2 \in \{m_1, \star\}$. Also, $p^\star \cong \top$.
- $(\gamma \lhd \varepsilon_1) \wedge (\gamma \lhd \varepsilon_2) \cong \gamma \lhd (\varepsilon_1 \vee \varepsilon_2)$ and $(\gamma \lhd \varepsilon_1) \vee (\gamma \lhd \varepsilon_2) \cong \gamma \lhd (\varepsilon_1 \wedge \varepsilon_2)$.

The Technical report ("Weakening Decidability" in [6], Section 2) describes a way to decide if two behavioural statements are related by weakening. From now on we consider process types and dependencies up to $\cong$ as equal, since every operator and relation considered commutes with $\cong$ (Lemma "Types may be seen up to $\cong$" in [6], Section 2).

Dependency reduction. Another relation highlighting the logical aspect of behavioural statements is the *reduction* relation, analogous to the *modus ponens* rule in logic. It occurs with process composition which may create dependency chains that must then be reduced. For example $a.\bar{b}$ and $b.\bar{c}$ satisfy respectively $\bar{b}_{\mathbf{A}} \lhd \bar{a}_{\mathbf{A}}$ and $\bar{c}_{\mathbf{A}} \lhd \bar{b}_{\mathbf{A}}$, while their composition $a.\bar{b} \mid b.\bar{c}$ satisfies $(\bar{b}_{\mathbf{A}} \lhd \bar{a}_{\mathbf{A}}) \wedge (\bar{c}_{\mathbf{A}} \lhd \bar{b}_{\mathbf{A}}) \wedge \underline{(\bar{c}_{\mathbf{A}} \lhd \bar{a}_{\mathbf{A}})}$ (where the underlined statement was derived from the other two) or, applying type equivalence, $(\bar{b}_{\mathbf{A}} \lhd \bar{a}_{\mathbf{A}}) \wedge \big(\bar{c}_{\mathbf{A}} \lhd (\bar{a}_{\mathbf{A}} \vee \bar{b}_{\mathbf{A}})\big)$. More generally:

Definition 1 (Dependency Reduction). *The* reduction relation $\hookrightarrow$ *on behavioural statements is a partial order relation satisfying*
1. $(s_{\mathbf{A}} \lhd \varepsilon) \wedge (\gamma \lhd \varepsilon') \;\hookrightarrow\; (s_{\mathbf{A}} \lhd \varepsilon) \wedge (\gamma \lhd \varepsilon'\{^{\varepsilon\{^{\bot}/_{\gamma}\} \vee s_{\mathbf{A}}}/_{s_{\mathbf{A}}}\})$,
2. $(p_{\mathbf{R}} \lhd \varepsilon) \wedge (\gamma \lhd \varepsilon') \;\hookrightarrow\; (p_{\mathbf{R}} \lhd \varepsilon) \wedge (\gamma \lhd \varepsilon'\{^{\varepsilon\{^{\bot}/_{\gamma}\} \wedge p_{\mathbf{R}}}/_{p_{\mathbf{R}}}\})$.
A closure *of a behavioural statement Ξ, written* close(Ξ)*, is Ξ' such that $\Xi \hookrightarrow \Xi'$ and if $\Xi' \hookrightarrow \Xi''$ then $\Xi' \cong \Xi''$.*

The different treatment of activeness and responsiveness (in γ's dependencies, the former gets a $\vee$ and the latter a $\wedge$), can be understood as follows: If two processes P_1 and P_2 both provide an a-input, it is enough that one of them is able to receive a request to have a active in $P_1|P_2$. On the other hand, they must both be responsive in order to guarantee that all a-requests will get a response. Also note how *self-dependencies* $\gamma \lhd \gamma$ are replaced by $\gamma \lhd \bot$. Activeness self-dependencies are found in deadlocks such as $\bar{a}.!\,b \mid \bar{b}.!\,a$ where $a_{\mathbf{A}}$ and $b_{\mathbf{A}}$ depend on each other, and responsiveness self-dependencies are found in livelocks such as $!\,a(x).\overline{b}\langle x\rangle \mid !\,b(x).\overline{a}\langle x\rangle$ where $a_{\mathbf{R}}$ and $b_{\mathbf{R}}$ depend on each other.

Most operators commute with the logical connectives: A *logical homomorphism* is a function f on behavioural statements or process types such that $f(X \vee Y) = f(X) \vee f(Y)$ and $f(X \wedge Y) = f(X) \wedge f(Y)$. It is now sufficient to describe how operators behave on behavioural statements not using $\wedge$ or $\vee$, as the general behaviour can be derived from the above.

2.4 Spatial Aspects

Every process constructor has a corresponding operator on types, which is the essence of any syntax directed type system such as ours. We focus on the (parallel) composition operation "$\Gamma_1 \odot \Gamma_2$" that, given the types Γ_1 and Γ_2 of two processes P_1 and P_2, constructs the type of $P_1|P_2$. On behavioural statements, $\odot$ is the logical homomorphism such that:

1. $(p^m) \odot (p^{m'}) \stackrel{\text{def}}{=} p^{m+m'}$
2. $(s_{\mathbf{A}} \lhd \varepsilon) \odot (s_{\mathbf{A}} \lhd \varepsilon') \stackrel{\text{def}}{=} (s_{\mathbf{A}} \lhd \varepsilon) \vee (s_{\mathbf{A}} \lhd \varepsilon')$
3. $(p_{\mathbf{R}} \lhd \varepsilon) \odot (p_{\mathbf{R}} \lhd \varepsilon') \stackrel{\text{def}}{=} (p_{\mathbf{R}} \lhd \varepsilon) \wedge (p_{\mathbf{R}} \lhd \varepsilon')$
4. When they don't have resources in common, $\Xi \odot \Xi' \stackrel{\text{def}}{=} \Xi \wedge \Xi'$.

