

Feasible reactivity in a synchronous π -calculus

Roberto AMADIO

Université de Paris 7

Laboratoire Preuves, Programmes et Systèmes

Joint work with Frédéric DABROWSKI

Outline

Develop **static analysis** methodology for **resource control** for a language of **threads** executing **synchronously** and interacting via **signals**.

Plan

- Resource control (in first-order functional languages).
- Synchronous π -calculus.
- Feasible reactivity in the synchronous π -calculus.

Data-size flow analysis

- A program transforms data in input into data in output.
- A data-size flow analysis bounds the size of the output as a function of the size of the input.
- In a nutshell:

Resource Control = Termination + Data-size flow analysis

Example: an algorithm to put 0's before 1's

$t = \epsilon \mid 0 \text{ of } t \mid 1 \text{ of } t$ (*binary words*)

$$\begin{array}{llll} \mathit{sort} = \epsilon & \Rightarrow \epsilon & \mathit{ins} = \epsilon & \Rightarrow 1(\epsilon) \\ 0(x) & \Rightarrow 0(\mathit{sort}(x)) & 0(x) & \Rightarrow 0(\mathit{ins}(x)) \\ 1(x) & \Rightarrow \mathit{ins}(\mathit{sort}(x)) & 1(x) & \Rightarrow 1(1(x)) \end{array}$$

Data-size analysis by **quasi-interpretation**, with:

$$q_\epsilon = 0, \quad q_0 = q_1 = q_{\mathit{ins}} = \lambda x.x + 1, \quad q_{\mathit{sort}} = \lambda x.x$$

Termination analysis by **RPO**, with: $\mathit{sort} > \mathit{ins}$.

Another termination analysis by **SCT**, using the quasi-interpretation.

A typical result [Bonfante, Marion, Moyen 2001]

If the program terminates by **lexicographic ordering** (**product ordering**) and the **quasi-interpretation is polynomially bounded** then the functions computed by the program are in **PSPACE** (**PTIME**).

NB Earlier approaches: Cook, Bellantoni-Cook, Leivant,...

Synchronous model

- In synchronous models, the computation is regulated by a notion of **instant** (or phase, or pulse, or round).
- At each instant each **thread** performs some actions and synchronizes with all the other threads.
- Threads interact through **signals**. An emitted signal persists within the instant and it is reset at the end of it.
- **Instant ends** when all threads are either terminated or suspended waiting for a signal that cannot come.

From ESTEREL to SL to S- π

- In ESTEREL [Berry-Gonthier 92], a thread reacts immediately both to the presence and to the absence of a signal.
- In SL [Boussinot-De Simone 96], reaction to the absence is only possible at the end of the instant.
- In S- π [A. 06], signals carry values including signal names.

NB The S- π model is an attempt at formalising –at a process calculus level– various embeddings of the SL model in C, Java, Scheme, CAML due to Boussinot, Serrano, Mandel-Pouzet.

π vs **S**- π

Assume $v_1 \neq v_2$ are two distinct values and

$$P = \nu s_1, s_2 (\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid \\ s_1(x). (s_1(y). (s_2(z). A(x, y) , B(!s_1)) \\ , 0) \\ , 0)$$

P is a π -calculus process if we forget about the **else branches of the read instructions.**

Spot the differences...

$$P = \nu s_1, s_2 (\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid s_1(x). (s_1(y). (s_2(z). A(x, y), B(!s_1)), 0), 0)$$

- In π , P reduces to

$$P_1 = \nu s_1, s_2 s_2(z).A(\sigma(x), \sigma(y))$$

where $\sigma(x), \sigma(y) \in \{v_1, v_2\}$ and $\sigma(x) \neq \sigma(y)$.

- In $S\pi$, **signals persist within the instant** and P reduces to

$$P_2 = \nu s_1, s_2 (\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid (s_2(z).A(\sigma(x), \sigma(y)), B(!s_1)))$$

where $\sigma(x), \sigma(y) \in \{v_1, v_2\}$.

- In π , P_1 is now **deadlocked**.
- In $S\pi$, the **current instant ends** and we move to the following one

$$P_2 \mapsto P'_2 = \nu s_1, s_2 \mathbf{B}(\ell)$$

where $\ell \in \{[v_1; v_2], [v_2; v_1]\}$.

- Thus at the end of the instant, $!s_1$ becomes *a list of (distinct) values* emitted on s_1 during the instant.

Another example: a server

Let $\text{pause}.K$ be $\nu s s.0, K$.

$$\text{Server}(s) = \text{pause}.\text{Handle}(s, !s)$$

$$\text{Handle}(s, \ell) = [\ell \triangleright \text{req}(s', x) :: \ell'] (\overline{s'} f(x) \mid \text{Handle}(s, \ell')), \text{Server}(s)$$

A ‘server’ handling a list of requests of the shape $\text{req}(s', x)$ emitted in the previous instant on the signal s . The answer which is a function of x goes along the signal s' .

Feasible reactivity

- A **computation** of a program P looks like this:

$$P_1 \xrightarrow{*} P'_1 \xrightarrow{I_1} P_2 \xrightarrow{*} P'_2 \xrightarrow{I_2} P_3 \xrightarrow{*} \dots$$

where I_i are the inputs at instant i .

- P is **reactive** if in all computations that start with P , we reach the end of the instant *infinitely often*.
- P is **feasibly reactive** if there is a **polynomial** Q such that for every computation, if d bounds the size of P and the sizes of the largest input *up to instant* k then the program at instant k reaches the end of the instant in *time* $Q(d)$.

