CLOUT (Computational Logic for Use in Teaching) with LPS (Logic-based Production Systems)

Robert Kowalski and Fariba Sadri
Imperial College London

Miguel Calejo
Interprolog.com

Outline:

The Goal

To reconcile and combine computational and logical thinking

The Problem

The Solution

CLOUT (Computational Logic for Use in Teaching)

Computational thinking

Logical thinking

Algorithmic thinking
using state transitions

Abstraction

Goals and beliefs

Problem decomposition
by backwards reasoning

Top down and bottom up reasoning
(= analysis and synthesis)

Outline:

The Goal

To reconcile and combine computational and logical thinking

The Problem

Two kinds of systems

The Solution

Two kinds of programming systems

STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

David HAREL

For transformational systems (e.g., many kinds of data-processing systems) one really has to specify a transformation, or function, so that an input/output relation is usually sufficient. While transformational systems can also be highly complex, there are several excellent methods that allow one to decompose the system's transformational behavior into ever-smaller parts in ways that are both coherent and rigorous. Many of these approaches are supported by languages and implemented tools that perform very well in practice. We are of the opinion that for reactive systems, which present the more difficult cases, this problem has not yet been satisfactorily solved. Several important and promising approaches have been proposed, and Section 8 of this paper discusses a number of them. However, the

e.g. logic programs

e.g. production systems

Science of Computer Programming 8 (1987) 231–274
North-Holland

STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

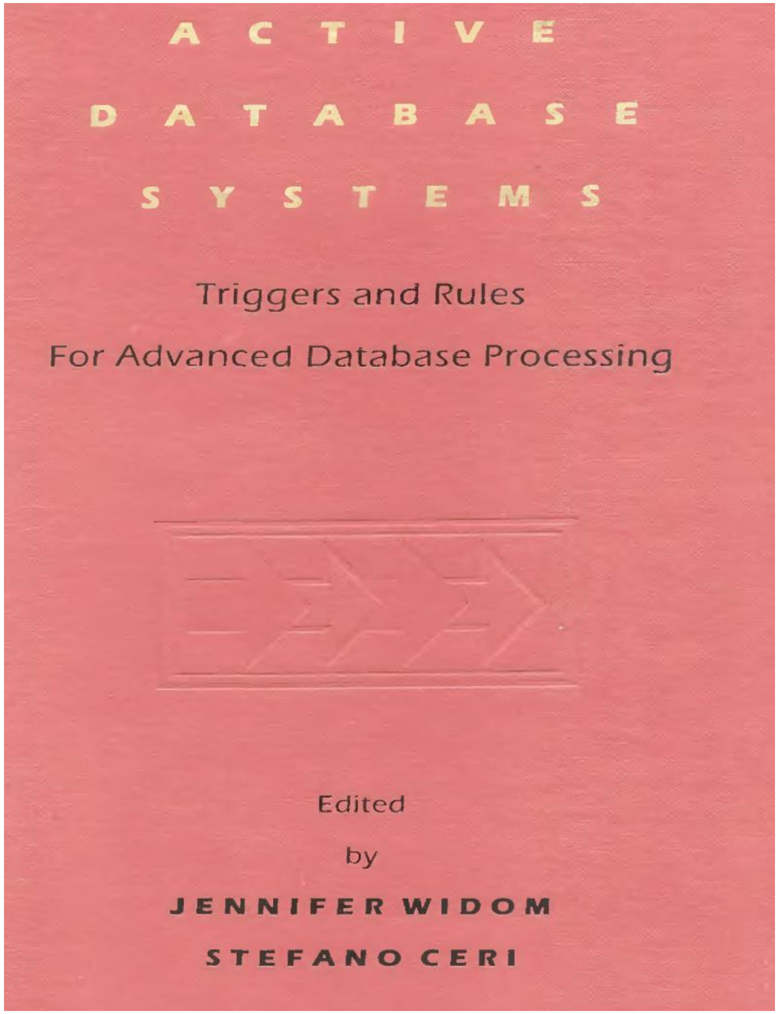
David HAREL

Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel

Much of the literature also seems to be in agreement that states and events are *a priori* a rather natural medium for describing the dynamic behavior of a complex system. See, for example, [7–9, 19, 23]. A basic fragment of such a description is a *state transition*, which takes the general form "when event α occurs in state A, if condition C is true at the time, the system transfers to state B". Indeed, many of the informal exchanges concerning the dynamics of systems are of this nature; e.g., "when the plane is in cruise mode and switch x is thrown it enters navigate mode",

reactive rule

Two kinds of database systems: Active Databases and Deductive Databases (e.g. Datalog)



In: The Knowledge Engineering Review, vol. 8 no. 2, pages 121–143, June 1993

An Overview of Production Rules in Database Systems

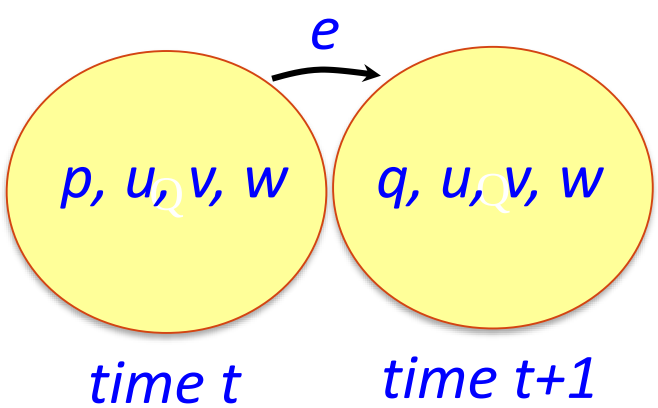
Eric N. Hanson

Jennifer Widom

Database researchers have discovered that with the addition of production rules facilities, database systems gain the power to perform a number of useful database tasks with one uniform mechanism: they can enforce integrity constraints, monitor data access and evolution, maintain derived data, enforce protection schemes, maintain version histories, and more. (Previous support

