What is Wrong with Process Calculi – And How to Recover?

Hubert Garavel
Inria Grenoble – LIG
Université Grenoble Alpes

http://convecs.inria.fr
Glory and misery of process calculi
**Achievements of process calculi**

- A **fruitful theory** for modeling concurrent systems
  - the proper way of expressing concurrency
  - early detection of design mistakes
- **Famous calculi**: CSP, CCS, ACP...
- **ISO standards**: LOTOS, E-LOTOS
- **Turing awards**: Hoare, Milner
- **Robust tools**: CADP, FDR, mCRL2, PAT...
  - with many successes on industrial case studies
- **Conferences**: CONCUR, EXPRESS/SOS ...
- **Process algebra handbook** (1342 pages)
But a shrinking audience...

- No longer a research priority for funding agencies
- Fewer industrial users:
  - industry still has many problems with concurrency
  - but concurrency theory is not seen as THE solution
- Fewer students:
  - no clear demand for learning concurrency theory
  - difficult to create (or even maintain) such courses
- Negative feedback loop:
  - fewer students ⇒ fewer tools ⇒ fewer aplications ⇒ ...
- Concurrency experts are progressively retiring
A declining influence (1/2)

- **Java (1995)**
  - parallelism based on shared variables and locks
  - no formal semantics – Java memory model issues
  - back in time to the 1970s (pre-Hoarian era)

- **UML (1997)**
  - concurrent state machines with a graphical syntax
  - no formal semantics – incompatible views

- **DSMLs (Domain-Specific Modelling Languages)**
  - XML-based syntax
  - semantics in natural language (with OCL constraints)
A declining influence (2/2)

- **Ocaml 5 (2023)**
  - formerly, JoCaml (2014) was based on the join-calculus
  - instead, Ocaml 5 brings shared-memory concurrency

- A modern **Cassandra complex**:
  - we know everything about concurrency, in full detail
  - but no one pays attention to our opinion
A few sharp statements

"Process algebra has lost the battle!"
Moshe Vardi (May 2020)

"Almost no one uses process calculi anymore these days."
Joost-Pieter Katoen (April 2023)
Why such a decline?
Many reasons, in combination

- Concurrency theory is inherently difficult
  - but we make it more obscure (Greek letters...)
- Concurrency theory is intrinsically diverse
  - but we encourage artificial proliferation
  - do we need hundreds of bisimulations?
  - do we need a different formalism in each university?
- Outsiders cannot distinguish key ideas from details
- Lack of critical mass, insufficient tool support
- Few solutions directly usable by practitioners
Error #1: Over-emphasis on "calculi"

- CSP (1978) was a programming language
- CCS (1980) was a "calculus"
  - elegant definition, with a syntax that fits on one line
  - but too simple for practical needs
  - few realistic systems have been modelled using CCS
- "calculi" ≠ "languages"
  - calculi focus on semantics, and ignore anything else
  - calculi must be extended, often in incompatible ways
  - they do not support good engineering practices
  - they do not care about developer productivity
Error #2: Purely functional style

- Originally, CSP (1978) was an imperative language.
- But CCS (TCSP, LOTOS...) chose a functional style.

**PRO:**
- CCS's formal semantics was state-of-the-art at its time.

**CONS:**
- No loop operator, only recursive processes.
- No mutable variables, only parameters.
- Parameter lists may become long and error-prone.
- Imperative style combined with static analysis is as safe as functional style, and much more flexible.
Error #3: Algebraic style (1/3)

■ Trend to use algebra everywhere:
  ▶ 1) for data types and functions: LOTOS, PSF, μCRL, etc.
  ▶ 2) for processes: PSF, μCRL, mCRL2

■ PRO 1 (for data types and functions):
  ▶ abstract data types were fashionable in the 80s
  ▶ formal semantics, independent from implementations
  ▶ evaluation of expressions is free from side effects

■ CONS 1:
  ▶ completeness and confluence (nondeterminism) issues
  ▶ no proper modelling of exceptions
  ▶ "ADTs really killed LOTOS."  Juan Quemada (E-LOTOS editor)
Error #3: Algebraic style (2/3)

