A second generation model

The notion of instant reconsidered

- In SCCS, at each instant, each thread performs exactly 1 action.
- In the semi-formal model for synchronous algorithms, at each instant, each thread writes and reads exactly once in the point-to-point channels.

A more liberal viewpoint At each instant, each thread performs an arbitrary (but hopefully finite) number of actions. The instant ends when each thread has either terminated its task for the current instant or it is suspended waiting for events that cannot arise.

Timed CCS

A formalisation of this viewpoint in the framework of CCS.

$$\begin{array}{lll} \alpha & ::= \tau \mid \ell & \text{(usual actions)} \\ \mu & ::= \alpha \mid \mathsf{tick} & \text{(extended actions)} \\ P & ::= \cdots \mid (P \triangleright Q) & \text{(extended processes)} \end{array}$$

tick represents the move to the following instant.

$(P \triangleright Q)$ *else_next* operator: if cannot run P now, run Q at the following instant.

Labelled transition system

Usual rules for the α actions plus:

$$\frac{P \xrightarrow{\alpha} P'}{(P \rhd Q) \xrightarrow{\alpha} P'}$$

Plus special rules for the tick action that say that a process can tick if and only if it cannot perform τ actions.

$$\frac{P \overset{\tau}{\not\to} \cdot}{(P \triangleright Q) \overset{\mathsf{tick}}{\to} Q} \qquad \frac{}{0 \overset{\mathsf{tick}}{\to} 0}$$

$$\frac{}{\ell.P \overset{\mathsf{tick}}{\to} \ell.P} \qquad \frac{P_i \overset{\mathsf{tick}}{\to} P_i' \quad i = 1,2 \quad (P_1 \mid P_2) \overset{\tau}{\not\to} \cdot}{(P_1 \mid P_2) \overset{\mathsf{tick}}{\to} (P_1' \mid P_2')}$$

$$\frac{P_i \overset{\mathsf{tick}}{\to} P_i' \quad i = 1,2}{(P_1 + P_2) \overset{\mathsf{tick}}{\to} (P_1' + P_2')} \qquad \frac{P \overset{\mathsf{tick}}{\to} P'}{\nu a\ P \overset{\mathsf{tick}}{\to} \nu a\ P'}$$

Exercise (on formalising tick actions)

1. Check that $P \xrightarrow{\mathsf{tick}} \cdot$ if and only if $P \stackrel{\tau}{\not\rightarrow} \cdot$. (Indeed, this is in perfect agreement with the usual feeling that you do not see time passing when you have something to do)

2. The previous lts uses the negative condition $P \stackrel{\tau}{\not\rightarrow} \cdot$. Show that this condition can be formalised in a positive way by defining a formal system to derive judgments of the shape $P \downarrow L$ where L is a set of observable actions and $P \downarrow L$ if and only if $P \stackrel{\tau}{\not\rightarrow} \cdot$ and $L = \{\ell \mid P \xrightarrow{\ell} \cdot\}$.

Exercise (continuations of tick action)

We say that P is a 'CCS process' if it does not contain the *else_next* operator. Show that:

1. If $P \xrightarrow{\mathsf{tick}} Q_1$ and $P \xrightarrow{\mathsf{tick}} Q_2$ then $Q_1 = Q_2$.
2. If P is a CCS process and $P \xrightarrow{\mathsf{tick}} Q$ then $P = Q$.

Exercise (programming a switch)

Let $\mathsf{tick}\ .P = (0 \triangleright P)$ and $\mathsf{tick}^{\,n}.P = \mathsf{tick}\ \cdots \mathsf{tick}\ .P$, n times.

1. Programme a *light switch*

$$Switch(press, off, on, brighter)$$

that behaves as follows:

- Initially the switch is off.
- If the switch is off and it is pressed then the light turns on.
- If the switch is pressed again in the following 2 instants then the light becomes brighter while if it is pressed at a later instant it turns off again.
- If the light is brighter and the switch is pressed then it becomes off.