There is a substantial body of work on another kind of database system with rules—deductive database systems. Deductive database systems are similar to conventional database systems in that they are passive, responding only to commands from users or applications. However, they extend conventional database systems by allowing the definition of PROLOG-like rules on the data and by providing a deductive inference engine for processing recursive queries using these rules. Deductive and active database rule systems are fundamentally different, and both types of rules could theoretically be present in a single system. We focus on active database systems and do

The Problem: Conventional logical languages are not computationally feasible



e initiates q
e terminates p

It is necessary to reason that
u is true at time *t+1* because *u* was true at time *t* and *u* was not terminated from *t* to *t+1.*

v is true at time *t+1* because *v* was true at time *t* and v was not terminated from *t* to *t+1.*

w is true at time *t+1* because *w* was true at time *t* and *w* was not terminated from *t* to *t+1.*

The Problem: Imperative languages do not have a logical meaning

if A then B means change of state. e.g.
If A holds then do B. ("imperative")

Programming
state charts
abstract state machines

Databases
active databases

AI
production systems
agent languages

if A then B
does not have a
logical meaning

States change
destructively.

Production systems do not have a logical meaning

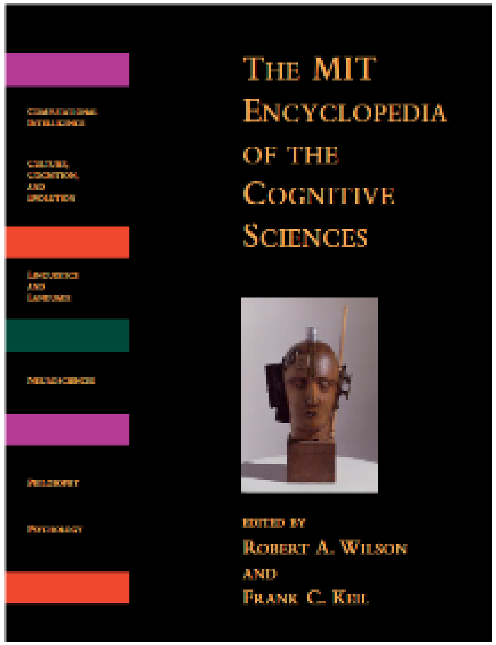
Production Systems —*Herbert A. Simon*

Production systems are computer languages that are widely employed for representing the processes that operate in models of cognitive systems (NEWELL and Simon 1972).

In a production system, all of the instructions (called productions) take the form:

IF<<conditions>, THEN<<actions>,

That is to say, "if certain conditions are satisfied, then take the specified actions" (abbreviated C → A). Production sys-

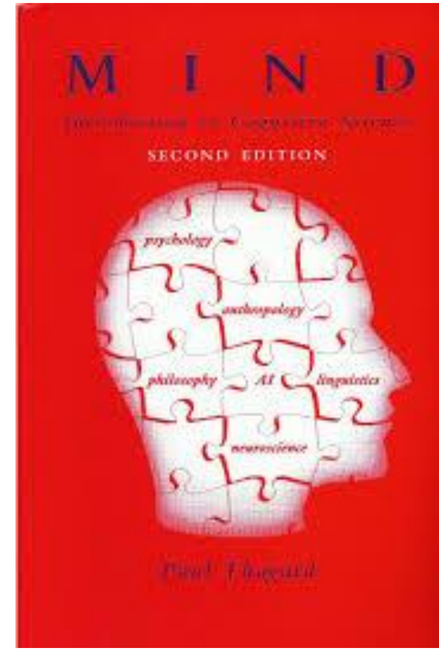


Production systems do not have a logical meaning

fire $\Rightarrow$ *deal-with-fire*
deal-with-fire $\Rightarrow$ *eliminate*
deal-with-fire $\Rightarrow$ *escape*

Adding *fire* to working memory.
Triggers two candidate actions *eliminate* and *escape*.
Conflict resolution decides between them.

Production rules and logic programs:
It can be hard to tell the difference.



"Rules are if-then structures...
very similar to the conditionals... (of logic)
but they have very different
representational and computational properties."

Reactive rules and logic programs: It can be hard to tell the difference

Unlike logic, rule-based systems can also easily represent strategic information about what to do. Rules often contain actions that represent goals, such as *IF you want to go home for the weekend, and you have bus fare, THEN you can catch a bus*. Such information about goals serves to focus the rule-

This production rule in Thagard's Mind
is a logic program (or belief) in LPS:

You go home from T1 to T2
if *you have the bus fare at T1,*
you catch a bus from T1 to T2.