When composing full process types, the *local* component of the whole is the composition of the local components of the parts, and the *environment* of the whole is the environment of one part, without the local component of the other part (we omit the formal definition of "\" that does just that). Formally:

$$(\Sigma; \Xi_{\mathrm{L1}} \blacktriangleleft \Xi_{\mathrm{E1}}) \odot (\Sigma; \Xi_{\mathrm{L2}} \blacktriangleleft \Xi_{\mathrm{E2}}) \stackrel{\text{def}}{=} (\Sigma; \Xi_{\mathrm{L1}} \odot \Xi_{\mathrm{L2}} \blacktriangleleft (\Xi_{\mathrm{E1}} \setminus \Xi_{\mathrm{L2}}) \wedge (\Xi_{\mathrm{E2}} \setminus \Xi_{\mathrm{L1}}))$$

The $\odot$ operator is *associative, commutative* and has $(\varnothing; \top \blacktriangleleft \top)$ as a *neutral element* (Lemma "Composition Properties" in [6], Section 2). See Sections 2.5 and 4 for examples.

2.5 Dynamical Aspects

We describe in this section a *transition operator* "$\Gamma \wr \mu$" on types, to answer to the following question: If a process P has type Γ, and $P \xrightarrow{\mu} P'$, what is the type of P'? The motivation for such an operator is three-fold:

Ruling out transitions that a well-behaved third party process can't cause and that force a process to misbehave. E.g. interference on a linear channel (a transition $l|\bar{l} \xrightarrow{l} \bar{l}$ is ruled out, as it contradicts $\bar{l}^0$ in the environment) and channel mismatches $(a(x).\overline{x}\langle 3\rangle \mid b(yz) \xrightarrow{a(b)} \overline{b}\langle 3\rangle \mid b(yz)$ introduces an arity mismatch and is ruled out, as a's parameter type is incompatible with b's type).

Secondly, to avoid semantics with universal quantification on third-party processes, we characterise the $\lhd$ connective with labelled transitions. However, those change the properties of processes: assume P and E represent a process and its environment. A request $P \xrightarrow{\overline{a}\langle b\rangle}$ is then received as $E \xrightarrow{a(b)} E'$, and if a was responsive in E then $\overline{b}$ is active and responsive and a is no longer active in E' (for linear a with a typical input-output-alternating channel type). The transition operator predicts the evolution of both the process and its environment.

Thirdly, to prove that the previous point is sound, subject reduction works with arbitrary labelled-transitions (see Proposition 1 on page 13).

For transitions not carrying parameters, we have the following equality:

$$(\Sigma; \Xi_\mathrm{L} \blacktriangleleft \Xi_\mathrm{E}) \wr p \stackrel{\mathrm{def}}{=} (\Sigma; \Xi_\mathrm{L} \setminus p \blacktriangleleft \Xi_\mathrm{E} \setminus \bar{p})$$

Based on $\odot$ and *channel type instantiation* $\sigma[\tilde{x}]$ (which transforms a channel type σ into a process type, essentially by substituting parameter references $1 \dots n$ by $x_1 \dots x_n$, but with extra care in case two x_i are equal), input transitions are simulated as follows. Let $\Gamma = (\Sigma; \Xi_\mathrm{L} \blacktriangleleft \Xi_\mathrm{E})$ with $\Sigma(a) = \sigma$.

$$\Gamma \wr a(\tilde{x}) \stackrel{\mathrm{def}}{=} \Gamma \wr a \odot \sigma[\tilde{x}] \triangleleft (a_\mathbf{R} \blacktriangleleft \bar{a}_\mathbf{R})$$

The $\Gamma \triangleleft (a_\mathbf{R} \blacktriangleleft \bar{a}_\mathbf{R})$ operation makes Γ's local component depend on $a_\mathbf{R}$ and its environment component depend on $\bar{a}_\mathbf{R}$. An output transition can be done by swapping the local and environment components, doing an input transition, and swapping the two resulting components back. We illustrate the above operator on the transition $A \xrightarrow{r(uv)} A' = A \,|\, \overline{a}(\boldsymbol{\nu} t' f').(t'.\overline{b}\langle uv\rangle + f'.\overline{v})$ where A is (1) and its type (4) is $\Gamma_A = \big(\Sigma \,;\, r^\omega_\mathbf{A} \wedge r_\mathbf{R} \triangleleft (a_\mathbf{AR} \wedge b_\mathbf{AR}) \;\blacktriangleleft\; a^\omega \wedge b^\omega \wedge r^0\big)$:

$$\Gamma_A \wr r(uv) = \Gamma_A \wr r \odot \big(u : (), v : (); (\bar{u}_\mathbf{A} \vee \bar{v}_\mathbf{A}) \triangleleft r_\mathbf{R} \blacktriangleleft (u^1 \vee v^1) \wedge (u+v)_\mathbf{A} \triangleleft \bar{r}_\mathbf{R}\big)$$