2. Programme a *fast user* $Fast(press)$ that presses the switch every 2 instants and a *slow user* $Slow(press)$ that presses the switch every 4 instants.

3. Consider the systems:

$$\nu press\ (\ Switch(press, off, on, brighter) \mid Fast(press)\ )$$
$$\nu press\ (\ Switch(press, off, on, brighter) \mid Slow(press)\ )$$

and determine when the light is going to be off, on, and bright.

Exercise (bisimulation for TCCS)

Let $\approx_{\mathsf{tick}}$ be the notion of weak bisimulation where as expected

$$\stackrel{\mathsf{tick}}{\Rightarrow} = \stackrel{\tau}{\Rightarrow} \circ \stackrel{\mathsf{tick}}{\longrightarrow} \circ \stackrel{\tau}{\Rightarrow} .$$

1. Show that $\approx_{\mathsf{tick}}$ is preserved by parallel composition.
2. Show that $((P_1 \triangleright P_2) \triangleright P_3) \approx_{\mathsf{tick}} (P_1 \triangleright P_3)$.

Exercise (CCS vs. TCSS)

Recall that a process is *reactive* if every derivative strongly normalizes with respect to τ reduction. Let Ω be the always diverging process $\tau.\tau.\tau\cdots$.

1. Is $0 \approx_{\mathsf{tick}} \Omega$?
2. Suppose P, Q are CCS processes. Does $P \approx_{\mathsf{tick}} Q$ imply $P \approx Q$?
3. Suppose further that P, Q are reactive. Does $P \approx Q$ imply $P \approx_{\mathsf{tick}} Q$?

References

- A notion of 'timed' CCS is first introduced in

  Yi Wang. A calculus of real time systems. PhD thesis. 1991.

  This calculus has a tick (x) operator that describes the passage of x time units where x is a non-negative real.

- A kind of *else_next* operator is proposed in

  Nicolin, Sifakis. The algebra of timed processes. Information and Computation, 1994.

- A *testing* semantics of a process calculus very close to the one presented here is given in

  Hennessy, Reagan. A process algebra of timed systems. Information and Computation, 1995.

However, it seems fair to say that all these works generalise to CCS ideas that were presented in

> Berry, Cosserat. The Esterel synchronous programming language and its mathematical semantics. INRIA report, 1988.

Two basic differences in the Esterel approach are that:

1. Threads interact through *signals*.
2. The resulting calculus is *deterministic*.

Next, we will take sometime to discuss this approach.

Another popular formalism for describing timed systems is

Alur, Dill. A theory of timed automata. Theoretical Computer Science, 1994.

- This is an enrichment of finite state automata with timing constraints which still enjoys decidable model-checking properties.
- It is more a *specification language* for finite control systems than a *programming language*.
- The *implementability* of certain specifications is actually problematic and it is still a research problem.

Signal based interaction and determinacy

SL model

- Threads interact through *signals* (rather than channels).
- A signal is either emitted or not. Once it is emitted it *persists* during the instant and it is *reset* at the end of it.
- Thus the collection of emitted signals grows *monotonically* during each instant.

A simple calculus for the SL model

We present the calculus as a fragment of timed CCS. Write $s, s', \ldots$ for *signal* names.

$$\text{Processes} \quad P \quad ::= 0 \parallel s.P, P \parallel (\mathsf{emit}\ s) \parallel (P \mid P) \parallel \nu s\ P \parallel A(\mathbf{s})$$

where:

$$\begin{aligned} s.P, Q \quad & = (s.P \triangleright Q) \\ (\mathsf{emit}\ s) \quad & = (\overline{s}.Emit(s) \triangleright 0) \\ & \quad \text{where: } Emit(s) = (\overline{s}.Emit(s) \triangleright 0) \end{aligned}$$

Remarks on the calculus

- There is no sum and no prefix for emission (cf. asynchronous π-calculus).
- The input is a specialised form of the input prefix and the $(\_ \triangleright \_)$ operator.
- The derived synchronisation rule is:

$$(\mathsf{emit}\ s) \mid s.P, Q \xrightarrow{\tau}\xrightarrow{\tau} (\mathsf{emit}\ s) \mid P$$

  (the second τ is just recursion unfolding and we will ignore it in the following).