(combined with backward reasoning)

Reactive rules and logic programs:
It can be hard to tell the difference

AgentSpeak(L): BDI Agents speak out in a logical computable language

Definition 5 If e is a triggering event, $b_1,\ldots,b_m$ are belief literals, and $h_1,\ldots,h_n$ are goals or actions then $e{:}b_1 \wedge \ldots \wedge b_m \leftarrow h_1;\ldots;h_n$ is a *plan*. The expression to the left of the arrow is referred to as the *head* of the plan and the expression to the right of the arrow is referred to as the *body* of the plan. The expression to the right of the colon in the head of a plan is referred to as the *context*. For convenience, we shall rewrite an empty body with the expression *true*.

With this we complete the specification of an agent. In summary, a designer specifies an agent by writing a set of base beliefs and a set of plans. This is similar to a logic programming specification of facts and rules. However, some of the major differences between a logic

LPS and BDI agents compared

This “logic programming-like” plan in AgentSpeak

```
+location(waste,X):location(robot,X) &
                   location(bin,Y)
                   <- pick(waste);
                      !location(robot,Y);
                      drop(waste).              (P1)
```

is a reactive rule (or goal) in LPS:

if *location(waste, X) at T1, location(robot, X) at T1, location(bin, Y) atT1*
then *pick(waste) fromT1 to T2,*
move-to-location(robot, Y) fromT2 to T3,
drop(waste) from T3 to T4.

Goals and Beliefs:
It can be hard to tell the difference.

All humans are mortal.
All humans are kind.

Goals: *if human(X) then mortal(X).*
if human(X) then kind(X).

or

Beliefs: *mortal(X) if human(X).*
kind(X) if human(X).

Outline:

The Goal

To reconcile and combine computational and logical thinking

The Problem

Two kinds of systems

The Solution

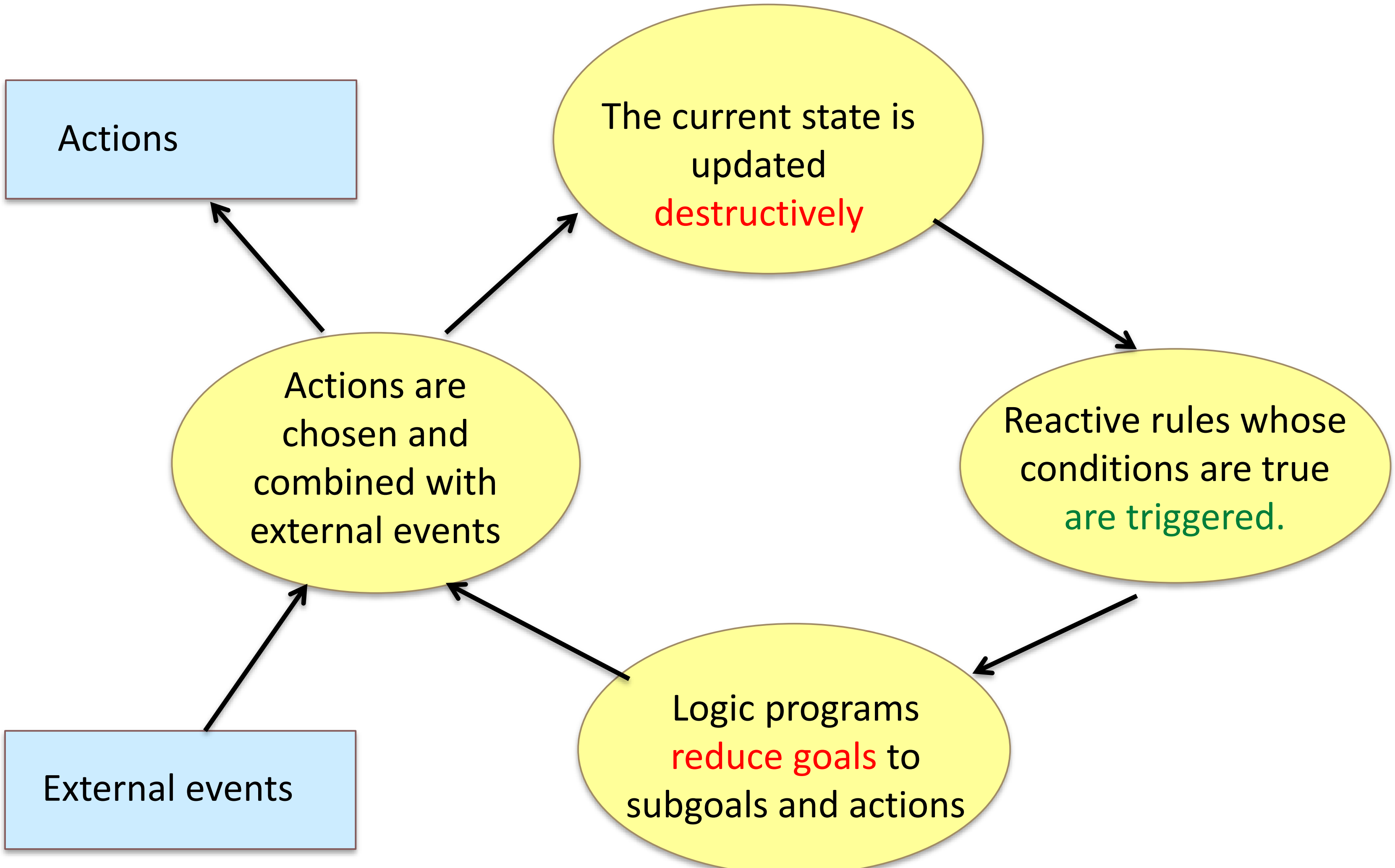
Goals and Beliefs
Model generation
(with explicit representation of events and time)

LPS: Computation = Model Generation

Computation executes actions
to generate a world model
to make goals true.

