Type checking liveness properties of mobile processes

Maxime Gamboni[1]

Instituto de Telecomunicações, Instituto Superior Técnico

October 30, 2008

[1] Joint work with António Ravara

Outline

1 Motivation

2 TyPiCal

3 Receptiveness, Responsiveness, Termination

4 Our Work

Motivation

This work uses the following language:

The Synchronous Polyadic π-calculus

$$P \quad ::= \quad \mathbf{0} \quad | \quad a(\tilde{x}).P \quad | \quad \overline{a}\langle\tilde{v}\rangle.P \quad | \quad (P|P) \quad | \quad (\nu x)\, P \quad | \quad !\, P$$

Running example:

- Client-Server interaction: $S \mid C$
- Example Server:
  $S = !\, a(x, n).(\text{some processing}).\overline{x}\langle r\rangle$
- Example Client:
  $C = \overline{a}\langle t, 42\rangle.t(n).P$

We now define a number of properties we would like this system to satisfy, and terminology used in this presentation.

Simple Types

Definition (Simple Type)

The simple type σ of a name is either a data type (*Int*, *Bool*, etc) or a channel type $\text{chan}(\tilde{\sigma})$.

Properties we want to verify (1)

$!\, a(x, n).(\text{some processing}).\overline{x}\langle r\rangle \mid \overline{a}\langle t, 42\rangle.t(n).P$

Simple Types

- There should exist a mapping of names to types that is consistent over the process.
- Types of values passed over a channel should match the parameter types of the channel's channel type.
- E.g., $a : \text{chan}(\text{chan}(\mathit{Int}), \mathit{Int})$ matches $t : \text{chan}(\mathit{Int})$, $42 : \mathit{Int}$.

Activeness

Definition (Activeness)

- *Activeness* $p_{\mathbf{A}}$ of a port $p \in \{a, \bar{a}\}$ in a process P: Ability of P to *reliably* receive ($p = a$) or send ($p = \bar{a}$) a message on it.
- *Strong Activeness* additionally requires the input (resp., output) transition to be available without prior τ-reduction.
- *ω-Activeness* additionally requires the activeness property to hold an arbitrarily large number of times.
- *Uniform activeness* of a port requires all requests to a name to be handled with the same continuation.

The definition of "reliable" depends on to what extent the environment may interfere.
Also note that some authors use for "activeness" the unrelated meaning of "outputs not under replication".

Activeness Examples

- Strong ω-activeness $\Rightarrow$ ω-activeness $\Rightarrow$ activeness
- Strong ω-activeness $\Rightarrow$ strong activeness $\Rightarrow$ activeness.
- Mentioning only the strongest property, assuming no environment interference:

$!\,a \mid b \mid a$:	a is strong ω-active.
$b.!\,a \mid \overline{b}$:	a is ω-active.
$b.a \mid \overline{b} \mid a$:	a is strong active.
$b(x).x \mid \overline{b}\langle a\rangle$:	a is active.
$\overline{t} \mid t \mid t.!\,a$:	a is not active.

- $!\,a(x).P$ is strong uniform ω-active on a.
- $a(x).(P \mid !\,a(x).Q)$ is strong *non-uniform* ω-active on a.

Properties we want to verify (2)

$!\, a(x, n).(\text{some processing}).\overline{x}\langle r\rangle \mid \overline{a}\langle t, 42\rangle.t(n).P$

Activeness

- The server should be ω-active on its input port a.

Responsiveness

Definition (Responsiveness)

Responsiveness $p_{\mathbf{R}}$ of a port p in a process P is the ability to reliably respond ($p = a$) or provide parameters ($p = \bar{a}$) to a request.

- "Respond" and "provide parameters" means being *active* and *responsive* at the parameters
- Note that for this to make sense we need *IO-Types*, i.e. which parameter polarity must be used by the server and the client.
- Activeness and responsiveness on a given port aren't related — activeness tells if a message is guaranteed to be exchanged; responsiveness tells what happens afterwards.

Responsiveness Examples

- Let $P = a(x).Q$ with $a \notin \mathrm{fn}(Q)$ and $x : \mathrm{chan}(Int)$. Then, assuming IO-alternation, a is responsive in P if and only if, for all b, having $P \xrightarrow{a(b)} Q'$, $\bar{b}$ is active in Q'.
- Port a is active and responsive in $a(x).\overline{x}\langle 3\rangle$.
- Writing $\bot.P$ for $(\nu t)\, t.P$, a is active but not responsive in $a(x).\bot.\overline{x}\langle 3\rangle$.
- Writing $?.P$ for $(\nu t)\,(\bar{t}|t|t.P)$, a is responsive but not active in $?.a(x).\overline{x}\langle 3\rangle$. It is active but not responsive in $a(x).?.\overline{x}\langle 3\rangle$.
- Port a is vacuously responsive in $\bot.a(x).Q$ for all Q.
- Port $\bar{b}$ is *not* responsive in $\overline{b}\langle a\rangle.a(x).\bot.\overline{x}\langle 3\rangle$, because its parameter a isn't.

