Semantics

Concurrency and Distributed Systems

January 2023

Contents

- Process language
- Traces
- Failures
- Divergences

Process language

Communicating Sequential Processes (CSP) is a process modelling language with a compositional, mathematical semantics.

Models representing designs, assumptions, or specifications described in terms of processes.

Processes and behaviours are described in terms of abstract transactions or events.

Origins

- **Cooperating Sequential Processes (lectures, monograph)**
  E. W. Dijkstra, 1965.
- **Communicating Sequential Processes (paper)**
  C. A. R. Hoare, 1978.
- **Communicating Sequential Processes (book)**
  C. A. R. Hoare, 1985.
- **Theory and Practice of Concurrency (book)**
  A. W. Roscoe, 1997.

Programming languages on the whole are very much more complicated than they used to be: object orientation, inheritance, and other features are still not really being thought through from the point of view of a coherent and scientifically well-based discipline or a theory of correctness.

My original postulate, which I have been pursuing as a scientist all my life, is that one uses the criteria of correctness as a means of converging on a decent programming language design [...] one in which the different components of the program correspond clearly to different components of its specification, so you can reason compositionally about it.

interview, July 2002

To have our best advice ignored is the common fate of all who take on the role of consultant, ever since Cassandra pointed out the dangers of bringing a wooden horse within the walls of Troy.

1980 Turing Award Lecture

Compositionality

The meaning of a composite term should be entirely determined by the meaning of its components.

In the case of our process modelling language, the meaning of a term is a set of behaviours.

The behaviours associated with a compound process description should be entirely determined by those of its components.

Constraints

We should think of a modelling language not as a language of programs, but as a language of constraints.

The behaviours associated with a process description are those allowed by that description.

If we extend a specification with an additional process, then we may further constrain the occurrence of existing transactions.

c.f. each new person added to the list of attendees further constrains the occurrence or scheduling of a meeting

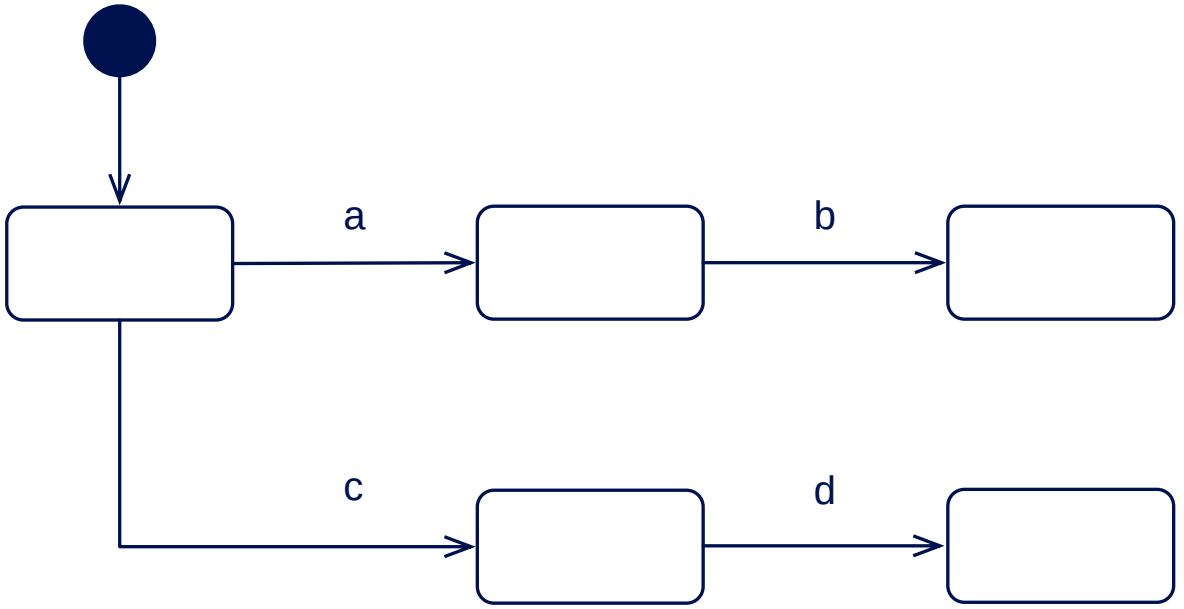
Syntax

`STOP`	**stop**	
`a -> P`	**prefix**	**then**
`P [] Q`	**external choice**	**or**
`P |~| Q`	**internal choice**	**or**
`P [A || B] Q`	**parallel**	**and**
`P \ A`	**hide**	

Example

```
a -> (b -> STOP)
[]
c -> (d -> STOP)
```

Example



Events

An event represents an abstract transaction, an atomic interaction between processes.