1. The "$\wr r$" part has no effect as $r^\omega \setminus r = r^\omega$ and $\bar{r}^\star \setminus \bar{r} = \bar{r}^\star$.
2. The channel type mapping is $\Sigma' = a : \mathsf{Bool}, b : \mathsf{Bool}, r : \mathsf{Bool}, u : (), v : ()$.
3. the remote component "Ξ_E" is just the conjunction of $\big(a^\omega \wedge b^\omega \wedge r^0\big)$ from Γ_A and $\big((u^1 \vee v^1) \wedge (u+v)_\mathbf{A} \triangleleft \bar{r}_\mathbf{R}\big)$.
4. The local component is $\Xi_\mathrm{L} = \Big(r^\omega_\mathbf{A} \wedge r_\mathbf{R} \triangleleft (a_\mathbf{AR} \wedge b_\mathbf{AR})\Big) \odot \Big((\bar{u}_\mathbf{A} \vee \bar{v}_\mathbf{A}) \triangleleft r_\mathbf{R}\Big) = \Big(r^\omega_\mathbf{A} \wedge r_\mathbf{R} \triangleleft (a_\mathbf{AR} \wedge b_\mathbf{AR})\Big) \wedge \Big((\bar{u}_\mathbf{A} \vee \bar{v}_\mathbf{A}) \triangleleft r_\mathbf{R}\Big)$.
5. Closure of Ξ_L reduces the $(\bar{u}_\mathbf{A} \vee \bar{v}_\mathbf{A}) \triangleleft r_\mathbf{R} \;\wedge\; r_\mathbf{R} \triangleleft (a_\mathbf{AR} \wedge b_\mathbf{AR})$ dependency chain into $(\bar{u}_\mathbf{A} \vee \bar{v}_\mathbf{A}) \triangleleft (r_\mathbf{R} \wedge a_\mathbf{AR} \wedge b_\mathbf{AR})$.
6. Finally, because of r^0 in the remote side Ξ_E, the dependency on $r_\mathbf{R}$ can be replaced[2] by $\top$ in the above statement, resulting in $(\bar{u}_\mathbf{A} \vee \bar{v}_\mathbf{A}) \triangleleft (a_\mathbf{AR} \wedge b_\mathbf{AR})$.
7. Omitting irrelevant parts, we end up with

$$\big(\Sigma'; (\bar{u}_\mathbf{A} \vee \bar{v}_\mathbf{A}) \triangleleft (a_\mathbf{AR} \wedge b_\mathbf{AR}) \blacktriangleleft a^\omega \wedge b^\omega \wedge r^0 \wedge (u^1 \vee v^1)\big) \tag{5}$$

as a type for $A \,|\, \overline{a}(\boldsymbol{\nu} t' f').(t'.\overline{b}\langle uv\rangle + f'.\overline{v})$, where the local component is read as "if active and responsive a and b inputs are provided, then an output will be sent on (exactly) one of u and v," which is indeed a correct statement for that process A'. Remember that this type was not obtained by analysing A', but is a prediction of the effect of a transition $\xrightarrow{r(uv)}$ on a process of type Γ_A.

Transitions on types and on processes are combined to form transitions on *typed processes*: $(\Gamma; P) \xrightarrow{\mu} (\Gamma \wr \mu; P')$ if $P \xrightarrow{\mu} P'$ and $\Gamma \wr \mu$ is well-defined.

[2] An unused port is vacuously responsive. Inversely, $r_\mathbf{A}$ could be replaced by $\bot$.

3 Activeness and Responsiveness

In this section we define *correctness* of a type Γ for a process P, denoted $\Gamma \models P$.

The *projection relation* "$\searrow$" permits extracting an "elementary" part of a process type for testing its validity. It simulates selections done by the environment by reducing any $\Delta_1 \wedge \Delta_2 \ldots$ to Δ_i and any $\gamma \triangleleft (\varepsilon_1 \vee \varepsilon_2 \ldots)$ to $\gamma \triangleleft \varepsilon_j$ for some i and j. Then, proving that a projection $\bigvee_i \gamma_i \triangleleft \varepsilon_i$ is correct for P is done with a *strategy* — a function f mapping typed processes to pairs of transition labels and typed processes such that $f(\Gamma; P) = (\mu; \Gamma'; P')$, also written $(\Gamma; P) \xrightarrow{f} (\Gamma'; P')$, implies $(\Gamma; P) \xrightarrow{\mu} (\Gamma'; P')$. For $(\Gamma; P) \notin \mathrm{dom}(f)$ we write $(\Gamma; P) \xrightarrow{f} (\Gamma; P)$. A valid strategy "leads to" a process where one of the γ_i is immediately available, using no more external resources than declared in ε_i. While projections deal with disjunctions on the right of the $\triangleleft$ connective, disjunctions on its left need to be handled specially: $(\Xi_1 \vee \Xi_2) \models P$ is weaker than $(\Xi_1 \models P) \vee (\Xi_2 \models P)$ as it could be that the selection is not yet decided in P, but will only be after a few transitions. This is addressed by first picking a full transition sequence and *then only* requiring the outcome of the selection to be decided, which can be seen in the definition in "$\exists\alpha$ s.t.". Correctness is stated similarly to the usual notion of *fairness* ("if a particular transition is constantly available, it will eventually occur") but with a strategy instead of a particular transition. Note how the transition sequence interleaves single invocations of the strategy between arbitrarily long transition sequences: this permits stating results in presence of divergence but still correct with a stochastic scheduler. The "eventually" aspect of activeness is given by "$\exists n$ s.t.". "Immediately correct" essentially means the corresponding port or sum is at top-level.

If a type Γ is correct for a process then so is any Γ' with $\Gamma' \succeq \Gamma$ (Lemma "Bisimulations and Type Equivalence" in [6], Section 4).