NB Reaction time depends on size of largest input but *not* on the number of instants.

Restrictions and annotations

- We impose (standard) syntactic restrictions that bound the number of running threads.
- We observe that in typical applications, each thread operates in a **cyclic** fashion. A cycle may span several instants (possibly an unbounded number).
- For each thread, the programmer indicates when a new cycle begins (*e.g.*, *Server* in the previous example).

Inequalities generation

Index 0 Guarantee that the computation of each thread within an instant **terminates** in polynomial time in the size of the parameters and the inputs.

Index 1 Guarantee that at the beginning of a cycle a thread is **non-size increasing**.

Technically, we generate term inequalities and look for a polynomially bounded quasi-interpretation that satisfies them. However, this is *not* enough!

An annoying example

$$A(s) = \text{pause}.B(s, !s)$$

$$B(s, \ell) = [\ell \triangleright n :: \ell'] (\bar{s}n \mid B(s, \ell')), A(s)$$

$$C(s) = \nu n \bar{s}n \mid \text{pause}.C(s)$$

- At each instant, the thread A re-emits on s the values that were emitted on s in the previous instant while the thread C emits a (fresh) value on s .
- The system satisfies the constraints of index 0 and 1, however it is *not* feasibly reactive as the set of values on signal s grows by one at each instant.

Region stratification

- We assume that signals are partitioned into a (finite) number of ordered **regions**.
- A signal type comes with an **annotation** specifying the region to which it belongs.
- The size of data written into a region must be polynomial in (the parameters and) the size of the data read in *smaller* regions (this is expressed by generating **index 2** inequalities).

Main result

A **quasi-interpretation** is a collection of numerical functions (with the usual properties) that satisfies the inequalities of index 0, 1, 2.

Theorem A program that admits a polynomially bounded quasi-interpretation is feasibly reactive.

NB In practice, inferring quasi-interpretations is easier than inferring interpretations for termination: *small* polynomials suffice and *inequalities* must not be strict.

Proof sketch

Index 0 A thread suspends in time polynomial in the size of the parameters at the beginning of the instant and the inputs.

Index 1 The size of a thread at the beginning of a cycle is non-size increasing.

Index 2, rank 0 The size of what is emitted and of the parameters is polynomial in the size of the parameters at the beginning of the cycle.

Index 2, rank $n+1$ The size of what is emitted and of the parameters is polynomial in the size of the parameters at the beginning of the cycle and the inputs on regions of smaller rank.

Conclusion

- A few key ingredients:
 - Polynomial time **termination** of loops within an instant.
 - An invariant that guarantees data are **non-size increasing** at the beginning of a cycle.
 - Another invariant, built on the **stratification of signals** in regions.
- This is a first result extending resource control techniques from functional to concurrent (synchronous) programs.
- More theoretical and practical work is needed.

Server continued: reset points and auxiliary variables

$$\underline{Server}(s) = \text{pause}.Handle(s, !^y s)$$

$$Handle(s, \ell) = [\ell \triangleright \text{req}(s', x) :: \ell'] (\overline{s'} f(x) \mid Handle(s, \ell')), Server(s)$$

- $Server$ marks the beginning of a cycle (an annotation).
- $Server^+(s, y)$ depends both on its parameter and the value y read within a cycle.
- $Handle^+(s, \ell)$ depends only on its parameters since no value is read within the cycle.

Server continued: masking and inequalities generation

$$\underline{Server}(s) = \text{pause}.Handle(s, !^y s)$$

$$Handle(s, \ell) = [\ell \geq \text{req}(s', x) :: \ell'] (\overline{s'} f(x) \mid Handle(s, \ell')), Server(s)$$

$$Handle^+(s, \text{req}(s', x) :: \ell') >_0 Handle^+(s, \ell')$$

$$Server^+(s, 0) \geq_1 Handle^+(s, 0),$$

$$Handle^+(s, 0) \geq_1 Handle^+(s, 0),$$

$$Handle^+(s, 0) \geq_1 Server^+(s, 0) .$$

NB In *Handle* we *mask* the second parameter (another annotation).

Server continued: regions stratification

The *Server* is accepted provided the region of s' is **higher** than the region of s .

$$\text{Server}(s) = \text{pause.Handle}(s, !s)$$

$$\text{Handle}(s, \ell) = [\ell \succeq \text{req}(s', x) :: \ell'] (\overline{s'} f(x) \mid \text{Handle}(s, \ell')), \text{Server}(s)$$

The constraints maintain the **invariant** that $f(x)$ is polynomially bounded in the parameters at the beginning of a cycle and what is read in regions smaller than ρ .

$$\text{Server}^+(s, y) \geq_2 \text{Handle}^+(s, y)$$

$$\text{Handle}^+(s, \text{req}(s', x) :: \ell') \geq_2 \text{Handle}^+(s, \ell'),$$

$$\text{Handle}^+(s, \text{req}(s', x) :: \ell') \geq_2 f(x) .$$

Why the annoying example is rejected

$$A(s) = \text{pause}.B(s, !s)$$

$$B(s, \ell) = [\ell \succeq n :: \ell'] (\bar{s}n \mid B(s, \ell')), A(s)$$

$$C(s) = \nu n \bar{s}n \mid \text{pause}.C(s)$$

The index 2 inequalities include

$$A^+(s, 0) \geq_2 B^+(s, y)$$

which cannot be satisfied.