Software Tools for Concurrent Programming

Peter Grogono

Department of Computer Science and Software Engineering

Concordia University

10 May 2012
The Erasmus Project

Goals:
- a new programming language
- associated infrastructure:
  - compiler
  - run-time system
  - development environment
  - libraries
  - ...
Why introduce a new language?

Cons:
- there are too many languages already
- why will a new language be an improvement over its predecessors?
- *There are only two kinds of programming language: those people always complain about and those nobody uses* — Bjarne Stroustrup

Pros:
- we need a context/environment in which to explore new ideas and validate hypotheses
- programming languages grow: young and clean → old and ugly
- a few languages become popular — at least for a while
The Problem

We need to build applications that are:

- complex
- distributed
- correct
- efficient
- understandable
- adaptable
- maintainable
- ...

Nothing new here 😊
The Problem

Why is it so hard to do this?

- complexity
- high coupling
- implicit coupling
- Object Oriented Programming — a nice idea, but:
  - powerful mechanisms can be overused (e.g., inheritance)
  - interfaces do not tell the whole story
  - objects + threads = disaster?
Hypothesis

Process are a better abstraction than objects

Why?
- control flow is local
- synchronization problems occur only at rendezvous
- interfaces are complete
- bonus: we can exploit multicore architecture

The Erasmus Project is an experiment designed to confirm (or refute) this hypothesis.
Objects
Single-threaded objects

A

B

C

D

E

F
Multi-threaded objects
Joe Armstrong: creator of Erlang
the problem with object-oriented languages is they’ve got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Joe Armstrong, Coders At Work
Processes with flow of control

A

B

C

D

E

F
Communicating processes

Diagram showing the interaction between processes A, B, C, D, E, and F.
Modularization using cells
An old idea:

- Dijkstra: *Cooperating Sequential Processes* (EWD 123, 1965)
  
  Concurrent Pascal, Solo, Edison, Joyce, . . .

- Hoare: *Communicating Sequential Processes* (CACM, 1978)
  
  occam, Ada, Erlang, JCSP, Go, . . .


  
  occam-π, Pict, JoCaml, CubeVM, . . .
The real problem going forward is not program decomposition, but composition. Why are we not currently designing programs as a series of small asynchronous tasks? After all, we have already crossed into a world in which we break programs into objects. Why not then into tasks? Properly done, this would move today’s OOP more closely to its original intent, which was to focus on the messages passed between objects, rather than the objects themselves (according to the widely quoted observation from Alan Kay, who coined the term “object orientation”).

The problems facing such an approach rest on its profound unfamiliarity. There are few languages that provide all the needs of this model, few frameworks that facilitate its design, and few developers conversant with the problems and limitations of this approach. In the meantime, it’s worth considering how an existing program broken down into smaller tasks might function. What exactly would it look like?

Andrew Binstock (Editorial, Dr Dobbs, 12/03/2012)
The big idea is “messaging” — that is what the kernel of Smalltalk/Squeak is all about (and it’s something that was never quite completed in our Xerox PARC phase). The Japanese have a small word — ma — for “that which is in between” — perhaps the nearest English equivalent is “interstitial”. The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

Alan Kay, Squeak mailing list, 1998.
Where are we now?

What we have:

- Desi: a simple PL; probably a subset of Erasmus
- UDC: a compiler that generates Desi Intermediate Language (DIL)
- JIT: a just-in-time compiler that generates machine code
- A very primitive IDE
Where are we now?