Definition 2 (Correctness). *Let Γ be a type and P a process. We say that or Γ is* correct *for P if, for some strategy f, for any infinite sequence of the form $(\Gamma; P) = (\Gamma_0; P_0) \xrightarrow{\tilde{\mu}_0}\searrow (\Gamma'_0; P'_0) \xrightarrow{f} (\Gamma'_1; P'_1) \cdots \xrightarrow{\tilde{\mu}_i}\searrow (\Gamma'_i; P'_i) \xrightarrow{f} (\Gamma_{i+1}; P_{i+1}) \cdots$: Let (for all i) p_i be the subject of the $(\Gamma'_i; P'_i) \xrightarrow{f} (\Gamma_{i+1}; P_{i+1})$ transition (or "τ" if it is the identity or a τ-transition). Then there is a number n and a resource α such that:*

1. *for all i with $p_i \neq \tau$, $(\alpha \triangleleft \overline{p_i}_{\mathbf{A}}) \preceq \Gamma'_i$*
2. *For some ε with $(\alpha \triangleleft \varepsilon) \preceq \Gamma_n$, $\alpha \triangleleft \varepsilon$ is immediately correct for $(\Gamma_n; P_n)$.*

We now sketch a proof that Γ_A given in (4) is a correct type for A given in (1). We only pick a representative transition sequence, but of course a complete proof would have to take all possible transitions into account. Following the pattern given in Definition 2 we alternate arbitrary transition sequences $\tilde{\mu}_i$ (odd-numbered steps) and those provided by the strategy (even-numbered steps).

1. We first send a request $\tilde{\mu}_0 = r(uv)$. The resulting type is (5) on page 9.

2. The strategy executes $\overline{a}(\boldsymbol{\nu}t'f')$ to bring the process closer to an output on u or v. This is allowed, as the subject's complement a is active in the dependencies. The local dependency network is now $(\bar{u}_{\mathbf{A}} \vee \bar{v}_{\mathbf{A}}) \lhd (a_{\mathbf{AR}} \wedge (\bar{t}'_{\mathbf{A}} \vee \bar{f}'_{\mathbf{A}}) \wedge b_{\mathbf{AR}})$.
3. As we do not want to help the strategy find the way out we set $\tilde{\mu}_1 = \varnothing$. However we must still do a projection "$\searrow$", i.e. simulate the choice made by the a-input. Let's pick $\bar{f}'$: $(\bar{u}_{\mathbf{A}} \vee \bar{v}_{\mathbf{A}}) \lhd (a_{\mathbf{AR}} \wedge \bar{f}'_{\mathbf{A}} \wedge b_{\mathbf{AR}})$.
4. The process is now $A \,|\, (t'.\overline{b}\langle uv\rangle + f'.\bar{v})$, so the strategy is just to consume the f' prefix, which is permitted because its complement is active ($\bar{f}'_{\mathbf{A}}$).
5. We are now at $A|\bar{v}$. If we set $\tilde{\mu}_2 = \varnothing$ at this point, $n = 2$ satisfies the requirement as $\bar{v}$ is at top-level. If instead we consume $\bar{v}$ with $\tilde{\mu}_2 = \bar{v}$, the transition operator removes activeness of both $\bar{u}$ and $\bar{v}$, and the process type becomes $(\bar{u}_{\mathbf{A}} \vee \bar{v}_{\mathbf{A}}) \lhd \bot \cong \top$ which is vacuously correct.

4 Type System

Given a process P, a mapping Σ of channel types for all free names, and optionally multiplicities for some names, our type system constructs a process type Γ for P. Processes that may violate multiplicity constraints or mismatch channel types are rejected. Typing is decidable and sound, but necessarily not complete (may reject safe processes, or construct a behavioural statement weaker than what is actually correct for the process). Most rules of the type system are straightforward (every process constructor has a corresponding process type operator). Appendix C presents the full system.

We focus on the *prefix* rule. It $\odot$-composes five statements, in order: subject type and total multiplicities, subject activeness, continuation, expected remote behaviour and subject responsiveness.

$$\frac{\Gamma \vdash P \quad \mathsf{sub}(G) = p \quad \mathsf{obj}(G) = \tilde{x} \qquad (\#(G) = 1 \text{ and } m' = \star) \Rightarrow \varepsilon = \bot}{\begin{array}{r}(\boldsymbol{\nu}\mathrm{bn}(G))\Big(\ \Big(p : \sigma;\ \blacktriangleleft p^{m} \wedge \bar{p}^{m'}\Big) \odot \\ \Big(;\, p_{\mathbf{A}}^{\#(G)} \lhd \varepsilon \blacktriangleleft\Big) \odot \\ \Gamma \lhd \bar{p}_{\mathbf{A}} \odot \\ \overline{\sigma}[\tilde{x}] \lhd \bar{p}_{\mathbf{AR}} \odot \\ (;\, p_{\mathbf{R}} \lhd \sigma[\tilde{x}] \blacktriangleleft)\ \Big) \vdash G.P\end{array}} \ \text{(R-PRE)}$$

We illustrate the five factors in order with the derivation of $r_{\mathbf{R}} \lhd (a_{\mathbf{AR}} \wedge b_{\mathbf{AR}})$ as a type for (1) on page 3. We omit parts not needed to get $r_{\mathbf{R}}$'s dependencies.

Subject type, multiplicities and activeness. The parameter-less output $\bar{f}$ is typed using (R-PRE). The name is linear ($m = m' = 1$) and, since there are no parameters or continuation, all but the first two factors of the typing are empty, leaving us with: $\big(f : ();\ \blacktriangleleft \bar{f}^1 \wedge f^1\big) \odot \big(;\, \bar{f}_{\mathbf{A}}^1 \lhd \top \blacktriangleleft\big)$, or:

$$\Gamma_6 = \big(f : (); \bar{f}_{\mathbf{A}}^1 \blacktriangleleft \bar{f}^0 \wedge f^1\big) \vdash \bar{f} \tag{6}$$

Continuation. A sequence $G.P$ is typed like composition $G|P$, except that activeness resources in P additionally depend on $\bar{p}_{\mathbf{A}}$, p being G's subject port. Here, $f'.\bar{f}$ is again typed with (R-PRE), where the first three terms are now non-null:

$$\Big(f' : (); \blacktriangleleft f'^1 \wedge \bar{f}'^1\Big) \odot \Big(; f'^1_{\mathbf{A}} \lhd \top \blacktriangleleft\Big) \odot \varGamma_6 \lhd \bar{f}'_{\mathbf{A}} \vdash f'.\bar{f}$$

