Ports, Protocols, and Processes:
a Programming Paradigm?

Peter Grogono
Computer Science and Software Engineering
Concordia University
22 November 2007

1 It is time for a new paradigm. The talk explains why.

2 Our project is named for Erasmus. Why? A clue: Erasmus visited Queens' College, briefly in 1505 and for longer during 1511-14, where there is a building named after him. We can claim that Brian, Peter, and Desiderius are alumni of the same college although, strictly speaking, Erasmus is not an alumnus.

> It was provided by the statutes that lectures were to be given on sophistry, logic, philosophy, *and the works of poets and orators*. Mr. Mullinger has suggested that the latter clause was due to the influence of Erasmus, who had a very high opinion of Fisher and who attributed to him the peaceful introduction of the study of Greek in Cambridge and all that was most encouraging in the University.[1]

3 People. East of the Atlantic: Brian. West: Peter and grad students. Rana is actually east of the Atlantic.

4 Erasmus said many interesting things; this one is appropriate for our project. Perlis says essentially the same thing about LISP. As disciples of Hoare, we stole practically everything from someone else.

5 Outline: cover as much as possible in the time available, leaving time for questions.

6 The next few slides are about programming in Erasmus. All of the code compiles and runs, although it may not do much. *This is what we have done so far. May raise questions: answers later.* Diagrams suggest a design notation. Distinguish *declaration/definition* and *instantiation* of cell.

6a Add a protocol; diagram unchanged.

6b Add an empty server process and an empty client cell. Server *provides* (+) protocol; client *needs* (−) protocol.

7 We can start filling in some details. Protocol specifies start/queries/stop. Server code corresponds to protocol. Client is still empty. If executed, server waits for ever. "`p.`" indicates send/receive/synchronize.

8 Client process nested in client cell. Cells and processes can be nested indefinitely.

9 We could continue coding but will move on to generalities instead. `query`: caret gives direction. `sequence`: protocol fields have types; no type implies signal. `method1`: protocol imitating function call. `method2`: multiple results. `class`: protocol imitating class interface.

[1] *Lady Margaret: A Memoir*, E.M.G. Routh, OUP, 1924. Dr. John Fisher was the President of Queens' 1505–8.

10 `select` chooses from different ports and fields. Top: can distinguish different fields of a protocol. Middle: branches can be guarded, directions can be mixed. Bottom: policies determine processing order.

11 Ports and processes can be instantiated dynamically. The code shows part of Eratosthenes' sieve: the process `filter` receives a prime number p, recursively generates a new filter, and sends non-multiples of p to it.

12 The diagram for a recursive process such as `filter` is a bit messy ...

12a ... but we can do better. Ports on the edge of a cell/process provide external linkage; ports within allow the cell/process to communicate with its components.

13 Diagram for a client/server application in which both processes are embedded in cells.

14 Code for this application—details do not matter. Compiling this code gives a program that runs on a single processor and switches context between the two processes.

15 Give the compiler the *same code* and this metacode: the result is a program that runs on two processors, sending data over the network. Crucial for refactoring.

16 Outline: from specifics to generalities.

17 Cells: any size; first-class; control flow; co-routines.

18 Cells of any size gives *scalability.*

19 Processes are similar in some ways to cells, but: actions, shared variables, no race conditions.

20 Processes in a cell run as coroutines and may share variables: `P1` and `P2` share `V1`. `P3` is nested in a cell and is *not* run as a coroutine. `P3` can access `V2` but not `V1`. We are not *advocating* shared variables but (tentatively) allowing them.

21 Process loses control when it communicates. The two displayed values of `sv` may differ.

21a In particular, sending a shared variable is unpredictable. (See above: we are not advocating sharing!)

21b A process behaves *as if* it is atomic, except that communication is non-atomic (open). "Atomic" is with respect to shared variables: there is no other interference. We cannot eliminate problems associated with concurrency, but we can tame them.

22 Protocols are like specifications. They define the type and sequence of communications. Satisfaction is like subtyping—more later.

23 Messages behave like lvalues and rvalues. Code does not specify location of communicator. Signals synchronize.

23a Receives can be used in expressions—they really are lvalues.

24 "Separation of concern" is a useful principle in CS and SE. These separations are useful: cells/processes; meaning/deployment; specification/satisfaction—what a process does/that it does what is needed.