A world model is the minimal model of a
logic program describing beliefs about
states, actions, external events,
intentional predicates, and
complex events and plans.

LPS: Computation generates actions to make reactive rules true



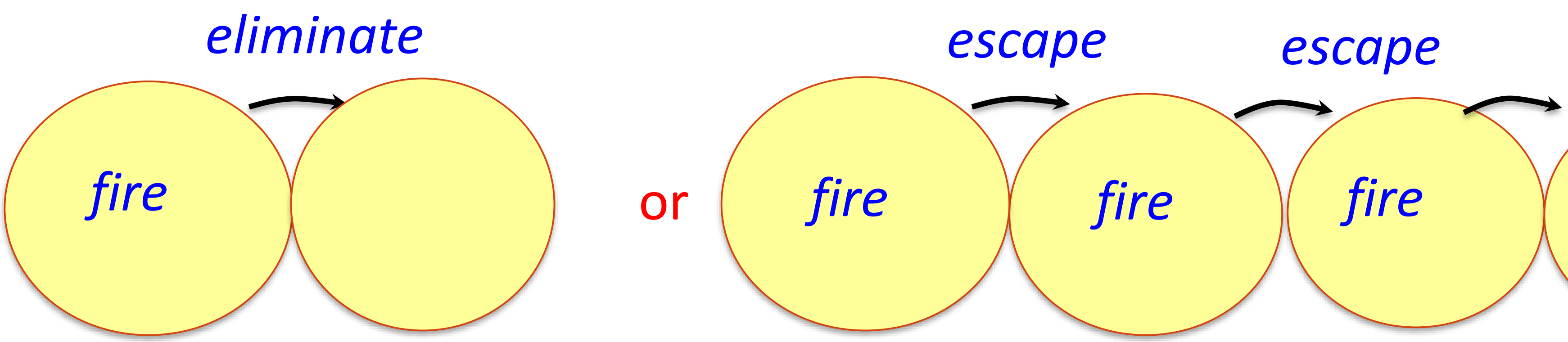
LPS combines reactive rules, logic programs and causal laws

Reactive rule: *if fire then deal-with-fire.*

Logic program: *deal-with-fire if eliminate.*
deal-with-fire if escape.

Causal law: *eliminate terminates fire.*

Adding *fire* to the current state.
Generates two alternative actions *eliminate* or *escape*.
Generates alternative sequences of states
to make the reactive rule true:



World models are sequences of states, actions and external events, described by atomic sentences

without time stamps
for efficiency

with time stamps
for logical semantics

States are sets of facts (or fluents):

fire

fire(10:15)

Events (including actions) cause state transitions:

eliminate

eliminate(10:15, 10:16)

The syntax of LPS

Reactive rules in First-order logic:

for all X [*antecedent* $\rightarrow$ *there exists Y consequent*]

or *if antecedent then consequent.*

Clauses in logic programming form:

for all X [*there exists Y conditions* $\rightarrow$ *conclusion*]

or *conclusion if conditions.*

The syntax of LPS

without time stamps
for readability

Reactive rules:

if *fire*
then *deal-with-fire.*

Logic programs:

deal-with-fire
if *eliminate.*

with time stamps
for logical semantics

if *fire at T1*
then deal-with-fire fromT2 to T3,
T1≤T2.

deal-with-fire from T1 to T2
if *eliminate fromT1 to T2.*

State transitions are described by a “programmable” causal theory

Postconditions (effects):

ignite(Object) initiates fire if flammable(Object).

eliminate terminates fire.

Preconditions (constraints):

false eliminate, fire, not water.

Persistence (inertia):

Fact/fluents persist without needing to reason that they persist.

```
% Fire example with keywords in blue.

fluents   fire.
actions   eliminate, escape.
events    deal_with_fire.

initially fire.

if        fire at T1
then      deal_with_fire from T1 to T2.

deal_with_fire from T1 to T2
if        eliminate from T1 to T2.

deal_with_fire from T1 to T2
if         escape from T1 to T2.

eliminate  terminates fire.
```

```
maxTime(10).
fluents     fire, water.
actions     eliminate, ignite(_), escape, refill.

observe   ignite(sofa)    from 1 to 2.
observe   ignite(bed)     from 4 to 5.
observe   refill          from 7 to 8.

initially   water.

flammable(sofa).
flammable(bed).

if          fire at T1
then        deal_with_fire from T2 to T3.

deal_with_fire            from T1 to T2
if          eliminate     from T1 to T2.

deal_with_fire            from T1 to T2
if          escape        from T1 to T2.

ignite(Object)  initiates   fire  if     flammable(Object).

eliminate       terminates fire.
eliminate       terminates water.
refill          initiates water.

false           eliminate, fire, not water.
```