Dropping the unneeded $f'^1_{\mathbf{A}}$ statement we get

$$\varGamma_T = \Big(f : (), f' : (); \bar{f}_{\mathbf{A}} \lhd \bar{f}'_{\mathbf{A}} \blacktriangleleft \bar{f}^0 \wedge f^1 \wedge f'^0 \wedge \bar{f}'^1\Big) \vdash f'.\bar{f} \tag{7}$$

Remote behaviour plays two roles, respectively through the local and environment parts of the instantiated channel $\overline{\sigma}[tf]$. First, if the input on a channel is active and responsive, it will behave according to the protocol specified in the channel type whenever queries are sent to it. For $\overline{b}\langle tf\rangle$, this is $(\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}}) \lhd b_{\mathbf{AR}}$, where the left side is (3) from page 6 with t and f replacing 1 and 2. Second, it sets upper bounds on the local side's use of parameters ports. In this case we get $t^1 \vee f^1$ in the environment side, which effectively prevents any part of the process to do at t and f anything more than an input-guarded sum at t and f. Together with the subject b handled as in (6), we get the following:

$$\big(b : \mathsf{Bool}, t : (), f : (); (\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}}) \lhd b_{\mathbf{AR}} \blacktriangleleft (t^1 \vee f^1) \wedge (\bar{b}^\star \wedge b^\omega)\big) \vdash \overline{b}\langle tf\rangle \tag{8}$$

As in (7), the t'-prefix adds a dependency on $\bar{t}'_{\mathbf{A}}$ to all activeness resources:

$$\varGamma_F = \big(\varSigma; (\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}}) \lhd (b_{\mathbf{AR}} \wedge \bar{t}'_{\mathbf{A}}) \blacktriangleleft (t^1 \vee f^1) \wedge (\bar{b}^\star \wedge b^\omega)\big) \vdash t'.\overline{b}\langle tf\rangle \tag{9}$$

A sum $T+F$ is given the type $(t'+f')_{\mathbf{A}} \wedge (\varGamma_T \vee \varGamma_F)$, where $\varGamma_T$ (here (7)) and $\varGamma_F$ (here (9)) are respectively the types of T and F, and t', f' their guards: the process offers a branching $t'+f'$, and ("$\wedge$") selects ("$\vee$") one of $\varGamma_T$ and $\varGamma_F$ (we use $(\varSigma; \varXi_{\mathrm{L1}} \blacktriangleleft \varXi_{\mathrm{E1}}) \vee (\varSigma; \varXi_{\mathrm{L2}} \blacktriangleleft \varXi_{\mathrm{E2}}) \stackrel{\text{def}}{=} (\varSigma; \varXi_{\mathrm{L1}} \vee \varXi_{\mathrm{L2}} \blacktriangleleft \varXi_{\mathrm{E1}} \wedge \varXi_{\mathrm{E2}})$ for $\varGamma_T \vee \varGamma_F$). The decoupling between the guards and the continuations is done to make explicit which channels must be used to make the process branch.

$$\begin{aligned}\big(\varSigma; (t'+f')_{\mathbf{A}} \wedge \Big(\big((\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}}) \lhd (b_{\mathbf{AR}} \wedge \bar{t}'_{\mathbf{A}})\big) \ \vee\ \big(\bar{f}_{\mathbf{A}} \lhd \bar{f}'_{\mathbf{A}}\big)\Big) \blacktriangleleft \\ \Big(\bar{f}^0 \wedge f'^0 \wedge \bar{f}'^1\Big) \wedge \Big((t^1 \vee f^1) \wedge \bar{b}^\star \wedge b^\omega\Big)\big) \vdash t'.\overline{b}\langle tf\rangle + f'.\bar{f}\end{aligned} \tag{10}$$

We run (R-PRE) once more for the full a-output. Now two names are bound $(\mathrm{bn}(\overline{a}(\boldsymbol{\nu} t'f')) = \{t', f'\})$, and we only need the third and fourth factors:
Remote behaviour $\Big(t' : (), f' : (); (\bar{t}'_{\mathbf{A}} \vee \bar{f}'_{\mathbf{A}}) \lhd a_{\mathbf{AR}} \blacktriangleleft t'^1 \vee f'^1\Big)$ and
Continuation $\big(\varSigma; (\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}}) \lhd (b_{\mathbf{AR}} \wedge \bar{t}'_{\mathbf{A}} \wedge a_{\mathbf{A}}) \vee (\bar{f}_{\mathbf{A}} \lhd (\bar{f}'_{\mathbf{A}} \wedge a_{\mathbf{A}})) \blacktriangleleft$
$(t^1 \vee f^1) \wedge \bar{b}^\star \wedge b^\omega \wedge \bar{f}^0 \wedge f'^0 \wedge \bar{f}'^1\big)$.