What we need:

- Libraries
- Testing tools — difficult for concurrent programs
- Debugger — but debuggers are not used much
- Methods and tools for analyzing programs
Language Features

- **Cells** provide abstraction, organization, and encapsulation
- **Process** perform independent tasks and pass messages to one another
- **Protocols** describe the contents and ordering of messages
- **Routines** perform small tasks without communication
A Differential Equation

\[ A \frac{d^2 x}{dt^2} + B \frac{dx}{dt} + Cx = 0 \]
A Differential Equation

\[ A \frac{d^2x}{dt^2} + B \frac{dx}{dt} + Cx = 0 \]

\[ \ddot{x} = -(B/A)\dot{x} - (C/A)x \]
\[ Nums = \text{protocol} \{ \text{*val: Real} \} \]
\textit{mulcon} = \textbf{process} \quad \text{kin: } +\text{Nums}; \text{ x: } +\text{Nums}; \text{ kx: } -\text{Nums} \{ \\
\quad k: \text{Real} := \text{kin.val}; \\
\quad \textbf{loop} \{ \\
\quad \quad kx.\text{val} := k \times \text{x.val} \\
\quad \} \\
\}
$add = \textbf{process} \ x: +\text{Nums}; \ y: +\text{Nums}; \ sum: -\text{Nums} \ \{ \\
 \hspace{1em} \textbf{loop \ select} \ \{ \\
 \hspace{2em} \| \ t: \text{Real} := x.\text{val}; \ sum.\text{val} := y.\text{val} + t \\
 \hspace{2em} \| \ t: \text{Real} := y.\text{val}; \ sum.\text{val} := x.\text{val} + t \\
 \} \\
\} $

add = process x: +Nums; y: +Nums; sum: -Nums { 
    loop select {
        t: Real := x.val; sum.val := y.val + t
        t: Real := y.val; sum.val := x.val + t
    }
}
$mul = \text{process } x: +\text{Nums}; y: +\text{Nums}; \text{prod: } -\text{Nums} \{$
    \text{loop select } \{$
        || t: \text{Real} := x.\text{val}; \text{prod.}\text{val} := y.\text{val} \times t$
        || t: \text{Real} := y.\text{val}; \text{prod.}\text{val} := x.\text{val} \times t$
    \}$
$\}$

\begin{center}
\begin{tikzpicture}
\node at (0,0) [circle, fill=white, draw, inner sep=2pt, minimum size=5mm] (x) {$x$};
\node at (0,-1) [circle, fill=white, draw, inner sep=2pt, minimum size=5mm] (y) {$y$};
\node at (1,0) [circle, fill=white, draw, inner sep=2pt, minimum size=5mm] (mul) {$x \times y$};
\node at (1.5,0) [circle, fill=white, draw, inner sep=2pt, minimum size=5mm] (mul) {$x \times y$};
\draw (x) -- ++(0.5,0) node[above] {$+$} -- (mul);
\draw (y) -- ++(0.5,0) node[above] {$+$} -- (mul);
\end{tikzpicture}
\end{center}
$DT = \text{constant } 0.001;$

\[ \text{integrate = \textbf{process}} \ x0: +\text{Nums}; \ xdot: +\text{Nums}; \ x: -\text{Nums} \{ \]
\[ \ x: \text{Real} := x0.\text{val}; \]
\[ \ \text{loop} \{ \]
\[ \ \ x += xdot.\text{val} * \ DT; \]
\[ \ \ x.\text{val} := x \]
\[ \} \]
\[ \} \]

\[ \int_{x_0}^{x} \dot{x} \]
split = process x: +Nums; x1: -Nums; x2: -Nums {
    loop {
        x: Real := x.val;
        select {
            || x1.val := x; x2.val := x
            || x2.val := x; x1.val := x
        }
    }
}
\[ \ddot{x} = -(B/A)\dot{x} - (C/A)x \]
\[
\ddot{x} = -(B/A)\dot{x} - (C/A)x
\]
A cell for integration

\[ \int \text{split} \times \text{intmul} \]

Software Tools for Concurrent Programming A Quick Tour 32/70
A cell for integration

\[
\text{intmul} = \text{cell}
\begin{align*}
\text{con1: } & +\text{Nums} & \text{con2: } & +\text{Nums} & \text{inp: } & +\text{Nums} \\
\text{out1: } & -\text{Nums} & \text{out2: } & -\text{Nums} \\
\end{align*}
\]