Properties we want to verify (3)

$!\, a(x, n).(\text{some processing}).\overline{x}\langle r\rangle \mid \overline{a}\langle t, 42\rangle.t(n).P$

Responsiveness

- The server should be responsive on its input port a (in this case, active on the output port $\bar{x}$)
- The client should be responsive on the output port $\bar{a}$ (in this case, active on the input port t)

Termination

Definition (Termination)

- A process P terminates if all its reduction sequences are finite in length.
- A port p in a process P terminates if, for any $P \xrightarrow{\mu} Q$ with $\mathrm{sub}(\mu) = p$, all reduction sequences caused by μ are finite in length.

- It can be tricky to formally define "caused by" ; intuitively, a reduction $Q \longrightarrow Q'$ is caused by μ if at least one of the communication partners has been brought to top-level by μ. A reduction sequence is caused by a transition if every reduction is caused by a transition earlier in that sequence.

Termination Examples

- Any process without replication terminates and all its ports terminate as well.
- In $!\,a(x).\overline{b}\langle x\rangle \mid !\,b(x).\overline{x}\langle 3\rangle$, both a and b terminate (and so does the process).
- In $!\,a(x).\overline{b}\langle x\rangle \mid !\,b(x).\overline{a}\langle x\rangle$, neither a nor b terminates, but the process terminates (it has no reductions)
- In $!\,\overline{a}\langle b\rangle \mid !\,a(x).\overline{x}\langle 3\rangle$, all ports terminate but the process doesn't (every request to a is handled finitely, but there's an unbounded number of them).
- Let $\Omega = (\nu t)\,(!\,t(x).\overline{t}\langle x\rangle|\overline{t}\langle x\rangle)$. Then in $a(x).(\overline{x}\langle 3\rangle \mid \Omega)$ the process terminates but a doesn't, and in $(a(x).\overline{x}\langle 3\rangle) \mid \Omega$, a terminates but the process doesn't.

Termination Examples (2)

Note that termination and responsiveness are not directly related:

- In $a(x).(\overline{x}\langle 3\rangle \mid \Omega)$, a is responsive but doesn't terminate.
- In $a(x).\bot.\overline{x}\langle 3\rangle$, a terminates but isn't responsive.

Properties we want to verify (4)

$!\, a(x, n).(\text{some processing}).\overline{x}\langle r\rangle \mid \overline{a}\langle t, 42\rangle.t(n).P$

Termination

- The server input port a should terminate.

Outline

1 Motivation

2 TyPiCal

3 Receptiveness, Responsiveness, Termination

4 Our Work

TyPiCal

An implementation of a lock-freedom type system (Naoki Kobayashi)

http://www.kb.ecei.tohoku.ac.jp/~koba/typical/

```
TyPiCal 1.6.1: A Type-based static analyzer for the Pi-Calculus
Usage:
  typical [option] filename
  Available options are:
    -d: deadlock-freedom analysis
    -i: information flow analysis
    -l: lock-freedom analysis (default)
    -wl: weak lock-freedom analysis
    -wlauto: weak lock-freedom analysis (with termination annotation inference)
    -s: simple type inference
    -u: useless code elimination
    -t: termination analysis
```

TyPiCal analysis

We now introduce a few concepts used by TyPiCal when analysing processes.

Deadlock-Freedom

As a dual to activeness, we introduce deadlock-freedom:

Definition (Deadlock)

- An input or output prefix in a process P is *deadlocked* if it is top-level and P can't be reduced.
- An input or output prefix in a process P is *deadlock-free* if no reduction of P leads to that prefix being deadlocked.

A sufficient condition for deadlock-freedom on an action on port p is to have the complement port $\bar{p}$ active — if $\bar{p}$ is active then either the process can be reduced or $\bar{p}$ is top-level and can communicate with p.

Deadlock Examples

- If $\nexists Q : P \longrightarrow Q$ then all top-level actions in P are deadlocked.
- In $!\, a(x).P \mid Q$, all a-outputs are deadlock-free.
- In $a.\bar{b} \mid b.\bar{a}$, both a and b are deadlocked.
- In $P = ?.a \mid \bar{a}$, a is deadlock-free, but $\bar{a}$ isn't because $P \longrightarrow \equiv \bot.a \mid \bar{a}$ in which $\bar{a}$ is deadlocked, although $P \longrightarrow \sim a \mid \bar{a}$ in which $\bar{a}$ is deadlock-free.
- All of $P = !\, a(x).\overline{b}\langle x\rangle \mid !\, b(x).\overline{a}\langle x\rangle \mid \overline{a}\langle s\rangle \mid s$, in particular the s-input, is deadlock-free, because P can always be reduced.
- All of $P|\Omega$ is deadlock-free because it can always be reduced.