The $\odot$ operator now does some dependency reduction (Definition 1 on page 7): The remote behaviour provides $(\bar{t}'_{\mathbf{A}} \lhd a_{\mathbf{AR}}) \vee (\bar{f}'_{\mathbf{A}} \lhd a_{\mathbf{AR}})$, and the continuation $(\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}}) \lhd t'_{\mathbf{A}} \quad \vee \quad (\bar{f}_{\mathbf{A}} \lhd f'_{\mathbf{A}})$. Remember that $(p_{\mathbf{A}} \lhd \gamma) \wedge (\alpha \lhd p_{\mathbf{A}}) \hookrightarrow$

$(p_{\mathbf{A}} \lhd \gamma) \wedge \alpha \lhd (p_{\mathbf{A}} \vee \gamma)$, so the two dependency statements in the continuation become respectively $(\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}}) \lhd (t'_{\mathbf{A}} \vee a_{\mathbf{AR}})$ and $\bar{f}_{\mathbf{A}} \lhd (f'_{\mathbf{A}} \vee a_{\mathbf{AR}})$. Therefore composing remote behaviour and continuation and binding (dropping) t' and f' yields:

$$\big(a : \mathsf{Bool}, t : (), f : (); (\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}}) \lhd (b_{\mathbf{AR}} \wedge a_{\mathbf{AR}}) \vee \bar{f}_{\mathbf{A}} \lhd a_{\mathbf{AR}} \blacktriangleleft a^{\omega} \wedge (t^1 \vee f^1)\big) \vdash \overline{a}(\boldsymbol{\nu} t' f').(t'.\overline{b}\langle tf \rangle + f'.\bar{f}) \quad (11)$$

Subject responsiveness. A port is responsive if it provides all resources given in the channel type, which is what the last statement in the (R-Pre) rule states. For $r(tf)$, this is written $r_{\mathbf{R}} \lhd (\bar{t}_{\mathbf{A}} \vee \bar{f}_{\mathbf{A}})$, where the right hand side is just (2) from page 5 with t and f replacing 1 and 2. Composing with (11) reduces the dependency chain and we obtain $r_{\mathbf{R}} \lhd (b_{\mathbf{AR}} \wedge a_{\mathbf{AR}})$, as required.

The type system sketched above has two important properties. It agrees with the transition operator on the type of a process after a transition...

Proposition 1 (Subject Reduction). $(\Gamma; P) \xrightarrow{\mu} (\Gamma \wr \mu; P')$ *implies* $\exists \Gamma'$ *s.t.* $\Gamma' \preceq \Gamma \wr \mu$ *and* $\Gamma' \vdash P'$.

...and decidable typability implies undecidable correctness.

Proposition 2 (Type Soundness). *If* $\Gamma \vdash P$ *then* $\Gamma \models P$.

5 Related Works

Acciai and Boreale's work on *Responsiveness* [1] (essentially our activeness, except that they work in a reduction-based setting, while we have to take the environment into account) addresses concerns very close to ours. It does not support choice or conditional properties, as it uses numerical levels to track dependencies, but presents an extension for recursive processes, in that it permits handling unbounded recursion such as a "factorial" function. Our dependency analysis would reject such a process, as the recursive call would create a dependency $f_{\mathbf{R}} \lhd f_{\mathbf{R}}$, that reduces to $f_{\mathbf{R}} \lhd \bot$. We conjecture that "delayed dependencies" [6] would permit integrating their recursion analysis with our work.

Kobayashi's *Livelock-Freedom Type System* (implemented as TyPiCal [8, 9]), does a very fine analysis of channel *usages*. Instead of counting how many times a port may be used, they permit arbitrary *channel usages* that describe using a CCS-like language in what way and order the two ports of a channel may be used. This permits describing usages such as "every input must be followed by an output". Using numerical levels, basic dependency relations can be forced between elements of the usages of different channels. This prevents encoding of selection and branching as it amounts to having no "$\vee$" in behavioural statements, but permits analysing other usage patterns such as semaphores, which the present work would dismiss as unreliable $\star$-multiplicities.

Kobayashi and Sangiorgi's *Hybrid Type System* for lock-freedom [10] combines (arbitrary) deadlock, termination and confluence type systems on *subprocesses* of the one being analysed (thereby permitting analysis of globally divergent processes). This work uses typed transitions reminiscent of ours, and their “robust” properties are analogous to our semantics permitting arbitrary transition sequences $\tilde{\mu}_i$. Channel usages are like those used by Kobayashi in previous works [8, 9], with the same expressive power and limitations. The typing rules discard those processes that rely on the environment in order to fulfil their obligation. Hence well-typed processes are lock-free without making any assumption on the environment. Advanced termination type systems such as those proposed by Deng and Sangiorgi [4] permit this hybrid system to deal with complex recursive functions like tree traversal.

The three following papers have a *generic* approach, as opposed to the previous ones (and the present paper) that are aimed at specific properties. They have to be *instantiated* with the desired property, expressed in various ways.

Kobayashi's *Generic Type System* [7] is a general purpose type system that can be *instantiated* with a subtyping relation and a consistency condition on types, resulting in type systems for various safety properties (unlike activeness which is a liveness property). Types are CCS-like abstractions of the process, and the consistency condition verifies that the type enjoys the desired property. Its types use “+” in essentially the same sense as we do, and “&” corresponds precisely to our $\vee$. The paper includes as examples of instantiations, arity-mismatch checking, race-freedom and deadlock-freedom type systems. However, simply by providing a subtyping relation and a consistency predicate one does not get the desired results “for free”. It is still necessary to prove several technical lemmas.

Caires and Vieira's *Spatial Logic Model Checker* [3] checks processes for a wide range of properties, expressed by expressions in a *spatial logic*. Activeness of a port p can be written $\mu X.(\langle p\rangle \vee \Box\Diamond X)$. Responsiveness of a port depends on the channel type, but it should be possible to give an inductive translation of channel types to modal formulæ corresponding to responsiveness on it. The selection connective $\vee$ is also present, with the same meaning. There is no direct equivalent of $\triangleleft$, so conditional properties need to be encoded by modifying the activeness formulæ, which may become too complex with dependencies on responsiveness as in $r_{\mathbf{R}}\triangleleft(a_{\mathbf{AR}}\wedge b_{\mathbf{AR}})$ (Section 4). Both its strengths and limitations come from it being a *model checker*. On the one hand, it takes logical formulæ in *input* rather than constructing them, it has a large complexity due to exhaustively exploring the state space, and doesn't terminate when given unbounded processes (our type system is polynomial in the process size and always terminates). On the other hand it is *complete* for bounded processes, and recognises activeness in cases deemed unsafe by our system due to over-approximation.