\{
\begin{align*}
\text{c1, c2: } & \text{Nums} \\
\text{integrate(} & \text{con1, inp, c1);} \\
\text{split(} & \text{c1, c2, out1);} \\
\text{mulcon(} & \text{con2, c2, out2);} \\
\end{align*}
\}
\[
\ddot{x} = -(B/A)\dot{x} - (C/A)x
\]
Static Analysis

- *Static analysis* derives properties of programs from their source code
- E.g., type checking
- Other static techniques:
  - process algebra
  - abstract interpretation
- We need static analysis because testing is inadequate for large, concurrent, distributed systems
A chain of responsibility

Software Tools for Concurrent Programming
Static Analysis
The requesting process

\[ P = \text{process} \ -p1, +p4 \{ \]
\[ \text{loop} \{ \]
\[ (0) \ p1.snd; \]
\[ (1) \ p4.rcv \]
\[ \} \]
\[ \} \]

The diagram shows the processes labeled as \( P, Q, \) and \( R \) with transitions labeled as \( e_1, e_2, e_3, \) and \( e_4 \). The transitions are:

- \( P_0 \xrightarrow{e_1} P_1 \)
- \( P_1 \xrightarrow{e_4} P_0 \)
The last responding process

\[ R = \text{process } +r2, -r3 \{ \]
\[ \text{loop } \{ \]
\[ (0) \ r2.\text{rcv}; \]
\[ (1) \ r3.\text{snd} \]
\[ \} \]
\[ \} \]

\[ R_0 \xrightarrow{e_2} R_1 \]
\[ R_1 \xrightarrow{e_3} R_0 \]
The intermediate responding process

\[ Q = \text{process } +q_1, -q_2, +q_3, -q_4 \{ \]

\[ \text{loop select } \{ (0) \]

\[ || \quad q_1.rcv; \]

\[ \text{if } \text{canAnswer} \]

\[ \text{then } (1) \quad q_4.snd \]

\[ \text{else } (2) \quad q_2.snd \]

\[ || \quad q_3.rcv; (3) \quad q_4.snd \]

\[ \} \]

\[ \}

\[ Q_0 \xrightarrow{e_1} Q_1 \]

\[ Q_0 \xrightarrow{e_1} Q_2 \]

\[ Q_0 \xrightarrow{e_3} Q_3 \]

\[ Q_1 \xrightarrow{e_4} Q_0 \]

\[ Q_2 \xrightarrow{e_2} Q_0 \]

\[ Q_3 \xrightarrow{e_4} Q_0 \]
### Process transitions

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>$e_1$ → P1</td>
<td>Q0</td>
<td>$e_2$ → R1</td>
</tr>
<tr>
<td>P1</td>
<td>$e_4$ → P0</td>
<td>Q0</td>
<td>$e_3$ → R0</td>
</tr>
<tr>
<td></td>
<td>Q0</td>
<td>$e_1$ → Q1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q0</td>
<td>$e_1$ → Q2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q0</td>
<td>$e_3$ → Q3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q1</td>
<td>$e_4$ → Q0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q2</td>
<td>$e_2$ → Q0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q3</td>
<td>$e_4$ → Q0</td>
<td></td>
</tr>
</tbody>
</table>
System transitions

\[ e_1 \rightarrow 110 \]

\[ e_2 \rightarrow 101 \]

\[ e_3 \rightarrow 130 \]

\[ e_4 \rightarrow 000 \]
The requesting process

\[ P = \text{process } -p1, \,+p4 \{ \]
\[ \text{loop select } \{ (0) } \]
\[ || \ p1.snd \]
\[ || \ p4.rcv \]
\[ \} \]
\[ \}

\[ P_0 \xrightarrow{e_1} P_0 \]
\[ P_0 \xrightarrow{e_4} P_0 \]
Process transitions