Livelocks

The last two examples of deadlock-freedom show that it is not a very interesting property on its own.

Definition (Livelock-freedom)

An action of a process P on a port p is *livelock-free* if it reaching top-level implies it can be consumed.

- For example, a request to a server is livelock-free is and only if it is guaranteed to be eventually received.
- Livelock-freedom implies deadlock-freedom.
- If a process terminates then livelock and deadlock-freedom are equivalent.
- We conjecture that p is livelock-free if and only if the complement port $\bar{p}$ is active (this however requires the activeness definition to be formalised in a specific way)

Livelock Examples

- In $!\,a(x).\bar{x} \mid \overline{a}\langle b\rangle \mid b$, the input at b is livelock-free.
- In $P = !\,a(x).\overline{b}\langle x\rangle \mid !\,b(x).\overline{a}\langle x\rangle \mid \overline{a}\langle s\rangle \mid s$, the s-input is not livelock-free (however all outputs are livelock-free).
- In $!\,a \mid !\,\bar{a}$, all actions are livelock-free: Any particular input or output "contained" in the replication can be consumed.

Channel Usages

Channel usages tell for a particular channel how many times the input and output ports are used, and in what order.

Channel Usages

- $U ::= \mathbf{0} \mid \rho \mid u.U \mid (U|U) \mid U\&U \mid \mu\rho.U$
  $u ::= ! \mid ?$

- Usage $!.U$ does an output and then U; Usage $?.U$ does an input and then U.
- $(U_1|U_2)$ uses according to U_1 and U_2 in parallel.
- $U_1\&U_2$ uses according to either U_1 or U_2 but not both.

We write $\text{chan}_U(\tilde{\sigma})$ for a channel of usage U and parameters $\tilde{\sigma}$. When the context is clear, we may write just the usage for a parameterless channel.

Some Examples

- $a :?, b :?|!, c :! \quad \vdash \quad a.b \mid \bar{b}.\bar{c}$

(this process uses a single port of a and c, and both ports of b, in parallel)

- $a : \mathrm{chan}_{*?|!}(!), b :! \quad \vdash \quad \mathbf{!}\, a(x).\bar{x}\langle 1\rangle \mid \bar{a}\langle b\rangle$

  $(*? \stackrel{\text{def}}{=} \mu\rho.(?.\rho))$

Note that the parameter usages give the behaviour of the channel's *input* side (the *a-input* outputs ("!") on x).

- $a :!|!|?, x :?|!, y :?, z :? \quad \vdash \quad \bar{a}.x \mid \bar{a}.y \mid a.z \mid \bar{x}.$

Here, the usages for x and y don't tell whether they will actually be input or not — just that they *may* be.

- If $a \neq t$ has usage U_1 in P and U_2 in Q, then it has usage $U_1 \& U_2$ in $(\nu t)\,(\bar{t} \mid t.P \mid t.Q)$.

TyPiCal

Obligation and Capability levels: Natural numbers similar to time tags.

- $u ::= !^{t_O}_{t_C} \mid ?^{t_O}_{t_C}$
- *Obligation level*: When is the primitive ready to fire (i.e. is at top-level)
- *Capability level*: If that primitive is at top-level, when will it actually be consumed

Some Examples

- $a : (?^0_t), b : (?^{t+1}_0 | !^0_{t+1}), c : (!^{t+2}_{t'}) \quad \vdash \quad a.b \mid \bar{b}.\bar{c}$

(Assuming a gets consumed at time t, b is ready to fire at time $t+1$, $\bar{b}$ is immediately ready, but gets actually consumed at time $t+1$. b has capability 0 because no matter when it is brought to top-level, $\bar{b}$ will be ready to communicate with it)

- $a : \mathrm{chan}_{(*?^0_\infty | !^0_0)}(!^0_{c_b}), b : (!^1_{c_b}) \quad \vdash \quad \mathbf{!}\, a(x).\overline{x}\langle 1\rangle \mid \overline{a}\langle b\rangle$

(a has capability level ∞ because it can't be fully consumed. $\bar{b}$ has obligation level 1 as its delegation to a "takes time". Parameter $\bar{x}$ has obligation 0 because as soon as a gets a request, $\bar{x}$ becomes top-level.)

- $a : (!^0_\infty | !^0_\infty | ?^0_0), x : (?^\infty_0 | !^0_\infty), y : ?^\infty_{c_y}, z : ?^1_{c_z} \quad \vdash \quad \bar{a}.x \mid \bar{a}.y \mid a.z \mid \bar{x}$

both $\bar{a}$ have capability zero because neither is guaranteed to succeed. Being at top-level, all a and $\bar{a}$ have obligation zero.