- Note that:

$$(\mathsf{emit}\ s) \mid s.P_1, Q_1 \mid s.P_2, Q_2 \stackrel{\tau}{\Rightarrow} (\mathsf{emit}\ s) \mid P_1 \mid P_2$$

Coding tick and await

- The tick action can be expressed as

$$\mathsf{tick}\ .P = \nu s\ s.0, P \qquad s \notin fn(P)$$

- A persistent input (as in TCCS) is expressed as:

$$\mathsf{await}\ s.P = A(\mathbf{s}), \quad \text{where } A(\mathbf{s}) = s.P, A(\mathbf{s})$$

where $fn(P) \cup \{s\} = \{\mathbf{s}\}$.

Coding an *if_then_else* on signals

- For every signal s, we can programme a process $Echo(s, s_-, s_+)$ as follows:

$$\begin{aligned} Echo(s, s_-, s_+) &= s.\mathsf{tick}\ .Echo_+(s, s_-, s_+), Echo_-(s, s_-, s_+) \\ Echo_+(s, s_-, s_+) &= (\mathsf{emit}\ s_+) \mid Echo(s, s_-, s_+) \\ Echo_-(s, s_-, s_+) &= (\mathsf{emit}\ s_-) \mid Echo(s, s_-, s_+) \end{aligned}$$

  that tells us in the following instant whether the signal s was emitted or not.

- Then we can define an if_then_else as follows:

$$(\mathsf{ite}\ s\ P\ Q) = s_+.P, 0 \mid s_-.Q, 0$$

Example: programming a NOR gate in SL

- Input signals: s_0, s_1.
- Output signal: s.
- At instant $i+1$, emit s iff neither s_0 nor s_1 were emitted at instant i.

$$
\begin{aligned}
N &= N_0 \mid \mathit{Echo}(s_0) \mid \mathit{Echo}(s_1) \\
N_0 &= \mathsf{tick}\ .(\ \mathsf{ite}\ s_0\ N_0\ (\mathsf{ite}\ s_1\ \ N_0\ N_1)\ ) \\
N_1 &= (\mathsf{emit}\ s) \mid N_0
\end{aligned}
$$

Remarks

- We can *program* the boolean function NOR rather than writing down its *truth table*.
- Several threads can *share the same signal*:

$$(\mathsf{emit}\ s) \mid s.P_1, Q_1 \mid s.P_2, Q_2$$

- We can react to the *absence of a signal* at the end of the instant and therefore we can regard a signal as a *binary information*.

Exercise (programming the light switch in SL)

- Reprogramme the light switch in SL.
- Compare with the solution based on TCCS.

Some historical remarks

- In the ESTEREL model it is actually possible to *react immediately* (rather than at the end of the instant) to the absence of a signal.
- This requires some semantic care, to avoid writing paradoxical programs such as:

  $$s.0, (\mathsf{emit}\ s)$$

  which are supposed to emit s when s is not there (cf. stabilization problems in synchronous circuits).
- It also requires some clever *compilation techniques* to determine whether a signal is not emitted. Infact these techniques (so far!) are specific to *finite state models*.

- SL is a *relaxation* of the ESTEREL model where the *absence of a signal* can only be detected at the end of the instant.
- If we forget about *name generation*, then the SL model essentially defines a kind of *monotonic Mealy machine*. Monotonic in the sense that output signals can only depend *positively* on input signals (within the same instant).
- The monotonicity restriction allows to *avoid the paradoxical programs* (monotonic boolean equations do not have a least fixed point!).
- The SL model has a natural and *efficient implementation model* that works well for *general programs* (not just finite state machines).