\[
\begin{array}{ccc}
&P & Q & R \\
P_0 & \xrightarrow{e_1} & P_0 & Q_0 & \xrightarrow{e_1} & Q_1 & R_0 & \xrightarrow{e_2} & R_1 \\
P_0 & \xrightarrow{e_4} & P_0 & Q_0 & \xrightarrow{e_1} & Q_2 & R_1 & \xrightarrow{e_3} & R_0 \\
Q_0 & & Q_0 & \xrightarrow{e_3} & Q_3 & & \\
Q_0 & & Q_0 & \xrightarrow{e_4} & Q_0 & & \\
Q_1 & & Q_0 & & \\
Q_2 & & Q_0 & & \\
Q_3 & \xrightarrow{e_4} & Q_0 & &
\end{array}
\]
System transitions

\[ e_4 \]

\begin{align*}
000 & \xrightarrow{e_1} 010 \\
020 & \xrightarrow{e_3} 030 \\
021 & \xrightarrow{e_1} 001 \\
011 & \xrightarrow{e_1} \quad \\
\end{align*}
A chain of responsibility
The last responding process

\[ R = \text{process } +r2, -r3 \{ \]
\[ \text{loop select } \{ 0 \}
\[ \parallel r2.rcv \\
\[ \parallel r3.snd \]
\[ \} \]
\[ \} \]

\[ R_0 \xrightarrow{e_2} R_1 \]
\[ R_1 \xrightarrow{e_3} R_0 \]
### Process transitions

<table>
<thead>
<tr>
<th></th>
<th>$P$</th>
<th>$Q$</th>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>$P_0$</td>
<td>$Q_0$</td>
<td>$R_0$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>$P_0$</td>
<td>$Q_0$</td>
<td>$R_0$</td>
</tr>
<tr>
<td>$Q_0$</td>
<td>$Q_1$</td>
<td>$Q_0$</td>
<td>$R_0$</td>
</tr>
<tr>
<td>$Q_0$</td>
<td>$Q_2$</td>
<td>$Q_0$</td>
<td>$R_0$</td>
</tr>
<tr>
<td>$Q_0$</td>
<td>$Q_3$</td>
<td>$Q_0$</td>
<td>$R_0$</td>
</tr>
<tr>
<td>$Q_0$</td>
<td>$Q_0$</td>
<td>$Q_0$</td>
<td>$R_0$</td>
</tr>
<tr>
<td>$Q_0$</td>
<td>$Q_0$</td>
<td>$Q_0$</td>
<td>$R_0$</td>
</tr>
<tr>
<td>$Q_0$</td>
<td>$Q_0$</td>
<td>$Q_0$</td>
<td>$R_0$</td>
</tr>
</tbody>
</table>

- $e_1$, $e_2$, $e_3$, $e_4$
System transitions

- Transition from 000 to 010: $e_1$
- Transition from 000 to 030: $e_3$
- Transition from 000 to 020: $e_2$
- Transition from 010 to 000: $e_4$
A better pattern

\[ X = \text{protocol} \{ \, q: \text{Text} \mid r: \text{Float} \, \} \]

\[ Q = \text{process} \, +q_1: X, -q_2: X \, \{ \]
\[ \text{loop select} \, \{ \]
\[ || \, q_1.q; \, \text{if canAnswer then } q_2.r \, \text{else } q_2.q \]
\[ || \, q_1.r; \, q_2.r \]
\[ \} \]
\[ \} \]
Model Checking

This approach is *model checking* — or close to it.

- Well-understood method
- Known to suffer from *state-explosion problem*
- Various techniques to avoid state explosion are also known
Ways of avoiding the state explosion

- *Abstraction* reduces the state space
- Erasmus *cells* allow modular analysis
- *Abstract Interpretation* avoids state space search
Abstract Interpretation

- Idea: approximate semantics using monotonic functions over lattices
- Accumulate properties by simulating execution
- Pioneers: Patrick and Radhia Cousot
- Examples: type checking, interval analysis, ...
Abstract Interpretation of CSP

Cousot applied his techniques to CSP:
Semantic Analysis of Communicating Sequential Processes
Patrick Cousot and Radhia Cousot
Automata, Languages and Programming
Seventh Colloquium, Noordwijkerhout, the Netherlands