Correspondence between concepts

- A term is active if and only if it has a finite obligation level and all complement actions have a finite capability level.
- A term is strongly active if and only if it has a zero obligation level and all complement actions have a zero capability level.
- A term is livelock-free if and only if it has a finite capability level

TyPiCal examples

The black lines show the input to the program, and coloured lines the result (Irrelevant portions removed for clarity).
Green is for success (livelock-free) and red for possible lock (potentially infinite capability level).

$$a \mid a.\bar{s} \mid \bar{a} \mid \bar{a} \mid s$$
$$\rightsquigarrow \quad a \mid a.\bar{s} \mid \bar{a} \mid \bar{a} \mid s$$

(Thanks to channel usages, TyPiCal sees that inputs and outputs are balanced)

$$a \mid a.\bar{s} \mid a \mid \bar{a} \mid \bar{a} \mid s$$
$$\rightsquigarrow \quad a \mid a.\bar{s} \mid a \mid \bar{a} \mid \bar{a} \mid s$$

(It is not known which two inputs will be consumed, so they are all unreliable and marked red, and therefore so is the input at s)

TyPiCal examples

$$\bar{t_1}.!\,a(x).\bar{u_1}.\bar{x} \mid \bar{t_2}.!\,a(x).\bar{u_2}.\bar{x} \mid \overline{a}\langle s\rangle \mid s \mid !\,t_1 \mid !\,u_1 \mid !\,u_2$$
$$\rightsquigarrow \quad \bar{t_1}.!\,a(x).\bar{u_1}.\bar{x} \mid \bar{t_2}.!\,a(x).\bar{u_2}.\bar{x} \mid \overline{a}\langle s\rangle \mid s \mid !\,t_1 \mid !\,u_1 \mid !\,u_2$$

Having *either* $!\,t_1$ or $!\,t_2$ is enough for $\overline{a}\langle s\rangle$ to succeed. We however need *both* $!\,u_1$ and $!\,u_2$ for s to succeed:

$$\bar{t_1}.!\,a(x).\bar{u_1}.\bar{x} \mid \bar{t_2}.!\,a(x).\bar{u_2}.\bar{x} \mid \overline{a}\langle s\rangle \mid s \mid !\,t_1 \mid !\,u_1$$
$$\rightsquigarrow \quad \bar{t_1}.!\,a(x).\bar{u_1}.\bar{x} \mid \bar{t_2}.!\,a(x).\bar{u_2}.\bar{x} \mid \overline{a}\langle s\rangle \mid s \mid !\,t_1 \mid !\,u_1$$

(Composing that process with $!\,t_2$ permits $\overline{a}\langle s\rangle$ to be caught by the second input, making it unreliable without a $!\,u_2$)

TyPiCal counter-examples

$$!\, a(x).\overline{x}\langle a\rangle$$

$\rightsquigarrow$ `The process is ill-typed.`

(This process requires a recursive channel type such as $(\mu\sigma.\,\mathrm{chan}(\mathrm{chan}(\sigma)))$, which TyPiCal doesn't handle.)

$$!\, s(c).c(a).c(b).\bar{a}.\bar{b} \mid (\nu c)\,\overline{s}\langle c\rangle.\overline{c}\langle a\rangle.\overline{c}\langle b\rangle.a.b \qquad (1)$$

$\rightsquigarrow$ $!\, s(c).c(a).c(b).\bar{a}.\bar{b} \mid (\nu c)\,\overline{s}\langle c\rangle.\overline{c}\langle a\rangle.\overline{c}\langle b\rangle.a.b$

(An encoding of $!\, s(a,b).\bar{a}.\bar{b} \mid \overline{s}\langle a,b\rangle.a.b$ into monadic π. Note that TyPiCal incorrectly marks the communication on the parameters as unreliable)

$$(\nu t)\left(\overline{t} \;\middle|\; t.(!\, z | !\, a(x).\bar{z}.\bar{x}) \;\middle|\; t.!\, a(y).\bar{y}\right)$$

$\rightsquigarrow$ $(\nu t)\left(\overline{t} \;\middle|\; t.(!\, z | !\, a(x).\bar{z}.\bar{x}) \;\middle|\; t.!\, a(y).\bar{y}\right)$

(Randomly picks a "slow" or a "fast" a-input. Note that the z-interaction in the "slow" one is incorrectly marked as unreliable)

Outline

1 Motivation

2 TyPiCal

3 Receptiveness, Responsiveness, Termination

4 Our Work

Uniform Receptiveness (Sangiorgi) [San99]

- Partitions the set of possible names into *linear receptive*, ω-receptive and *plain* names.
- Linear receptiveness and ω-receptiveness correspond respectively to *strong* activeness and strong uniform ω-activeness, always on input ports.
- The type system makes sure linear names are used once for input and once for output and ω names are used once
- Monadic π-calculus, but with sums and matching
- Defines "receptiveness-aware" bisimulation relations.