The Dining Philosophers

```
maxTime(7).
fluents   available(_).
actions   pickup(_,_), putdown(_,_).

initially available(fork1),  available(fork2), available(fork3),
          available(fork4),  available(fork5).

philosopher(socrates).
philosopher(plato).
philosopher(aristotle).
philosopher(hume).
philosopher(kant).

adjacent(fork1, socrates, fork2).
adjacent(fork2, plato, fork3).
adjacent(fork3, aristotle, fork4).
adjacent(fork4, hume, fork5).
adjacent(fork5, kant, fork1).
```

```
% dining philosophers

if      philosopher(P)
then    dine(P) from T1 to T2.

dine(P) from T1 to T3  if
        adjacent(F1, P, F2),
        pickup(P, F1) from T1 to T2,
        pickup(P, F2) from T1 to T2,
        putdown(P, F1) from T2 to T3,
        putdown(P, F2) from T2 to T3 .

pickup(P, F)      terminates    available(F).
putdown(P, F)     initiates     available(F).

false   pickup(P, F),    not available(F).
false   pickup(P1, F),  pickup(P2, F), P1 \= P2.
```

What happens if we replace:

```
dine(P) from T1 to T3  if
      adjacent(F1, P, F2),
      pickup(P, F1) from T1 to T2,
      pickup(P, F2) from T1 to T2,
      putdown(P, F1) from T2 to T3,
      putdown(P, F2) from T2 to T3.
```

with:

```
dine(P) from T1 to T5  if
      adjacent(F1, P, F2),
      pickup(P, F1) from T1 to T2,
      pickup(P, F2) from T2 to T3,
      putdown(P, F1) from T3 to T4,
      putdown(P, F2) from T4 to T5.
```

Conclusions

LPS combines
computational thinking and
logical thinking.

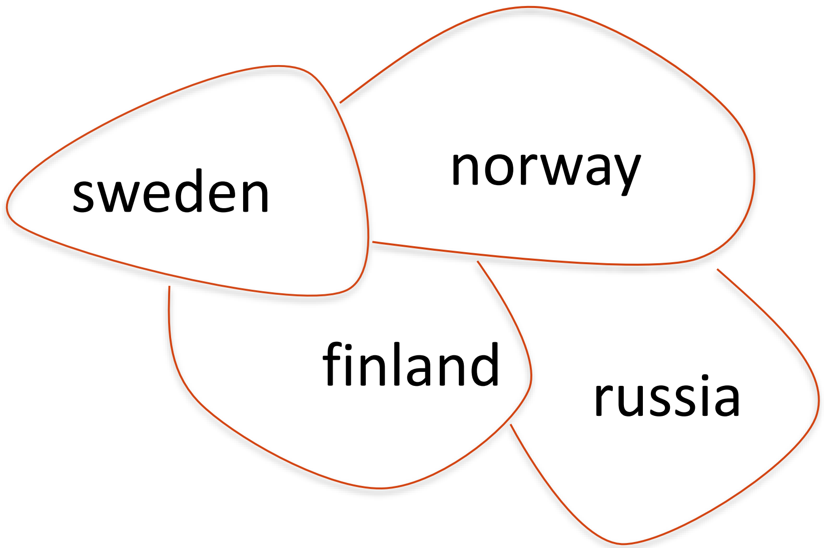
LPS is a practical, logical framework for computing.

LPS is not a full-scale framework for intelligent thinking,
but it can be extended.

```
maxTime(7).
fluents        available(_).
actions        pickup(_,_), putdown(_,_).

initiallyavailable(fork1), available(fork2), available(fork3), available(fork4), available(fork5).

philosopher(socrates).
philosopher(plato).
philosopher(aristotle).
philosopher(hume).
philosopher(kant).

adjacent(fork1, socrates, fork2).
adjacent(fork2, plato, fork3).
adjacent(fork3, aristotle, fork4).
adjacent(fork4, hume, fork5).
adjacent(fork5, kant, fork1).

if      philosopher(P)
then    dine(P) from T1 to T2.

dine(P) from T1 to T3  if
        adjacent(F1, P, F2),
        pickup(P, F1) from T1 to T2,
        pickup(P, F2) from T1 to T2,
        putdown(P, F1) from T2 to T3,
        putdown(P, F2) from T2 to T3 .

pickup(P, F)     terminates     available(F).
putdown(P, F)  initiates         available(F).

false   pickup(P, F),    not available(F).
false   pickup(P1, F),  pickup(P2, F), P1 \= P2.
```

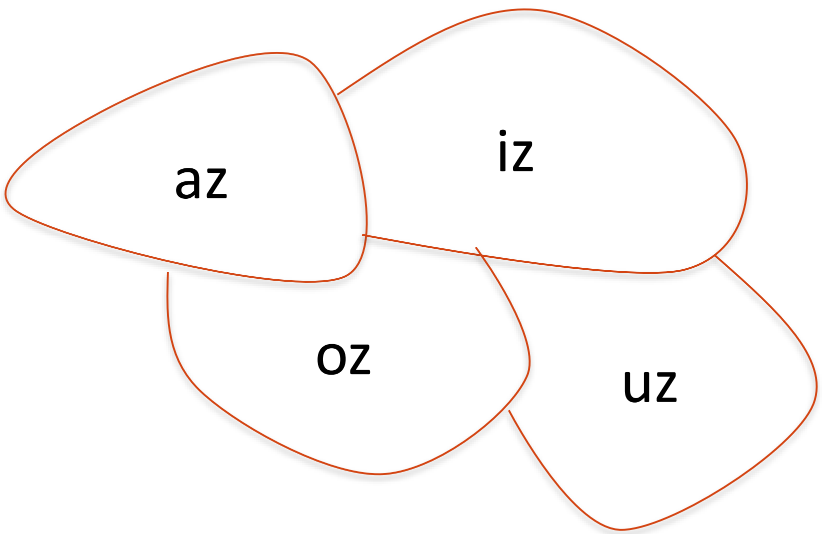
% The map colouring problem.

```
maxTime(5).
actions paint(_, _).

country(sweden).
country(norway).
country(finland).
country(russia).
colour(red).
colour(yellow).
colour(blue).
adjacent(sweden, norway).
adjacent(sweden, finland).
adjacent(norway, finland).
adjacent(norway, russia).
adjacent(finland, russia).
```