Acciai and Boreale's *Spatial Type System* [2] combines ideas from Kobayashi's Generic Type System (types abstract the behaviour of processes) and Spatial Logic, by performing model checking with spatial formulæ on the types rather than on the processes. This results in a generic type system able to characterise liveness properties such as activeness and supporting choice, both through the process constructor + and logical connective $\vee$. It is parametrised by “shallow” (without direct access to the object parts of transitions) logical formulæ, that it verifies using model-checking. Being based on model checking, it suffers from the same limitations as the previous work, in terms of computation complexity, and difficulty of expressing conditional properties or responsiveness (again, "responsiveness" in that paper corresponds to our "activeness"). On the other hand, restricting it to shallow logic formulæ allows working on the abstracted process, making it more efficient than a fully general model checker. Like the previous work and unlike the Generic Type System, it doesn't require proving soundness of a consistency predicate, as it is based on a fixed formula language.

6 Conclusion

We described a type notation and semantics that combine statements about liveness properties ($s_{\mathbf{A}}$ and $p_{\mathbf{R}}$), choice (through branching $(p+q)_{\mathbf{A}}$ and selection $\Delta \vee \Delta$) and conditional properties ($\Delta \triangleleft \Theta$). Then the type system outlined in Section 4 is able, given a process P, channel types and optionally port multiplicities, to construct a process type whose local component Ξ_{L} contains all information the type system was able to gather about P's behaviour. As the type system is sound and decidable, it is necessarily incomplete, but still powerful enough to recognise activeness and responsiveness in many important applications such as data representation or conversation-based programming. We chose to focus on choice itself, leaving out features like recursivity [1] subtyping [13], and complex channel usages such as locks [8], well explored before in a choice-less context.

References

1. Lucia Acciai and Michele Boreale. Responsiveness in process calculi. *Theoretical Computer Science*, 409(1):59–93, 2008.
2. Lucia Acciai and Michele Boreale. Spatial and behavioral types in the pi-calculus. In *Proceedings of CONCUR'08*, volume 5201 of *LNCS*, pages 372–386. Springer, 2008.
3. Luís Caires. Behavioral and spatial observations in a logic for the π-calculus. In *Proceedings of FOSSACS'04*, volume 2987 of *LNCS*. Springer, 2004.
4. Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Information and Computation*, 204(7):1045–1082, 2006.
5. Maxime Gamboni and António Ravara. Activeness and responsiveness in mobile processes. In *7th Conference on Telecommunications*, pages 429–432. Instituto de Telecomunicações, 2009.
6. Maxime Gamboni and António Ravara. Responsive choice in process calculi. Technical report, SQIG — IT and IST, UTL Portugal, 2009. http://gamboni.org/i.pdf.
7. Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. *ACM SIGPLAN Notices*, 36(3):128–141, 2001.
8. Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.
9. Naoki Kobayashi. Typical 1.6.2, 2008.
10. Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. In *Proceedings of CAV'08*, volume 5123 of *LNCS*, pages 80–93. Springer, 2008.

11. Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification, Proceedings of the International NATO Summer School (Marktoberdorf, Germany, 1991)*, volume 94 of *NATO ASI Series F*. Springer, 1993.
12. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i and ii. *Information and Computation*, 100(1):1–77, 1992.
13. Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of LICS'93*, pages 376–385. IEEE Computer Society, 1993.
14. Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999.
15. Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

A Labelled Transition System

The labelled transition system is inductively defined by the following rules. Labels, ranged over by μ, are τ, input $a(\tilde{x})$ and output $(\boldsymbol{\nu}\tilde{z} : \tilde{\sigma})\,\overline{a}\langle\tilde{x}\rangle$ where $a \notin \tilde{z} \subseteq \tilde{x}$.

$$\frac{-}{\overline{a}\langle\tilde{x}\rangle.P \xrightarrow{\overline{a}\langle\tilde{x}\rangle} P}\ (\textsc{Out}) \qquad \frac{-}{a(\tilde{y}).P \xrightarrow{a(\tilde{x})} P\{^{\tilde{x}}/_{\tilde{y}}\}}\ (\textsc{Inp})$$

$$\frac{P \xrightarrow{(\boldsymbol{\nu}\tilde{y}:\tilde{\theta})\,\overline{a}\langle\tilde{x}\rangle} Q \quad z \in \tilde{x} \setminus (\{a\} \cup \tilde{y})}{(\boldsymbol{\nu}z : \sigma)\,P \xrightarrow{(\boldsymbol{\nu}z:\sigma,\tilde{y}:\tilde{\theta})\,\overline{a}\langle\tilde{x}\rangle} Q}\ (\textsc{Open})$$

$$\frac{P \xrightarrow{\mu} P'}{!\,P \xrightarrow{\mu} P' \,|\, !\,P}\ (\textsc{Rep}) \qquad \frac{P \xrightarrow{\mu} Q \quad z \notin \mathrm{n}(\mu)}{(\boldsymbol{\nu}z : \sigma)\,P \xrightarrow{\mu} (\boldsymbol{\nu}z : \sigma)\,Q}\ (\textsc{New})$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \varnothing}{P \,|\, Q \xrightarrow{\mu} P' \,|\, Q \qquad Q \,|\, P \xrightarrow{\mu} Q \,|\, P'}\ (\textsc{Par})$$

$$\frac{P \xrightarrow{(\boldsymbol{\nu}\tilde{z}:\tilde{\sigma})\,\overline{a}\langle\tilde{x}\rangle} P' \quad Q \xrightarrow{a(\tilde{x})} Q' \quad \tilde{z} \cap \mathrm{fn}(Q) = \varnothing}{\begin{array}{c} P \,|\, Q \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{z} : \tilde{\sigma})\,(P' \,|\, Q') \\ Q \,|\, P \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{z} : \tilde{\sigma})\,(Q' \,|\, P') \end{array}}\ (\textsc{Com})$$

$$\frac{P \xrightarrow{\mu} P'}{P{+}Q \xrightarrow{\mu} P' \quad Q{+}P \xrightarrow{\mu} P'}\ (\textsc{Sum})$$

$$\frac{P \equiv_\alpha P' \quad P' \xrightarrow{\mu} Q' \quad Q' \equiv_\alpha Q}{P \xrightarrow{\mu} Q}\ (\textsc{Cong})$$