We can introduce an event name using the keyword `channel`**.**

(In general, we will be introducing families of parameterised events, or channels.)

Example

A pair of events `a` and `b` may be introduced using

```
channel a
channel b
```

or

```
channel a, b
```

Traces

We can define a function `traces` from the terms in our language to the set of all sequences of events.

```
traces : CSP -> Set(seq(Event))
```

Convexity

Not every set of sequences of events corresponds to the trace semantics of a process.

Any set of sequences S representing the traces of a process must satisfy the following two conditions:

- **S contains the empty sequence <>**
- **if `tr^<a>` is in S, then so is `tr`**

where

```
tr1^tr2
```

denotes the concatenation of sequences `tr1` and `tr2`

Example

```
{ <>, <a>, <a,b>, <c>, <c,d> }
```

is the trace set of a process

```
{ <>, <a,b>, <c,d> }
```

is not

Stop

STOP is a process that allows no events.

It corresponds to a state with no outgoing (or internal) transitions.

It is also equivalent to a combination of processes in which there is no transaction that would be allowed next by all of the parties involved: for this reason it is sometimes called deadlock.

Traces: stop

```
traces(STOP) = { <> }
```

Event prefix

We can include an event in a process description using the event prefix operator ->.

The left operand is an event-valued expression; the right is a process-valued expression.

The operator associates to the right.

Interpretation: prefix

If `a` is an event-valued expression and `P` is a process-valued expression, then

```
a -> P
```

is a process that will allow only the event with value `a`.

If this event occurs, the process will then allow whatever `P` allows.

Traces: prefix

```
traces(a->P) = union( { <> },
                      { <a>^tr | tr <- traces(P) } )
```

where

```
union(s1,s2)
```

denotes the union of sets `s1` and `s2`.

Example

```
traces(a -> b -> STOP)

  = union({<>},
          {<a>^tr | tr <- traces(b -> STOP)})

  = union({<>},
          {<a>^tr | tr <- union({<>},
                                {<b>^tr |
                                  tr <- traces(STOP)})}

  = {<>,<a>,<a,b>})
```

External choice

We can describe an external choice between two processes using the external choice operator [].

The left-hand and right-hand arguments are both process-valued expressions.

Interpretation: external choice

If `P` and `Q` are process-valued expressions, then

```
P [] Q
```

is a process that is ‘ready to behave’ as either `P` or `Q`.

Until the first event occurs, it will allow any event that `P` or `Q` would allow.

After the first event occurs, it will allow any sequence of events, and refuse any set of events, that the chosen process could allow.

Traces: external choice

```
traces(P [] Q) = union(traces(P), traces(Q))
```

compare: internal choice

Example

```
EXT =
  a -> b -> STOP
  []
  c -> d -> STOP

traces(EXT)
= union(traces(a -> b -> STOP), traces
                      (c -> d -> STOP))
  = {<>,<a>,<a,b>,<c>,<c,d>}
```

note that -> binds more closely than []

Definition

We can introduce a new process name by following it with an equality symbol and a process-valued expression.

The new process name can then be used in other process definitions at the same level.