% The map colouring problem.

```
maxTime(5).
actions paint(_, _).

 country(iz).
 country(oz).
 country(az).
 country(uz).
 colour(red).
 colour(yellow).
 colour(blue).
 adjacent(az, iz).
 adjacent(az, oz).
 adjacent(iz, oz).
 adjacent(iz, uz).
 adjacent(oz, uz).
```

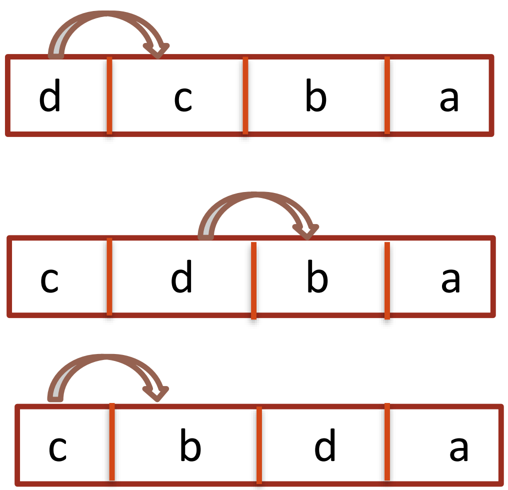


```
% The map colouring problem

% For every country X, there exists a colour C.

if      country(X)
then    colour(C), paint(X, C) from 1 to 2.

% Two adjacent countries cannot be painted the same colour.

false   paint(X, C), adjacent(X, Y), paint(Y, C).


/* We can also write
if      country(X)
then    colour(C), paint(X, C) from T1 to T2.
*/
```

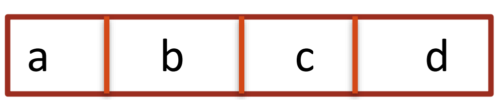
```
% The map colouring problem.

maxTime(5).
actions paint(_, _).

 country(iz).
 country(oz).
 country(az).
 country(uz).
 colour(red).
 colour(yellow).
 colour(blue).
 adjacent(az, iz).
 adjacent(az, oz).
 adjacent(iz, oz).
 adjacent(iz, uz).
 adjacent(oz, uz).

if     country(X)
then  colour(C), paint(X, C) from 1 to 2.

false        paint(X, C), adjacent(X, Y), paint(Y, C).
```

Bubble sort

Keep swapping adjacent elements that are out of order until the array is ordered.



And so on .....



```
% bubble sort with relational data structure .
maxTime(5).
fluents     location(_, _).
actions     swap(_,_,_,_).
initially   location(d, 1), location(c, 2), location(b, 3),  location(a,4).

if      location(X, N1) at T1, N2 is N1 +1,  location(Y, N2) at T1,  Y@<X
then    swapped(X, N1, Y, N2) from T2 to T3.

% swapped may not work if the order of the two clauses below is
% reversed. Perhaps for good reasons.

swapped(X, N1, Y, N2) from T1 to T2
if      location(X, N1) at T1, location(Y, N2) at T1,
        Y@<X, swap(X, N1, Y, N2) from T1 to T2.

swapped(X, N1, Y, N2) from T to T
if      location(X, N1) at T, location(Y, N2) at T, X@<Y.

swap(X, N1, Y, N2)   initiates     location(X, N2).
swap(X, N1, Y, N2)   initiates     location(Y, N1).

swap(X, N1, Y, N2)   terminates    location(X, N1).
swap(X, N1, Y, N2)   terminates    location(Y, N2).

false       swap(X, N1, Y, N2), swap(Y, N2, Z, N3).
```