The Receptive Distributed π-calculus (Amadio et al.) [ABL03]

- Works on *distributed* π-calculus
- Communication not permitted across sites, so deadlocks may occur between two strongly active complementary ports if they are at different sites. (Their type system provides safety against this).
- Types *non-uniform* ω-activeness.
- Does not provide an equivalent to responsiveness, being only interested in *input* activeness.

Responsiveness (Acciai, Boreale) [AB07]

- Works on monadic asynchronous π-calculus.
- Their "responsiveness" corresponds to our activeness, *not* our responsiveness.
- Describes two independent type systems covered in the next two slides.

First Type System (Acciai, Boreale)

- A dependency network checks strong linear activeness or strong ω-activeness on input ports, and activeness for linear output ports. For a process like $\overline{b} \mid b.\overline{a}$, a dependency $a \longrightarrow b$ indicates the order in which linear channels are consumed.
- *Levels* are used much like parameter obligation levels in TyPiCal for checking delegation, in that $!\, a(x).\overline{b}\langle x\rangle$ requires b's level to be smaller than a's.
- Types recursive functions like $!\, f(n, r).\, \mathrm{if}(n = 0)\, \overline{r}\langle 1\rangle$ else $(\nu r')\, (\overline{f}\langle n - 1, r'\rangle \mid r'(m).\overline{r}\langle n * m\rangle)$
- Rejects guarded inputs and "half-linear names" like t in $(\nu t)\, (\overline{t} \mid t.P \mid t.Q)$

Second Type System (Acciai, Boreale)

- Introduces *capabilities* — not related to capability levels, but similar in idea to TyPiCal's channel *usages*.
- Allows guarded inputs, the "half-linear names" mentioned previously and replicated outputs, but rejects some recursive functions such as the "factorial" one given previously.

Termination (Deng, Sangiorgi) [DS06]

This paper gives type systems for termination. The first type system it provides stratifies names into *levels*.
The level of a (replicated) server:

- Indicates the maximal delegation depth
- If it is finite, then the server input port terminates

Example:

- $P = !\, a(x).(\overline{a}\langle x\rangle + \overline{b}\langle x\rangle) \mid !\, b(x).\overline{c}\langle x\rangle \mid !\, c(x).\overline{x}\langle 1\rangle$

a: level ∞, b: level 1, c: level 0
The paper then describes three refinements of the type system to allow for some forms of recursion, (much like [AB07]), and do a special treatment of input sequences, to type even more processes.

Outline

1 Motivation

2 TyPiCal

3 Receptiveness, Responsiveness, Termination

4 Our Work

Our Work

- Handles activeness and responsiveness separately
- Designed for polyadicity
- Uses accurate channel types, separately specifying the behaviour of a channel's input and output.
- Handles recursive channel types
- Working in an open, labelled setting; The type system takes in input what the process' environment is permitted to do, and outputs a description of how the program's behaviour changes depending on environment interference.

Dependency Network

- Instead of calculating levels: dependency statements.
- Actually the way TyPiCal works internally.
- Informally, α depends on β in P if composing P with a process Q that provides β gives a process $P|Q$ that provides α.

Example: What are $\bar{s}$'s (activeness-) dependencies in that process?

$$\left(\bar{t}.a \mid \bar{u}.a \mid \bar{v}.\bar{a}.\bar{w}.\bar{s}\right) \mid \left(u \mid w\right) \tag{2}$$

- $\bar{s}$ depends on (activeness for) v, a and w
- a depends on any one of t or u
- and u, w are provided on the right hand side
- Therefore $\bar{s}$ only depends on v

Dependency Network

Grammar

$$
\begin{array}{lcl}
\delta & ::= & \varepsilon\gamma \text{ — Dependency Statements} \\
\alpha, \beta, \gamma & ::= & p_{\mathbf{A}} \mid p_{\mathbf{R}} \text{ — } \mathbf{A}\text{ctiveness, } \mathbf{R}\text{esponsiveness Resources} \\
\varepsilon & ::= & \alpha < \mid \beta \leq \mid (\varepsilon|\varepsilon) \\
 & & \mid \varepsilon\&\varepsilon \mid \neg \mid \emptyset \text{ — Dependencies}
\end{array}
$$

- $\alpha < \beta$ and $\alpha \leq \beta$ are strong and weak dependencies, respectively (a resource is allowed to depend on itself only through weak dependencies, which occur when computing responsiveness dependencies).
- $\varepsilon_1|\varepsilon_2$ is satisfied if either of the ε_i is.
- $\varepsilon_1\&\varepsilon_2$ is satisfied if both ε_i are.
- $\neg\gamma$ means γ is never satisfied.
- $\emptyset\gamma$ means γ is satisfied without dependencies.