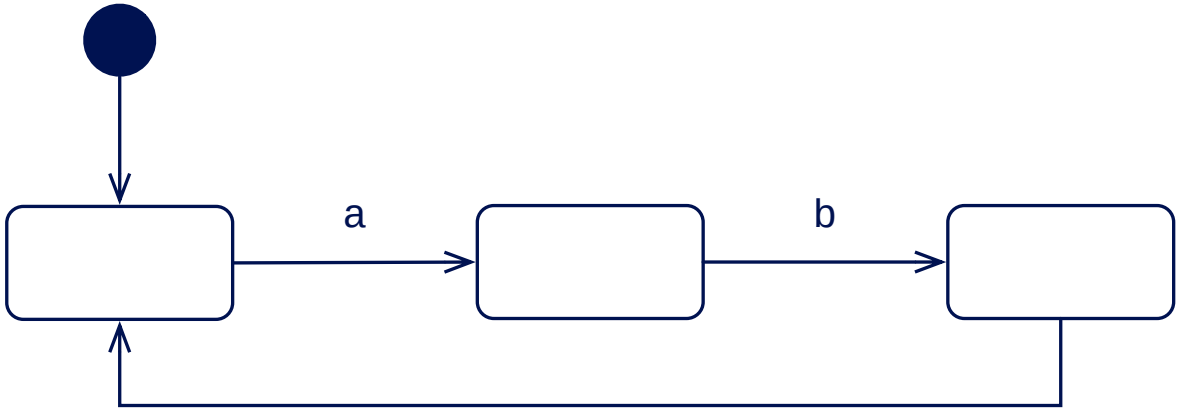
It can also be used on the right-hand side of its own definition, within the process-valued expression, making that definition recursive.

Example

```
REC = a -> b -> REC
```

```
traces(REC) = {<>,<a>,<a,b>,<a,b,a>,<a,b,a,b>,...}
```

Example



Internal choice

We can describe an internal choice between two processes using the internal choice operator |˜|.

The left-hand and right-hand arguments are both process-valued expressions.

Interpretation: internal choice

If `P` and `Q` are process-valued expressions, then

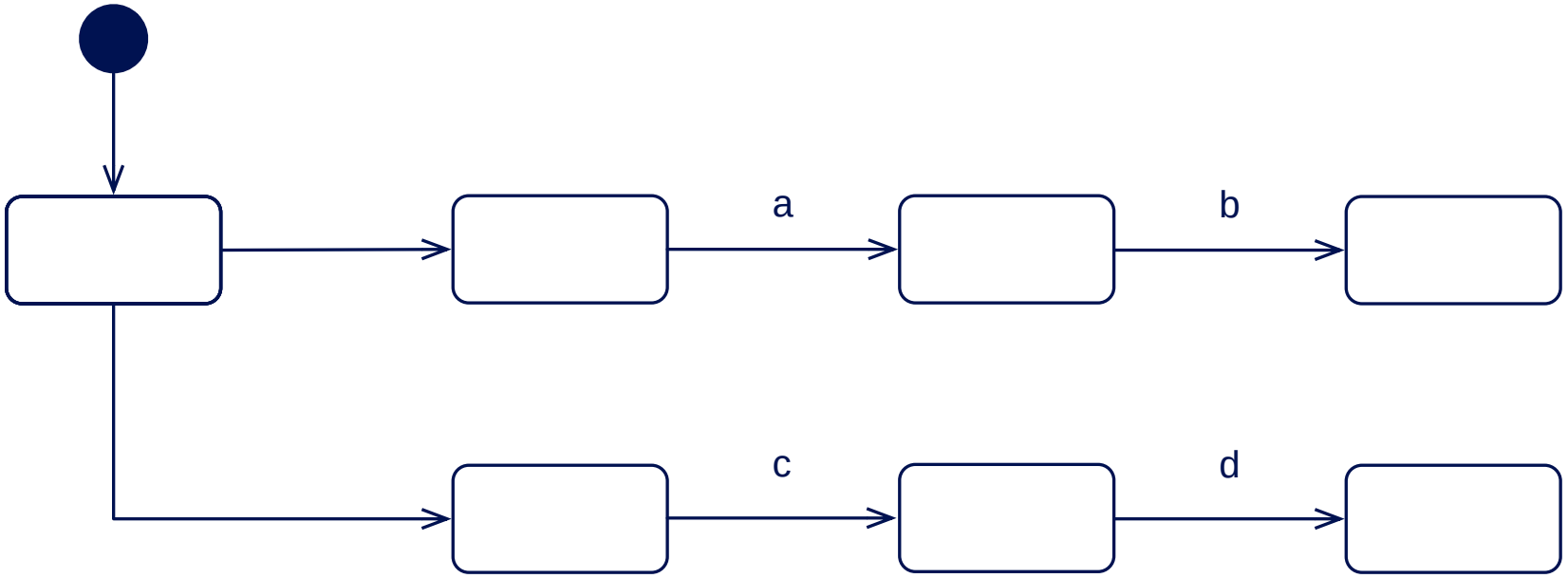
```
P |~| Q
```

is a process that could be either `P` or `Q`.