LPS executes actions concurrently

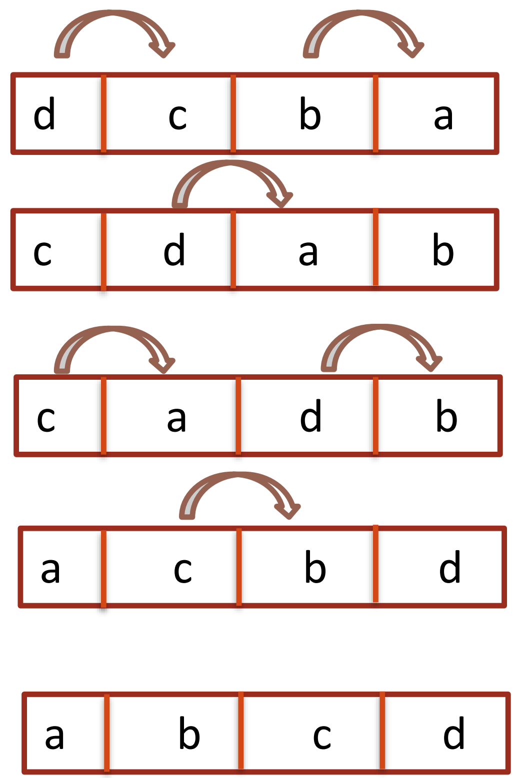
Time 1

Time 2

Time 3

Time 4

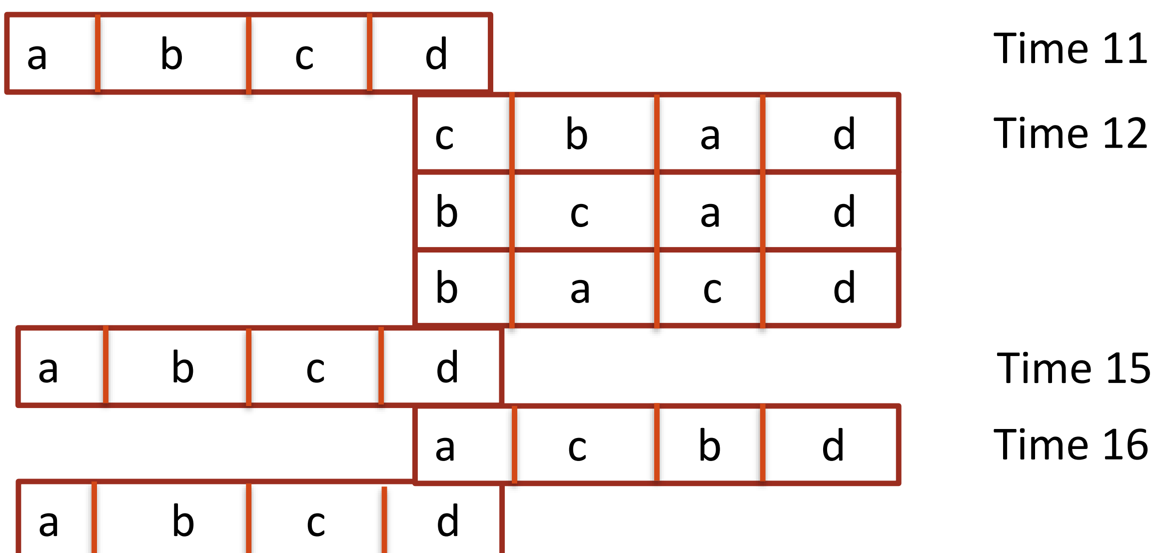
Time 5



Teleo-reactivity

If later an object is moved, the same program will sort them again.

observe swap(a,1,c,3) from 11 to 12.
observe swap(b,2,c,3) from 15 to 16.



```
maxTime(20).
fluents     location(_, _).
actions     swap(_,_,_,_).
observe     swap(a,1,c,3) from 11 to 12.  %new
observe     swap(b,2,c,3) from 15 to 16.  %new
initially   location(d, 1), location(c, 2), location(b, 3),  location(a,4).

if          location(X, N1) at T1, N2 is N1 +1,  location(Y, N2) at T1,  Y@<X
then        swapped(X, N1, Y, N2) from T2 to T3.

% swapped may not work if the order of the two clauses below is
% reversed. Perhaps for good reasons.
%
swapped(X, N1, Y, N2) from T1 to T2
if      location(X, N1) at T1, location(Y, N2) at T1,
        Y@<X, swap(X, N1, Y, N2) from T1 to T2.

swapped(X, N1, Y, N2) from T to T
if      location(X, N1) at T, location(Y, N2) at T, X@<Y.

swap(X, N1, Y, N2)   initiates      location(X, N2).
swap(X, N1, Y, N2)   initiates      location(Y, N1).
swap(X, N1, Y, N2)   terminates     location(X, N1).
swap(X, N1, Y, N2)   terminates     location(Y, N2).
false       swap(X, N1, Y, N2), swap(Y, N2, Z, N3).
```

```
% bankTransfer

maxTime(9).
actions        transfer(From, To, Amount).
fluents        balance(Person, Amount).

initially      balance(bob, 0), balance(fariba, 100).
observe        transfer(fariba, bob, 10)    from 0 to 1.

if       transfer(fariba, bob, X)      from  T1 to T2
then     transfer(bob, fariba, 10)     from T2 to T3.

if       transfer(bob, fariba, X)      from  T1 to T2
then     transfer(fariba, bob, 20)     from  T2 to T3.
```

% bankTransfer – the Causal Theory.

```
transfer(From, To, Amount)      initiates       balance(To, New)
if          balance(To, Old),  New is Old + Amount.

transfer(From, To, Amount)      terminates    balance(To, Old).

transfer(From, To, Amount)      initiates       balance(From, New)
if          balance(From, Old),  New is Old - Amount.

transfer(From, To, Amount)      terminates    balance(From, Old).

false       transfer(From, To, Amount), balance(From, Old),  Old < Amount.

false       transfer(From, To1, Amount1),
            transfer(From, To2, Amount2),  To1 \=To2.

false       transfer(From1, To, Amount1),
            transfer(From2, To, Amount2),  From1 \= From2.
```