Dependency Network Examples

We omit the **A** index for now, focusing on activeness dependencies.

- In $a.b$, $\bar{a} < b$.

$$t.u.a \tag{3}$$

- In (3), $(\bar{t} <)\&(\bar{u} <)a$, also written $(\bar{t}\&\bar{u}) < a$ or even $\bar{t}\bar{u} < a$.
- In $t.a \mid u.a$, $(\bar{t} <)|(\bar{u} <)a$, also written $(\bar{t}|\bar{u}) < a$.
- In $\bot.a$, $\neg a$.
- In (2), $\left(\bar{t}.a \mid \bar{u}.a \mid \bar{v}.\bar{a}.\bar{w}.\bar{s}\right) \mid \left(u \mid w\right)$:

$$\begin{aligned} & ((t|u) < a) \odot (v\&a\&w < \bar{s}) \odot (u, w) \\ = \quad & (v\&(t|u)\&w < \bar{s}) \odot (u, w) \\ = \quad & (v < \bar{s}) \end{aligned}$$

($\odot$ stands for dependency network composition)

- In $\bar{a}.b \mid \bar{b}.a$, $a < b$ and $b < a$ compose into $a < a$ and $b < b$, equivalent to $\neg a$ and $\neg b$.

Protocols

- Example servers for two parameter channels

$S_1 = !\, a(x,y).(\bar{x}|\bar{y})$, $S_2 = !\, a(x,y).\bar{x}.\bar{y}$, $S_3 = !\, a(x,y).\bar{y}.\bar{x}$.

- Example clients for two parameter channels

$C_1 = \bar{a}\langle b,c\rangle.(b|c)$, $C_2 = \bar{a}\langle b,c\rangle.b.c$, $C_3 = \bar{a}\langle b,c\rangle.c.b$.

- Which $S_i \mid C_j$ give a deadlock?

$S_1 \mid C_1$, $S_1 \mid C_2$, $S_1 \mid C_3$, $S_2 \mid C_1$, $S_2 \mid C_2$, $S_3 \mid C_1$, and $S_3 \mid C_3$ ok.
$S_2 \mid C_3$ and $S_3 \mid C_2$ create a deadlock!

- Solution: Parameter Protocols

Protocols

$$S_2 \mid C_3 = !\, a(x,y).\bar{x}.\bar{y} \mid \bar{a}\langle b,c\rangle.c.b \xrightarrow{\tau} \bar{b}.\bar{c} \mid c.b$$

- It is unclear if it is the server or the client that should be changed to fix the deadlock.
- Provide a "protocol"; check who violates it.

Definition (Protocol)

A *protocol* is a pair $(\xi_I; \xi_O)$ where each ξ_p is a set of dependency statements on parameter resources n or $\bar{n}$ (I=Input, O=Output; $n \geq 1$ is a parameter number).

A communication party is allowed to have a parameter depend on another if combining it with the protocol doesn't create a circularity.

Protocol Examples

- $(1 < \bar{2}; \bar{1} < 2)$, the *left to right protocol*, rejects C_3 because C_3 contains $c.b$ which entails $\bar{c} < b$, which, combined with $(1 < \bar{2})\{^{bc}/_{12}\} = (b < \bar{c})$, creates a loop.
- $(2 < \bar{1}; \bar{2} < 1)$, the *right to left protocol*, similarly rejects S_2.
- $(1 \leq \bar{2}, 2 \leq \bar{1}; \bar{1} \leq 2, \bar{2} \leq 1)$, the *empty protocol*, rejects both because any dependency on the server or the client creates a loop when composed with the protocol. Only the dependency-less system $S_1 \mid C_1$ is accepted.

Responsiveness Dependencies

The *responsiveness dependencies* is the set of all parameter dependencies not given by the protocol.
Examples:

- In $\bar{t}.!\,a(x).\bar{u}.\bar{x}$, $t_{\mathbf{A}} < a_{\mathbf{A}}$ and $u_{\mathbf{A}} < a_{\mathbf{R}}$.
- In $!\,a(x).\overline{b}\langle x\rangle$, $b_{\mathbf{A}} \& b_{\mathbf{R}} < b_{\mathbf{R}}$.
- In $a(x).\bar{t}_1.\bar{x} \mid a(x).\bar{t}_2.\bar{x}$, $t_1 \& t_2 < a_{\mathbf{R}}$ (both are required for responsiveness because if a request is sent it is not known which input will receive it).
- More generally: if $\varepsilon_i a_{\mathbf{R}}$ holds in P_i then $(\varepsilon_1 \& \varepsilon_2) p_{\mathbf{R}}$ holds in $P_1 \mid P_2$.
- Assuming IO-alternation, in $\bar{a}\langle x\rangle$, $x_{\mathbf{A}} \& x_{\mathbf{R}} \leq \bar{a}_{\mathbf{R}}$, $a_{\mathbf{A}} \& a_{\mathbf{R}} < \bar{x}_{\mathbf{A}}$ and $a_{\mathbf{A}} \& a_{\mathbf{R}} < \bar{x}_{\mathbf{R}}$.