Example

```
INT =
  a -> b -> STOP
  |~|
  c -> d -> STOP
```

Example



Traces: internal choice

```
traces(P |~| Q) =
  union(traces(P), traces(Q))
```

compare: external choice

Example

```
INT =
  a -> b -> STOP
  |~|
  c -> d -> STOP

traces(INT)
= union(traces(a -> b -> STOP), traces
                      (c -> d -> STOP))
  = {<>,<a>,<a,b>,<c>,<c,d>}
```

Traces are not enough

Our trace semantics does not contain enough information to distinguish between internal and external choice.

A more detailed semantics is needed if we wish to reason about the availability of events.

Example

Used as a specification, INT corresponds to the requirement that either a or c should be initially available*.

```
INT =
  a -> b -> STOP
  |~|
  c -> d -> STOP
```

In contrast, EXT would require that both events should be initially available.

*and that if a occurs, b should be available, and so on.

Failures

We can define a function `failures` from the terms in our language to the set of all pairs of sequences and sets of events.

```
failures : CSP -> Set(seq(Event).Set(Event))
```

Each pair consists of a trace that the process may allow, together with a set of events that may then be refused.

Failures: stop

```
failures(STOP) = { (<>,s) | s <- Set(Events) }
```

Failures: prefix

```
failures(a->P) =
  union( { (<>,ref) |
                ref <- Set(Event), not(member(a,ref)) },
         { (<a>^tr,ref) | (tr,ref) <- failures(P) } )
```

Example

The process

```
a -> b -> STOP
```

allows only a **as a first event.**

If a **occurs, it will then allow the event** b**, and no other. If** b **is then performed, it will allow no further events.**

Example

```
traces(a -> b -> STOP) =
  {<>,<a>,<a,b>}
```

Example

```
failures(a -> b -> STOP) =
  { (<>, {}),
    (<>,{b}),
    (<a>,{}),
    (<a>,{a}),
    (<a,b>,{}),
    (<a,b>,{a}),
    (<a,b>,{b}),
    (<a,b>,{a,b}) }
```

Failures: internal choice

```
failures(P |~| Q) = union(failures(P),failures(Q))
```

Failures: external choice

```
failures(P [] Q) =
  union({ (<>,ref) |
            (<>,ref) <- inter(failures(P),failures(Q)) },
          { (tr,ref) |
            (tr,ref) <- union(failures(P),failures(Q)),
            tr != <> })
```

where `inter(s1,s2)` **denotes the intersection of sets** `s1` **and** `s2`**.**

Example

```
failures(INT) = {
  (<>, {}), (<>,{a}), (<>,{b}), (<>,{c}), (<>,{d}), (<>,{a,b}),
            , (<>,{a,d}), (<>,{b,c}), (<>,{b,d}), (<>,{c,d}),
              , (<>,{a,b,d}),               , (<>,{b,c,d}),
  (<a>,{}), (<a>,{a}), (<a>,{c}), (<a>,{d}), (<a>,{a,c}),
  (<a>,{a,d}), (<a>,{c,d}), (<a>,{a,c,d}), (<a,b>,{}),
  (<a,b>,{a}), (<a,b>,{b}), (<a,b>,{c}), (<a,b>,{d}),
  (<a,b>,{a,b}), (<a,b>,{a,c}), (<a,b>,{a,d}), (<a,b>,{b,c}),
  (<a,b>,{b,d}), (<a,b>,{c,d}), (<a,b>,{a,b,c}), (<a,b>,{a,b,d}),
  (<a,b>,{a,c,d}), (<a,b>,{b,c,d}), (<a,b>,{a,b,c,d}), (<c>,{}),
  (<c>,{a}), (<a>,{b}), (<a>,{c}), (<c>,{a,b}), (<c>,{a,c}),
  (<c>,{b,c}), (<c>,{a,b,c}), (<c,d>,{}), (<c,d>,{a}), (<c,d>,{b}),
  (<c,d>,{c}), (<c,d>,{d}), (<c,d>,{a,b}), (<c,d>,{a,c}), (<c,d>,
  {a,d}), (<c,d>,{b,c}), (<c,d>,{b,d}), (<c,d>,{c,d}),
  (<c,d>,{a,b,c}), (<c,d>,{a,b,d}), (<c,d>,{a,c,d}), (<c,d>,{b,c,d}),
  (<c,d>,{a,b,c,d}) }
```