- Difficult to read and understand
- Limited results (e.g., only two processes)
- Generally discouraging
- Not much work has been done since 1980
Cousot’s semantics for CSP

- \( i \in [[SxS] \to B] \)
  \[
  i = \lambda((xa, ca), (xb, cb)).[\forall i \in \{1, \pi\}. \ (ca(i) = cb(i) = \lambda(i, 1) \land xa(i) = xb(i)] \lor \{ca(i) = \lambda(i, 1) \land \tau[i]\} \land \tau[i][\{(xa(i), ca(i)), (xb(i), cb(i))\} \land cb(i) = \lambda(i, Cl(i))])]
  \]
  (If \( E, E_1, E_2 \) are sets, \( f \in [E_1 \to E_2] \) and \( E \subseteq E_1 \) then \( f(E) \) is defined as \( \{f(x) : x \in \text{dom}(f) \cap E\} \). The transition relation \( i \) defines the "ready to communicate" or "stop" states which are possible successors of the "entry" states. As far as cooperation between processes is concerned, a process which is never willing to communicate and never terminates does not progress.

- \( Ch = \{<i, j, k \to \ell, m, n> : i, j \in [1, \pi] \land j \in \text{Cl}(i) \land k \in \text{Cl}(\ell) \land n \in \text{Cl}(\ell, m) \land 1 = \Theta(\ell, m, n) \land \ell = \Theta(i, j, k) \land \{e(i, j, k)(x) : x \in \text{dom}(e(i, j, k))\} \cap \ell(\alpha(\ell, m, n)) \neq \emptyset\} \)
  (The set \( Ch \) of communication channels is isomorphic with the set of statically matching pairs of input-output guards).

- \( \mu \in [[SxS] \to B] \)
  \[
  \mu = \lambda((xa, ca), (xb, cb)).[\exists i, j, k \to \ell, m, n] \epsilon Ch :
  \[
  [\forall q \in \{1, \pi\} - \{i, \ell\}, (ca(q) = cb(q)) \land (xa(q) = xb(q))]
  \land [Rsa(i, j, k)((xa(i), ca(i)), (xb(i), cb(i))) \land cb(i) = \lambda(i, Cl(i))]
  \land [e(i, j, k)(xa(i)) \epsilon \ell(\alpha(\ell, m, n))]
  \land [Rsa(i, j, k)((xa(\ell), ca(\ell)), e(i, j, k)(xa(i)), (xb(\ell), cb(\ell))) \land cb(\ell) = \lambda(\ell, Cl(\ell))]
  \]
  (The transition relation \( \mu \) defines the "ready to communicate" or "stop" states which are the possible successors of "ready to communicate" states. The dynamic discrimination of input messages is modeled by dynamic type checking. When several rendez-vous are possible the selection is free. Hence \( \mu \) specifies all possible orderings of the communications between processes).
Abstract Interpretation for Erasmus

Why should we try Abstraction Interpretation for Erasmus?

- Erasmus is not the same as CSP
- We do not aim for a full semantics
- We need only an *approximation* of actual behaviour
- “Fail safe”: if a program has a bad property, we must detect it.
- Accept false positives: if we detect a bad property, the program may not actually possess it.
State: $\langle P_0, Q_0, R_0 \rangle$

State transitions (events omitted):

$\langle P_0, Q_0, R_0 \rangle \Rightarrow \langle P_0, Q_1, R_0 \rangle$

$\langle P_0, Q_0, R_0 \rangle \Rightarrow \langle P_0, Q_2, R_0 \rangle$

Set of states:

$\Sigma = \{ \langle P_0, Q_0, R_0 \rangle, \langle P_0, Q_1, R_0 \rangle, \langle P_0, Q_2, R_0 \rangle, \ldots \}$

Semantic function:

$F(\Sigma) = \Sigma \cup \{ s' \mid s \in \Sigma \text{ and } s \Rightarrow s' \}$
Abstract Interpretation for Erasmus


- $F$ is monotonic. From the definition

\[
F(\Sigma) = \Sigma \cup \{ s' \mid s \in \Sigma \text{ and } s \Rightarrow s' \}
\]

and so $F(\Sigma) \supseteq \Sigma$.