Multiplicities

For simplicity, we don't handle arbitrary usages, but merely *channel multiplicities*:

Definition (Channel Multiplicities)

The *multiplicities* of a channel is a pair $(m_I; m_O)$ where both $m_p \in \{0, 1, \bigstar, \omega\}$, standing respectively for inaction, linearity, lack of constraints and uniform replication

We express that information as $a^m, \bar{a}^{m'}$, which is equivalent, in term of usages, to $a : \mathrm{chan}_{!^m|?^{m'}}(\tilde{\sigma})$, with $u^0 = 0$, $u^1 = u$, $u^\bigstar = \mu\rho.(u.\rho \,\&\, 0)$ and $u^\omega = \mu\rho.(u.\rho)$ (for both $u \in \{!, ?\}$).

Labelled Dependencies

For a port p, responsiveness ($p_{\mathbf{R}}$) dependencies generally do not include activeness ($p_{\mathbf{A}}$) dependencies.

$$P = (\nu t)\left(\bar{t} \,\middle|\, t.(!\, z | !\, a(x).\bar{z}.\bar{x}) \,\middle|\, t.!\, a(y).\bar{y}\right) \tag{4}$$

The leftmost a-input gives $z_{\mathbf{A}} < a_{\mathbf{R}}$, the z-input gives $\bar{t}_{\mathbf{A}} < z_{\mathbf{A}}$, but we have $\neg\bar{t}_{\mathbf{A}}$, as the t-output isn't reliable. Yet, a is responsive!
Solution: Labelled dependencies.

Labelled Dependencies

- *Labels* l are objects belonging to some infinite set, disjoint from other objects considered so far.
- $l : \varepsilon$: Labels part of a dependency with a unique label l.
- $\neg l : \varepsilon$: This dependency may be ignored if used from within an l-labelled region

Labelled Dependencies

The type system uses labelled dependencies as follows.

- Every responsiveness dependency statement receives its own fresh label ($l : \varepsilon p_{\mathbf{R}}$).
- When typing $\overline{a}\langle\tilde{v}\rangle.P$ or $a(\tilde{x}).P$, the $(\neg\tilde{l} : p_{\mathbf{A}} <)$ dependency ($p = \bar{a}$ or $p = a$) is added to all activeness dependencies, with $\tilde{l}$ being the set of all responsiveness labels in use in P.

Then, activeness resources get the extra $p_{\mathbf{A}}$ dependency, but responsiveness resources don't.

Labelled Dependency Example

Viewing the example again:
$P = (\nu t)\left(\bar{t} \,\middle|\, t.(!\, z | !\, a(x).\bar{z}.\bar{x}) \,\middle|\, t.!\, a(y).\bar{y}\right)$

- The definition for responsiveness gives $(l : \bar{x}_{\mathbf{A}} \& r : \bar{y}_{\mathbf{A}}) \leq a_{\mathbf{R}}$.
- For simplicity we ignore the bindings inherent to input prefixes, and then $\bar{x}$ and $\bar{y}$ have the following activeness dependencies: $(z_{\mathbf{A}} \& \neg l : (\bar{t}_{\mathbf{A}} \& \bar{a}_{\mathbf{A}})) < \bar{x}_{\mathbf{A}}$ and $(\neg r : (\bar{t}_{\mathbf{A}} \& \bar{a}_{\mathbf{A}})) < \bar{y}_{\mathbf{A}}$.
- Substituting in $a_{\mathbf{R}}$'s dependencies:
  $l : (\neg l : (\bar{t}_{\mathbf{A}} \& \bar{a}_{\mathbf{A}})) \& z_{\mathbf{A}}$ & $r : (\neg r : (\bar{t}_{\mathbf{A}} \& \bar{a}_{\mathbf{A}})) < a_{\mathbf{R}}$
- Dropping the $(r : \neg r : \dots)$ and $(l : \neg l : \dots)$ parts: $l : z_{\mathbf{A}} < a_{\mathbf{R}}$
- Resource $z_{\mathbf{A}}$ depends on $(\neg l : \bar{t}_{\mathbf{A}})$, so we get $l : (\neg l : \bar{t}) < a_{\mathbf{R}}$
- As $(l : \neg l : \dots)$ is equivalent to $\emptyset$ we get $\emptyset a_{\mathbf{R}}$.