B Weakening on behavioural statements

Definition 3 (Weakening Relation).
Relation $\preceq$ is the smallest preorder defined by the following rules, where $\cong$ is its symmetric closure.

1. *On behavioural statements or process types (ranged over by η):*
   - $\eta_1 \wedge \eta_2 \preceq \eta_1 \preceq \eta_1 \vee \eta_2$, *and* $\bot \preceq \eta \preceq \top$. $\eta \wedge (\eta_1 \vee \eta_2) \cong (\eta \wedge \eta_1) \vee (\eta \wedge \eta_2)$.
   - $\wedge$ *and* $\vee$ *are commutative, associative and idempotent, up to* $\cong$.
   - *If* $\eta_1 \preceq \eta_2$ *then* $\eta \wedge \eta_1 \preceq \eta \wedge \eta_2$ *and* $\eta \vee \eta_1 \preceq \eta \vee \eta_2$.
   - *If* $\eta_1 \cong \eta_2$ *then* $\gamma \triangleleft \eta_1 \cong \gamma \triangleleft \eta_2$, $(\eta \blacktriangleleft \eta_1) \cong (\eta \blacktriangleleft \eta_2)$ *and* $(\eta_1 \blacktriangleleft \eta) \cong (\eta_2 \blacktriangleleft \eta)$.

2. *On multiplicities,* $m_1 \preceq m_2$ *and* $p^{m_1} \preceq p^{m_2}$ *if* $m_1 = 0$ *or* $m_2 \in \{m_1, \star\}$. *Also,* $p^\star \cong \top$.
3. *On dependency statements:* $(\gamma \triangleleft \varepsilon_1) \wedge (\gamma \triangleleft \varepsilon_2) \cong \gamma \triangleleft (\varepsilon_1 \vee \varepsilon_2)$, $(\gamma \triangleleft \varepsilon_1) \vee (\gamma \triangleleft \varepsilon_2) \cong \gamma \triangleleft (\varepsilon_1 \wedge \varepsilon_2)$ *and* $\gamma \triangleleft \bot \cong \top$

C Type System

The type system is constituted by the following rules. (R-Pre) is detailed in Section 4, and the reader is invited to have a look at the technical report for a detailed discussion of the notation and operators used in the other rules.

$$\frac{-}{(\varnothing; \top \blacktriangleleft \top) \vdash \mathbf{0}} \text{ (R-Nil)}$$

$$\frac{\forall i : \Gamma_i \vdash P_i}{\Gamma_1 \odot \Gamma_2 \vdash P_1 \mid P_2} \text{ (R-Par)} \qquad \frac{\Gamma \vdash P \qquad \Gamma(x) = \sigma}{(\boldsymbol{\nu} x)\, \Gamma \vdash (\boldsymbol{\nu} x : \sigma)\, P} \text{ (R-Res)}$$

$$\frac{\begin{array}{c}\forall i : \big(\mathsf{sub}(G_i) = \{p_i\}, \quad (\Sigma_i; \Xi_{\mathrm{L}i} \blacktriangleleft \Xi_{\mathrm{E}i}) \vdash G_i.P_i\big) \\ \Xi_{\mathrm{E}} \preceq \bigwedge_i \Xi_{\mathrm{E}i} \\ \big(\Xi_{\mathrm{E}} \text{ has concurrent environment } p_{i'}\big) \Rightarrow \varepsilon = \bot\end{array}}{\big(\bigwedge_i \Sigma_i; (\sum_i p_i)_{\mathbf{A}} \triangleleft \varepsilon \wedge \bigvee_i \Xi_{\mathrm{L}i} \blacktriangleleft \Xi_{\mathrm{E}}\big) \vdash \sum_i G_i.P_i} \text{ (R-Sum)}$$

$$\frac{\begin{array}{c}\Gamma \vdash P \quad \mathsf{sub}(G) = p \quad \mathsf{obj}(G) = \tilde{x} \\ (\#(G) = 1 \text{ and } m' = \star) \Rightarrow \varepsilon = \bot\end{array}}{\begin{array}{r}(\boldsymbol{\nu}\mathrm{bn}(G))\Big(\ \Big(p : \sigma;\ \blacktriangleleft p^m \wedge \bar{p}^{m'}\Big) \odot \\ \Big(; p_{\mathbf{A}}^{\#(G)} \triangleleft \varepsilon \blacktriangleleft\Big) \odot \\ \Gamma \triangleleft \bar{p}_{\mathbf{A}} \odot \\ \overline{\sigma}[\tilde{x}] \triangleleft \bar{p}_{\mathbf{AR}} \odot \\ (; p_{\mathbf{R}} \triangleleft \sigma[\tilde{x}] \blacktriangleleft)\ \Big) \vdash G.P\end{array}} \text{ (R-Pre)}$$

In the rule (R-Sum), a process type having no “concurrent environment $p_{i'}$” prevents a third-party process to attempt selecting more than one branch of the sum, and, by contraposition, guarantees that any attempt to select a branch of the sum (by communicating with its guard) will succeed, which is what activeness of the branching means.