25 Outline: one question we are often asked is "how will you verify correctness?". Here is one possible approach.

26 Milner meets Erasmus.

27 Labelled Transition System (LTS) has states, labels, and transition relation and is a very common basis for reasoning about concurrency.

28 Strong simulation is a useful relation on LTS. Assume pSq and take any transition $p \xrightarrow{\alpha} p'$. Then there must be q' that enables completion of the square with $p'Sq'$ and $q \xrightarrow{\alpha} q'$.

29 There is a strong simulation in our original diagram: left can simulate right, but not *vice versa*. Conventional theory continues with *bisimulation* (effective equivalence of concurrent systems) but we don't.

30 Define *simulates* as a relation on sets of states. If B simulates A then (roughly) B can do anything that A can.

31 There is a simple algorithm that extracts an LTS from a protocol.

32 There is a simple algorithm that extracts an LTS from a process.

33 The compiler can derive LTSs for the server, client, and their protocols. The process and client may not declare the same protocols. The compiler can check the simulations statically and efficiently. We can check that the server *satisfies* the client. ("The customer is always satisfied.")

34 Another example from our original diagram. The server handles an unlimited sequence of queries; the client sends only two.

35 No time for details, but there is a simple and intuitive reasoning system for LTS: Hennessy-Milner Logic. A formula is a mapping from sets of states to sets of states. The logic is modal because of the quantifiers $\langle\cdot\rangle$ and $[\cdot]$.

36 The semantics show how formulas translate to state-set maps.

37 Summary: code/protocols map to LTS; simulation ensures that servers "satisfy" clients.

38 Outline: a few notes on software development/engineering.

39 We believe in modelling!

40 An overview of the software development process. Requirements are very broad—all possible applications, many approaches, not "one size fits all". Machine code is all the same, "one size fits all". (Of course, machine code is not the *only* product of development.) The "funnel" shows how very different things are squeezed into one thing. Point of diagram: process evolution tends to occur bottom-up.

41 Left shows language evolution; right shows development processes that they inspired. Note that mechanical translation is possible *upwards* but not *downwards*. E.g., OO program can be translated to SP program but not *vice versa*.

42 Conclusion: if you want to improve the process, start with a programming paradigm—although this is controversial!

43 Our hypothesis: process-oriented model-driven development will succeed because there are many forces pushing towards it. In particular, Erasmus, with cells and processes to reduce coupling, will be very successful.

44 Outline: why do we bother with this? Will anyone take any notice? (Present company excluded.)

45 Arguments against …

45a Many say that our PLs are good enough: the problem is the process. Not true: programmers make many mistakes, cannot (or are not allowed to) handle concurrency, software becomes "brittle", etc.

45b Put pressure on the anti-PL people and they respond with "objects" as the ultimate solution. But objects are becoming too complex—cf. Brian's talk.

45c Under more pressure, they admit that objects are not ultimate but perhaps aspects are. But aspects are part of the problem—they just make things more complex.

45d Current approaches split applications into the "hard bits" (concurrency, transactions, database access, etc) and the "easy bits" that can be coded by "ordinary programmers". Hard bits need gurus. Increasingly pervasive concurrency will doom this approach.

45e No one will buy into a new paradigm. POPL anecdote about UNIX. Think 10 years, or 20 years, ahead: will OOP/AOP still dominate? When do we start changing paradigms?

46 Arguments in favour …

46a More and more applications require concurrency

46b Multicore requires concurrent thinking

46c No time for a detailed argument, but we believe that "thinking with processes" is better than "thinking with objects/methods". It will take time to train a generation of programmers to think in this way, just as it took a generation to produce OO programmers. Processes are closer to the real world. Alan Kay said that an object abstracts a computer; we think a process abstracts a PU and a cell abstracts a computer.

46d We must be able to write programs that grow. This is difficult/impossible with current techniques. It will be easier with communicating processes and cells.

47 We are not the only people who feel this way. There is too much related work to discuss in a talk but we can mention several languages. Joyce, and other work by Per Brinch Hasnen, is a primary source of inspiration.

48 Adding threads and locks to the current mess won't work. Guru code is not a good long-term solution. Modular concurrency is the way to go.

49 Erasmus will provide a lot of good things …

49a … we hope.

50 Questions?