Application (Recursive Channel Types)

This example justifies the use of processes like $!\, a(x).\overline{x}\langle a\rangle$, with an encoding of sequential-style programming.

- Program Counter pc: one of $\{a_1, \ldots, a_n\}$
- Memory state m: Data that can be passed along a channel
- Program Line: $!\, a_i(m, out, next).\overline{out}\langle \ldots \rangle.\overline{next}\langle a_j, m' \rangle$

Executing a program:
$!\, run(m, out, pc).\overline{pc}\langle m, out, next\rangle.next(pc', m').\overline{run}\langle pc', out, m'\rangle$

```
for (;;) print ‘‘hello world!’’;
```

$$!\, a(m, t, x).(\overline{t}\langle \texttt{"hello world!"} \rangle \mid \overline{x}\langle a, m\rangle) \tag{5}$$

Port a is responsive and, for some σ_m, has a recursive type $\mu\sigma.\,\mathrm{chan}(\sigma_m, \mathrm{chan}(Str), \mathrm{chan}(\sigma, \sigma_m))$.

Expressiveness

- We now compare the expressiveness of our work to that of other papers we covered, in terms of sets of correctly handled processes.
- Note that our type system is not concerned about termination, and therefore it would make no sense to compare it with termination type systems (as shown before, termination neither implies nor is implied by activeness or responsiveness).

TyPiCal's lock-freedom analysis

- Not comparable. Although their obligation and capability levels can be directly mapped to dependency networks, our work doesn't handle arbitrary channel usages. For instance, in $(\nu a)\,(a \mid a \mid \bar{a} \mid \bar{a}.\bar{s})$, it classifies a as plain ($a^{\bigstar}\bar{a}^{\bigstar}$) and unreliable, and therefore marks $\bar{s}$ as non-active.
- On the other hand, our system's handling of labelled dependencies (4) and recursive channel types (5) makes our system accept processes rejected by TyPiCal.

Acciai, Boreale [AB07]

- Their first type system is subsumed by ours (their levels and dependency graphs can respectively be translated into responsiveness and activeness dependencies, and semi-linear names (2) or guarded inputs (3) well-handled by our system are rejected by theirs)
- Note however that our system does not recognise as responsive any form of recursivity. We believe however that their way of handling it could easily be adapted to our system: An output call $\overline{b}\langle\tilde{v}\rangle$ found in a server $!\, a(\tilde{x})$ should create a *weak* responsiveness-dependency ("$b_{\mathbf{R}} \leq a_{\mathbf{R}}$") if $\tilde{v}$'s "weight" is smaller than $\tilde{x}$'s.
- We answer affirmatively their question about generalised dependency graphs (see Remark 1 (1) in their paper, as well as Section 6.2)

Acciai, Boreale's second system

We conjecture that our system subsumes their second type system as well.

- They introduce "capabilities", which closely correspond to our multiplicities, so that we can translate their types into ours. Having translated the types, typability in their system implies typability in ours.
- On the other hand they have a number of restrictions absent from our system. E.g. "+-responsive" names carrying "responsive" names must have an unguarded replicated input, so that $a(x).!\, b(y).\overline{y}\langle x\rangle$ (all names bilinear active, or "responsive") would be rejected[2].

[2] Note that a generalisation of their system, not having this limitation, is to be published soon.

Strong activeness type systems [ABL03, San99]

- We believe that typability in these systems, when only considering non-distributed processes without sums, implies typability in our system.
- However this is not particularly significant because we are interested in ("weak") activeness, so that for instance our type system recognises r as active in $(\bar{a}|a.r)$, which is rightfully rejected by their type systems, as r is not strong active.

Future Work

- Better understand, and formally prove the relationship between our work and those mentioned in this presentation.
- Prove type soundness
- Possible type system extensions: Usages, recursion, sums, choice types, ...
- Prove the equivalence of TyCo and the π-calculus, as an application of this type system.
- Write papers ...

References I


L. Acciai and M. Boreale.
Responsiveness in Process Calculi.
In M. Okada and I. Sato, eds, *Proc. of 11th Annual Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *Lecture Notes in Computer Science*, pages 136–150. Springer-Verlag, 2007.


R. M. Amadio, G. Boudol and C. Lhoussaine.
The receptive distributed π-calculus.
ACM Trans. Program. Lang. Syst., 25(5):549–577, 2003.


Y. Deng and D. Sangiorgi.
Ensuring termination by typability.
Information and Computation, 204(7):1045–1082, 2006.

References II

D. Sangiorgi.
The Name Discipline of Uniform Receptiveness.
Theoretical Computer Science, 221(1–2):457–493, 1999.
An abstract appeared in the *Proceedings of ICALP '97*, LNCS 1256, pages 303–313.