- The ESTEREL/SL models were conceived in Sophia-Antipolis shortly after the *SCCS/Meije models* and in the *same research team*.
- In spite of this, there is *no* strong formal result on the possibility/impossibility of embedding one model into the other up to some reasonable equivalence (*e.g*, SCCS vs. TCCS).

Coming next

1. Determinacy.
2. A condition to ensure reactivity.
3. Remarks on SL semantics and extensions of SL with data types.
4. A programming exercice (Turing equivalence).

(Very) Strong confluence

A basic property is:

$$\frac{P \xrightarrow{\tau} P_1 \qquad P \xrightarrow{\tau} P_2}{P_1 = P_2 \text{ or } \exists\, Q \;\; (P_1 \xrightarrow{\tau} Q, P_2 \xrightarrow{\tau} Q)}$$

NB We close the diagram in *at most one step* and *up to* α*-renaming*.

Proof idea

- Internal reductions are due either to *unfolding* or to *synchronisation*.
- The only possibility for a *superposition* of the redexes is:

$$(\mathsf{emit}\ s) \mid s.P_1, Q_1 \mid s.P_2, Q_2$$

- And we exploit the fact that emission is *persistent*.

A simple static analysis that guarantees reactivity

- We assume the instruction tick is *explicitly* used in the program.
- We compute an *over-approximation* of the *control flow* of the system of equations.
- We check that within an instant it is *not possible to loop* through a thread identifier.

Call graph

If P is a process then $Call(P)$ is an *over-approximation* of the set of process identifiers that P may possibly call within the current instant:

$$
\begin{array}{ll}
Call(P) & = \mathsf{case}\ P \\
0 & : \emptyset \\
\mathsf{tick}\ .P & : \emptyset \\
B(\mathbf{a}) & : \{B\} \\
(\mathsf{emit}\ s) & : \emptyset \\
s.P, Q & : Call(P) \\
\nu s\ P & : Call(P) \\
P_1 \mid P_2 & : Call(P_1) \cup Call(P_2)
\end{array}
$$

Given a system of equations:

$$\begin{aligned} A_1(\mathbf{a}_1) &= P_1 \\ \ldots & \\ A_n(\mathbf{a}_1) &= P_n \end{aligned}$$

build a (directed) *call graph* with *nodes* $\{A_1, \ldots, A_n\}$ and such that

$$(A_i, A_j) \text{ is an edge iff } A_j \in \mathit{Call}(A_i)$$

Proposition If the call graph has no loops then any process relying on the related system of equations is reactive.

Proof idea

- If the graph has no loops then we can define a *well-founded order* $>$ on thread identifiers such that $A > B$ whenever there is an edge from A to B in the call graph.
- A process is essentially a *multi-set* of threads:

$$\{\!|P_1, \ldots, P_n|\!\}$$

- Whenever we perform an internal reduction either we reduce the size of a P_i or we unfold a recursive equation $A_i(\mathbf{a}) = P_i$ and then we have:

$$Call(A_i) = \{\!|A_i|\!\} >_{mset} Call(P_i)$$

Exercise

Define a well-founded measure on processes that shows that all internal reductions terminate.

Remarks on a compositional semantics for the SL model

- We have defined a bisimulation semantics $\approx_{\mathsf{tick}}$ for TCCS. This semantics can be applied to SL too.
- In SL one can expect additional equations to hold. For instance,

  $$s.(\mathsf{emit}\ s), 0 \text{ should be 'equivalent' to } 0$$

  (cf. asynchronous communication).

  Exercise Check that the equation does not hold in the TCCS embedding.
- More generally, because SL is deterministic one can expect a collapse of the *bisimulation* semantics with a *trace* semantics.