% bankTransfer

```
maxTime(9).
actions         transfer(From, To, Amount).
fluents         balance(Person, Amount).

initially       balance(bob, 0), balance(fariba, 100).
observe         transfer(fariba, bob, 10)  from 0 to 1.

if       transfer(fariba, bob, X)    from  T1 to T2
then     transfer(bob, fariba, 10)  from T2 to T3.

if       transfer(bob, fariba, X)    from  T1 to T2
then     transfer(fariba, bob, 20)  from  T2 to T3.

transfer(From, To, Amount)initiates     balance(To, New)
if              balance(To, Old),  New is Old + Amount.

transfer(From, To, Amount) terminates balance(To, Old).

transfer(From, To, Amount) initiates    balance(From, New)
if              balance(From, Old),  New is Old - Amount.

transfer(From, To, Amount)       terminates   balance(From, Old).

false   transfer(From, To, Amount), balance(From, Old),  Old < Amount.

false   transfer(From, To1, Amount1),
        transfer(From, To2, Amount2),  To1 \=To2.

false   transfer(From1, To, Amount1),
        transfer(From2, To, Amount2),  From1 \= From2.
```

Natural language grammars can be represented by logic programs

sentence -> nounphrase, verbphrase
nounphrase -> adjective, noun
nounphrase -> noun
verbphrase -> verb, nounphrase
verbphrase -> verb
adjective -> my
adjective -> your
noun -> name
noun -> what
noun -> bob
verb -> is

-> is the opposite of logical *if*.

% sentences as complex events and as complex plans

```
maxTime(10).

observe   say(turing, what)    from 0 to 1.
observe   say(turing, is)      from 1 to 2.
observe   say(turing, your)    from 2 to 3.
observe   say(turing, name)    from 3 to 4.

if     saying(turing, sentence) from T1 to T2
then saying(robot, sentence) from T3 to T4.

saying(Agent, sentence)               from T1 to T3  if
       saying(Agent, nounphrase)      from T1 to T2,
       saying(Agent, verbphrase)      from T2 to T3.
```

```
saying(Agent, nounphrase)      from T1 to T3  if
      saying(Agent, adjective)  from T1 to T2,
      saying(Agent, noun)  from T2 to T3.

saying(Agent, nounphrase)      from T1 to T2  if
      saying(Agent, noun)       from T1 to T2.

saying(Agent, verbphrase)      from T1 to T3  if
      saying(Agent, verb)       from T1  to T2,
      saying(Agent, nounphrase) from T2 to T3.

saying(Agent, verbphrase)      from T1 to T2   if
      saying(Agent, verb)       from T1 to T2.

saying(Agent, adjective) from T1 to T2  if      say(Agent, my) from T1 to T2.
saying(Agent, adjective) from T1 to T2 if       say(Agent, your) from T1 to T2.

saying(Agent, noun) from T1 to T2   if     say(Agent, name) from T1 to T2.
saying(Agent, noun) from T1 to T2   if     say(Agent, what) from T1 to T2.
saying(Agent, noun) from T1 to T2   if     say(Agent, bob) from T1 to T2.

saying(Agent, verb) from T1 to T2   if     say(Agent, is) from T1 to T2.
```

```
fluents  said(_,_).
actions say(_,_).

initially    said(turing, []), said(robot, []).

say(Agent, Word)  initiates      said(Agent, NewPhrase)
if              said(Agent, OldPhrase),
                append(OldPhrase, [Word], NewPhrase).

say(Agent, Word) terminates  said(Agent, OldPhrase)
if               said(Agent, OldPhrase).

false         say(Agent, Word1),
              say(Agent, Word2),
              Word1 \= Word2.
```

```
maxTime(10).

fluents  said(_,_).
actions say(_,_).

observe      say(turing, what)      from 0 to 1.
observe      say(turing, is)  from 1 to 2.
observe      say(turing, your)      from 2 to 3.
observe      say(turing, name)      from 3 to 4.

if      saying(turing, sentence)      from T1 to T2  then saying(robot, sentence) from T3 to T4.

saying(Agent, sentence)      from T1 to T3  if      saying(Agent, nounphrase)    from T1 to T2,
                                                    saying(Agent, verbphrase)    from T2 to T3.

saying(Agent, nounphrase)    from T1 to T3  if      saying(Agent, adjective)     from T1 to T2,
                                                    saying(Agent, noun)    from T2 to T3.

saying(Agent, nounphrase)    from T1 to T2  if      saying(Agent, noun)          from T1 to T2.

saying(Agent, verbphrase)    from T1 to T3  if      saying(Agent, verb)    from T1  to T2,
                                                    saying(Agent, nounphrase) from T2 to T3.
saying(Agent, verbphrase)    from T1 to T2   if     saying(Agent, verb)          from T1 to T2.

saying(Agent, adjective) from T1 to T2       if     say(Agent, my) from T1 to T2.
saying(Agent, adjective) from T1 to T2       if     say(Agent, your) from T1 to T2.

saying(Agent, noun) from T1 to T2            if     say(Agent, name) from T1 to T2.
saying(Agent, noun) from T1 to T2            if     say(Agent, what) from T1 to T2.
saying(Agent, noun) from T1 to T2            if     say(Agent, bob) from T1 to T2.
saying(Agent, verb) from T1 to T2            if     say(Agent, is) from T1 to T2.

initially        said(turing, []), said(robot, []).
say(Agent, Word)  initiates    said(Agent, NewPhrase)      if     said(Agent, OldPhrase),  append(OldPhrase, [Word], NewPhrase).
say(Agent, Word) terminates  said(Agent, OldPhrase)        if      said(Agent, OldPhrase).
false            say(Agent, Word1),  say(Agent, Word2),    Word1 \= Word2.
```