- $\Sigma$ is bounded above by the set of all possible states.

- Consequently, $F$ has a fixed point.
  (That is, an $X$ such that $F(X) = X$.)
To find the fixed point of $F$:

- Start with $\Sigma_0 = \{\sigma_0\}$.  
  $\sigma_0$ is an initial state (e.g., $\langle P_0, Q_0, R_0 \rangle$).

- Compute $\Sigma_0, \Sigma_1, \Sigma_2, \ldots, \Sigma_n, \ldots$  
  where $\Sigma_{n+1} = F(\Sigma_n)$
  until $\Sigma_{n+1} = \Sigma_n$.

- Then $\Sigma_n$ is the fixed point.

- $\Sigma_n$ is the set of all reachable states.
State set for the example

- Case 1 (no concurrency):
  \[ \Sigma = \{000, 001, 101, 110, 120, 130\} \]

- Case 2 (deadlock):
  \[ \Sigma = \{000, 001, 010, 011, 020, 021, 030\} \]

- Case 3 (concurrency, no deadlock):
  \[ \Sigma = \{000, 010, 020, 030\} \]
Detecting deadlock

- Practical procedure for computing $F$:

\[
S := \{ s0 \}
\]
\[\textbf{while choose unmarked } s \text{ from } S:\]
\[\text{mark } s;\]
\[\textbf{for each successor } s' \text{ of } s:\]
\[\text{insert } s' \text{ into } S\]

- Computation detects states with no successors
- These are deadlocked states
- Computation checks reachable states \textit{only}
- Processes with only one state can be ignored
- Computation can be performed per cell
Abstract Interpretation: Alternatives

- We can check properties other than deadlock:
  All we have to do is define a suitable abstract semantics.
- We can use partial orders other than reachable states:
  E.g., the lattice of failures in conventional CSP semantics.
Communication without selection

\[ P = \text{process } p1: -K \{ \text{loop } \{ p1.snd \} \} \]

\[ Q = \text{process } q1: +K \{ \text{loop } \{ q1.rcv \} \} \]
\[ P = \text{process } p1: \ -K \{ \]
\[ \quad \text{loop select} \{ \]
\[ \quad \quad | | \quad p1.snd \]
\[ \quad \quad | | \quad p2.rcv \]
\[ \quad \quad | | \quad p3.rcv \]
\[ \quad \} \]
\[ \}
\]

\[ Q = \text{process } q1: \ +K \{ \]
\[ \quad \text{loop} \{ \]
\[ \quad \quad q1.rcv \]
\[ \quad \} \]
\[ \}
\]
Communication with selection by two processes

P = process p1: -K {
  loop select {
    || p1.snd
    || p2.rcv
    || p3.rcv
  }
}

Q = process q1: +K {
  loop select {
    || q1.rcv
    || q2.rcv
    || q3.rcv
  }
}
One server, many clients on a channel
Multiple servers and clients on a channel
Verifying Communication Algorithms

- Describe the communication algorithm in pseudocode
- Translate the pseudocode into a specification written in micro Common Representation Language 2 (mCRL2)
- Process the specification with the Linearizer and LTS Generator from the mCRL2 toolkit
- Verify that the Labelled Transition System (LTS) has the desired properties
If we can achieve these goals:

- High-level analysis (abstract interpretation), performed once for each program, will detect potential communication problems.
- Low-level analysis (verification of communication algorithms by process algebra), performed once only, will prevent synchronization problems.
Brian Shearing and Peter Grogono, principal investigators
Nima Jafroodi, Ph.D., process algebras
Maryam Zakeryfar, Ph.D., abstract interpretation
Duo Peng, M.C.Sc., web applications
Shruti Rathee, M.C.Sc., not decided yet