SL with data types

- The language with *pure* signals is *deterministic*.
- Reasonable extension to *(infinite) data domains*. The resulting language becomes *non-deterministic*.
- Efficient *implementation model*.
- Embedded in many *programming environments*: C, C++, Scheme, ML.
- Significant *applications*: event-driven control, data flow, GUI, simulations, web services, multiplayer games.

Two references

- Boussinot. Reactive C: an extension of C to program reactive systems. *Soft. Practice and Experience*, 1991.
- Mandel-Pouzet. Reactive ML, a reactive extension to ML. In *Proc. ACM PPDP*, 2005.

A *Synchronous* π-calculus based on the SL model

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

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

- P is a π-calculus process if we forget about the else branches of the read instructions.
- In $S\pi$, $\overline{s}v$ should be understood as (emit s v).

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

$$P_1 = \nu s_1, s_2\ (s_2(z).0, A(\sigma(x), \sigma(y)), B(!s_1))$$

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

$$P_2 \xrightarrow{\mathsf{tick}} P_2' = \nu s_1, s_2 \; B(\ell)$$

  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.

Non-determinacy

Non-determinism arises when emitting *distinct values* on the same signal:

Within the instant...

$$s(x).P, Q \mid \overline{s}0 \mid \overline{s}1$$

with suitable encodings of 0 and 1.

...and at the end of the instant

$$\mathsf{tick}\ .A(!s) \mid \overline{s}0 \mid \overline{s}1$$

Typing constraints can be imposed to avoid these situations (cf. affine typing for the π-calculus).

Simulating push-down automata

- We want to write a SL program that simulates a *deterministic push-down automata*.
- We adapt ideas used in the encoding of 2 counters machines in CCS.
- The example serves three purposes:
  1. Some *non-trivial hacking* in SL.
  2. An opportunity for *reactivity analysis*.
  3. Shows that the SL model is *Turing equivalent*.

- A *configuration* is a pair (q, w) where $q \in Q$ is a *state* and $w = S \cdots SZ$ is a *stack*.
- Possible transitions are:

$$\begin{array}{lll} (q, w) & \to (q', Sw) & \text{(increment)} \\ (q, Sw) & \to (q', w) & \text{(decrement)} \\ (q, Z) & \to (q', Z) & \text{(positive zero test)} \\ (q, Sw) & \to (q', Sw) & \text{(negative zero test)} \end{array}$$

- We assume that the automaton is *deterministic* and that we *check that the stack is not empty* before running a decrement instruction.

- Associate a *recursive equation* with each state/instruction:

$$q \quad = (\mathsf{emit}\ inc) \mid \mathsf{await}\ ack.\mathsf{tick}\ .q'$$

$$q \quad = (\mathsf{emit}\ dec) \mid \mathsf{await}\ ack.\mathsf{tick}\ .q'$$

$$q \quad = zero.(\mathsf{tick}\ .q'), q''$$

- With a configuration $(q, S \cdots SZ)$ associate the program:

$$\nu \mathbf{s}_0, \ldots \mathbf{s}_n \ (q(\mathbf{s}_0) \mid S(\mathbf{s}_0, \mathbf{s}_1) \mid \cdots \mid S(\mathbf{s}_{n-1}, \mathbf{s}_n) \mid Z(\mathbf{s}_n))$$

- Z is described by the equation:

$$Z(\mathbf{s}) \quad = (\mathsf{emit}\ \mathit{zero}) \mid \mathit{inc}.((\mathsf{emit}\ \mathit{ack}) \mid \mathsf{tick}\ .\nu \mathbf{s}'(S(\mathbf{s}, \mathbf{s}') \mid Z(\mathbf{s}'))), Z(\mathbf{s})$$