- **PRO 2** (for processes):
  - appealing (?) analogy with arithmetics: 0, 1, +, .
  - a few intuitive axioms: commutativity, associativity...
  - binary sequential composition (>> CCS's action prefix)

- **CONS 2**:
  - poorly readable
  - overloading: "+") means either addition or choice
  - LISP-like parentheses: "))))" mixing data and processes
  - insufficient expression of data flow, e.g.,
    \[
    \text{sum } x. (\text{RECV~}(x).\text{SEND~}(x)) \text{ instead of } \text{RECV~}?x; \text{SEND~}!x
    \]
Error #3: Algebraic style (3/3)

- Also:
  - software/hardware engineers are not mathematicians
    ⇒ algebra is not so appealing to them
  - algebraic specifications are harder to implement efficiently than, e.g., finite-state machines
  - algebraic laws (but congruence) do not help much in formal verification, done by state-space exploration

- All in one, algebra brings more problems than solutions
How to recover?
Back to the roots

What is really **essential** in process calculi?

1. An effective way to precisely **model** concurrency
2. **Message-passing** communication
3. **Action-based** semantics (transitions, not states)
4. Formal semantics given by **SOS rules**
5. Algebraic properties:
   - commutativity, associativity, etc. of operators
   - congruence of parallel composition for bisimulation (to fight state-space explosion)
Guidelines for a better language

- Stay away from calculi
  - a one-line language like CCS is not sufficient in real life

- Stay away from the fully functional style
  - mainstream programming languages are imperative
  - but functional traits (e.g. pattern matching) are ok

- Stay away from fully algebraic approaches
  - most programmers are not mathematicians
  - reuse the advances of structured programming

- Retrospectively, CSP-1978 was very well done
Global map of process calculi

Bristol track
- Occam1 (1983)
- Occam2 (1988)
- Occam3 (1992)

Oxford track
- CSP (1978)
- TCSP (1984)
- CSPm (1997)

Edinburgh track
- CCS (1981)
- π-calculus
- VPL (1997)
- bigraphs

ISO track
- LOTOS (1989)
- LOTOS NT (1997)

Grenoble track
- E-LOTOS (2001)

Amsterdam track
- ACP (1984)
- PSF (1989)
- μCRL (1995)
- mCRL2 (2006)
- LNT 1.0 (2006)
- LNT 7.2 (2023)
A few words on LNT

- **LNT**: language being developed at INRIA Grenoble
  - inspired from CSP-1978, Occam, and E-LOTOS
  - process calculus with imperative and functional traits
  - formal semantics given by SOS rules
  - strong typing and static analyses to detect mistakes
  - support for proofs: assertions, pre-/post-conditions

- Language primarily **designed for engineers**:
  - keep things as simple as possible
  - use notations as standard as possible (Ada-like syntax)
  - emphasize readability by non-experts
A few results about LNT

- **Tool chain for LNT:**
  - two compilers (LNT2LOTOS and TRAIAN) – 90,000 locs
  - 80% of these compilers written in LNT ("self-hosted")
  - LNT is both a specification and programming language
  - part of the CADP toolbox ([https://cadp.inria.fr](https://cadp.inria.fr))

- **On-going dissemination:**
  - engineering and master courses (easier than LOTOS!)
  - 28 published case studies done with LNT:
    - e.g. Google, Nokia, Orange, STMicroelectronics, Tiempo
  - 14 research tools generating LNT code
Conclusion
Concurrency theory today

■ The audience of concurrency theory is shrinking
  ➤ its valuable results might fade to oblivion

■ Time has come for encyclopedic synthesis:
  ➤ reexamine / select / simplify / sort
  ➤ tutorials needed ("Concurrency for the dummies")
  ➤ contributions to Wikipedia

■ Strengthen the links of concurrency theory with:
  ➤ industrial applications
  ➤ other branches of computer science
Process calculi have a future

- There are still industrial needs:
  - concurrent systems everywhere: hardware, software
  - safety, security, performance issues

- Other languages are not that good:
  - limited expressiveness/scalability, dubious semantics
  - absence of sound verification tools

- Merge process calculi with more general languages
  - extend the scope and applicability of process calculi
  - use them as target languages to implement DSMLs