Example

failures(EXT) = {
(<>, {}), , (<>,{b}), , (<>,{d}), ,
, , , (<>,{b,d}), ,
, , , ,
(<a>,{}), (<a>,{a}), (<a>,{c}), (<a>,{d}), (<a>,{a,c}),
(<a>,{a,d}), (<a>,{c,d}), (<a>,{a,c,d}), (<a,b>,{}),
(<a,b>,{a}), (<a,b>,{b}), (<a,b>,{c}), (<a,b>,{d}),
(<a,b>,{a,b}), (<a,b>,{a,c}), (<a,b>,{a,d}), (<a,b>,{b,c}),
(<a,b>,{b,d}), (<a,b>,{c,d}), (<a,b>,{a,b,c}), (<a,b>,{a,b,d}),
(<a,b>,{a,c,d}), (<a,b>,{b,c,d}), (<a,b>,{a,b,c,d}), (<c>,{}),
(<c>,{a}), (<a>,{b}), (<a>,{c}), (<c>,{a,b}), (<c>,{a,c}),
(<c>,{b,c}), (<c>,{a,b,c}), (<c,d>,{}), (<c,d>,{a}), (<c,d>,{b}),
(<c,d>,{c}), (<c,d>,{d}), (<c,d>,{a,b}), (<c,d>,{a,c}), (<c,d>,
{a,d}), (<c,d>,{b,c}), (<c,d>,{b,d}), (<c,d>,{c,d}),
(<c,d>,{a,b,c}), (<c,d>,{a,b,d}), (<c,d>,{a,c,d}), (<c,d>,{b,c,d}),
(<c,d>,{a,b,c,d}) }

Recursion

The semantics of a recursively-defined process is the least fixed point of the function corresponding to the defining equation.

The least fixed point exists if every instance of the process name on the right of the equation is preceded by at least one event.

you can’t get to the next invocation without performing at least one transaction

Traces

If `f` is the function upon sets of traces corresponding to the defining equation

```
P = F(P)
```

then

```
traces(P) =
  Union({apply f n traces(STOP) | n <- Num})
```

where

```
apply f 0 s = s
apply f (k+1) s = apply f k (apply f s)
```

Failures

If `f` is the function upon sets of failures corresponding to the defining equation

```
P = F(P)
```

then

```
failures(P) =
  Inter({apply f n failures(CHAOS) | n <- Num})
```

where CHAOS is a process that can perform any sequence of events and refuse any set of events.

this is a simplified version of the actual semantics

Example

If

```
P = a -> b -> P
```

then

```
f(S) = union({<>,<a>},{<a,b>^t | t <- S})
```

and

```
traces(P) = { <>,<a>,<a,b>,<a,b,a>,<a,b,a,b,a>, ... }
```

Definedness

If there is an instance of the process name on the right not preceded by an event, then there is no unique fixed point and the process is not properly defined.

Traces and failures do not tell us about definedness.

We need to add another (final!) component to our semantics.

Divergences

A divergence is a trace of a process after which we have no further information: the behaviour of the process is no longer well defined.

If `tr` is a divergence of a process `P`, then so is any extension `tr^<a>`: we cannot recover from divergence.

Example

```
REC = a -> b -> REC

divergences(REC) = {}
```

Example

```
DIV = DIV

divergences(DIV) = { s | s <- seq(Event) }
```

Example

```
DIVA = a -> b -> DIV

divergences(DIVA) = { <a,b>^s | s <- seq(Event) }
```

Example

```
DIVB = a -> b -> DIVB
       []
       DIVB

divergences(DIVB) = { s | s <- seq(Event) }
```

Semantics

A process is completely characterised by its failures and divergences.

Two processes are equivalent if they have exactly the same set of failures, and exactly the same set of divergences.

Summary

- Process language
- Traces
- Failures
- Divergences

Index

16 Convexity

17 Example

18 Stop

19 Traces: stop

20 Event prefix

21 Interpretation: prefix

22 Traces: prefix

23 Example

24 External choice

25 Interpretation: external choice

26 Traces: external choice

27 Example

28 Definition