- and S by the equations:

$$S(\mathbf{s},\mathbf{s}') \quad = S_{inc}(\mathbf{s},\mathbf{s}') \mid S_{dec}(\mathbf{s},\mathbf{s}')$$

$$S_{inc}(\mathbf{s},\mathbf{s}') \quad = inc.\mathsf{tick}\ .S^{+}(\mathbf{s},\mathbf{s}'),(\mathsf{ite}\ dec\ 0\ S_{inc}(\mathbf{s},\mathbf{s}'))$$

$$S_{dec}(\mathbf{s},\mathbf{s}') \quad = dec.\mathsf{tick}\ .S^{r}(\mathbf{s},\mathbf{s}'),(\mathsf{ite}\ inc\ 0\ S_{dec}(\mathbf{s},\mathbf{s}'))$$

$$S^{+}(\mathbf{s},\mathbf{s}') \quad = (\mathsf{emit}\ ack) \mid \nu\mathbf{s}''\ (S(\mathbf{s},\mathbf{s}'') \mid S(\mathbf{s}'',\mathbf{s}'))$$

$$S^{r}(\mathbf{s},\mathbf{s}') \quad = zero'.(\mathsf{tick}\ .(\mathsf{emit}\ ack) \mid Z(\mathbf{s})),((\mathsf{emit}\ dec') \mid S^{l}(\mathbf{s},\mathbf{s}'))$$

$$S^{l}(\mathbf{s},\mathbf{s}') \quad = ack'.\mathsf{tick}\ .((\mathsf{emit}\ ack) \mid S(\mathbf{s},\mathbf{s}')),S^{l}(\mathbf{s},\mathbf{s}')$$

Some dynamics

Increment

$$SSZ \to S^{+}SZ \to SSSZ$$

Decrement

$$SSSZ \to S^{r}SSZ \to S^{l}S^{r}SZ \to S^{l}S^{l}S^{r}Z \to S^{l}S^{l}Z \to S^{l}SZ \to SSZ$$

Remarks

- In S we have two parallel threads: one waiting for the signal *inc* and the other for the signal *dec*. At the end of the instant, the first thread will abort if the signal *dec* has been emitted (and symmetrically).
- The call graph associated with the system of equations is *acyclic*. Hence the program is *reactive*.
- It is easy to adapt the program to simulate a *two counters machine* (name generation is essential here).

Exercise

Show that the presented encoding of push-down automata can be adapted to CCS.

Some references

- G. Berry and G. Gonthier, *The Esterel synchronous programming language*. Science of computer programming, 1992.

  It introduces an imperative language to program reactive systems. The language can be compiled to finite automata. The semantics allows to react immediately to the absence of a signal. Static analysis is required to avoid 'causality problems'.

- F. Boussinot and R. De Simone, *The SL Synchronous Language*. IEEE Trans. on Software Engineering, 1996.

  Relaxation of the Esterel model. It allows reaction to the absence of a signal only at the end of the instant.

- A., *The SL synchronous language, revisited. Journal of Logic and Algebraic Programming*, 2007.

  A process calculus description of the SL model with pure signals.

- A., *A synchronous π-calculus. Information and Computation*, 2007.

  The generalisation of the previous work to signals carrying data values.

Exercise (revision)

In the context of SCCS/Meije, we specify a ternary operator rr (round robin) by the rule:

$$\frac{P_0 \xrightarrow{\alpha} P_0'}{rr(P_0, P_1, P_2) \xrightarrow{\alpha} rr(P_1, P_2, P_0')}$$

Show that the operator rr is definable as a SCCS/Meije process. This amounts to define a SCCS/Meije process $RR(P_0, P_1, P_2)$, parametric in P_0, P_1, P_2, which is strongly bisimilar to $rr(P_0, P_1, P_2)$.

Exercise (revision)

We can embed SL terms in TCCS and then compare them using the weak bisimulation for TCCS. Prove or disprove:

1. $s.(s.P, Q), Q \approx_{\mathsf{tick}} s.P, Q.$
2. $(\mathsf{emit}\ s) \mid s.P, Q \approx_{\mathsf{tick}} (\mathsf{emit}\ s) \mid P.$