LPS (Logic-based Production System)

Robert Kowalski and Fariba Sadri

Imperial College London

Incomplete, but usable implementation under development at:

https://bitbucket.org/lpsmasters/lps_corner

with acknowledgements to Miguel Calejo
LPS (Logic-based Production System)

A logic and computer language for practical programming
databases
AI knowledge representation and problem solving
Outline:

- LPS gives a logical semantics to production systems
- LPS combines reactive rules with logic programs
- The Turing test in LPS
- Bank account transfer in LPS
- Dining philosophers in XSB/Studio implementation of LPS
- CLOUT (Computational Logic for Use in Teaching in Schools)
Production Systems

Combine states described by a working memory of facts. State transitions represented by condition-action rules.

Popular for implementing expert systems. As a computational model of human thinking (e.g. SOAR, ACT, Steven Pinker’s *How the Mind Works*)
Production systems do not have a logical semantics.

\[
\begin{align*}
\text{threat} & \Rightarrow \text{deal-with-threat} \\
\text{fire} & \Rightarrow \text{threat} \\
\text{flood} & \Rightarrow \text{threat} \\
\text{deal-with-threat} & \Rightarrow \text{eliminate} \\
\text{deal-with-threat} & \Rightarrow \text{escape}
\end{align*}
\]

Adding \textit{fire} to working memory triggers two candidate actions \textit{eliminate} and \textit{escape}. Conflict resolution decides between them.
LPS combines logic programs and FOL reactive rules

Reactive rule: \( \text{threat} \rightarrow \text{deal-with-threat} \)

Logic program: \( \text{threat} \leftarrow \text{fire} \)
\( \text{threat} \leftarrow \text{flood} \)
\( \text{deal-with-threat} \leftarrow \text{eliminate} \)
\( \text{deal-with-threat} \leftarrow \text{escape} \)

Adding \text{fire} to the current state generates two alternative actions \text{eliminate} or \text{escape}. LPS generates one model to make the reactive rule true.

\[ \text{M1} = \{\text{fire, threat, deal-with-threat, eliminate}\} \text{ or } \text{M2} = \{\text{fire, threat, deal-with-threat, escape}\} \]

The current implementation in Prolog generates M1.
LPS combines logic programs with reactive rules in FOL.

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
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 $\alpha$ 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
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
LPS combines logic programs and FOL reactive rules

Reactive rules in FOL:

\[ \forall X [\text{antecedent} \rightarrow \exists Y [\text{consequent}]] \]

or

\[ \text{antecedent} \rightarrow \text{consequent} \]

or

\[ \text{if antecedent then consequent} \]

Clauses in logic programming form:

\[ \forall X [\text{conditions} \rightarrow \text{conclusion}] \]

or

\[ \text{conclusion} \leftarrow \text{conditions} \]

or

\[ \text{conclusion if conditions} \]

Atomic sentences are a special case of clauses.

The syntax of LPS is fluid.
The Turing Test
(not completely catered for in the current implementation)

\[ \text{sentence}(turing, T_1, T_2) \rightarrow \text{sentence}(robot, T_3, T_4) \land T_2 < T_3 < T_2 + 3 \text{ sec} \]

\[ \text{sentence}(Agent, T_1, T_3) \leftarrow \text{noun-phrase}(Agent, T_1, T_2) \land \text{verb-phrase}(Agent, T_2, T_3) \]

\[ \text{noun-phrase}(Agent, T_1, T_3) \leftarrow \text{adjective}(Agent, T_1, T_2) \land \text{noun}(Agent, T_2, T_3) \]

\[ \text{adjective}(Agent, T_1, T_2) \leftarrow \text{say}(Agent, your, T_1, T_2) \]

\[ \text{noun}(Agent, T_1, T_2) \leftarrow \text{say}(Agent, name, T_1, T_2) \]

etc.
The same clauses can be used to recognise complex events and to generate complex plans.

**Observed events:**

- `say(turing, what, 1, 2)`
- `say(turing, is, 2, 3)`
- `say(turing, your, 3, 4)`
- `say(turing, name, 4, 5)`

**Action events:**

- `say(robot, my, 7, 8)`
- `say(robot, name, 8, 9)`
- `say(robot, is, 9, 10)`
- `say(robot, bob, 10, 11)`

The actions make the reactive rule true.
States and events can be described by atomic sentences without time stamps for efficient updates.

States (sets of facts, also called fluents):

\[ balance(bob, 0) \]
\[ balance(bob, 0, 31/08/2016) \]

Events (including actions):

\[ transfer(fariba, bob, 10) \]
\[ transfer(fariba, bob, 10, 31/08/2016, 1/9/2016) \]
State transitions are performed by destructive updates without explicit timestamps.

**state at time 0:** \( \text{balance}(\text{bob}, 0) \quad \text{balance}(\text{fariba}, 100) \)

**events from time 0 to time 1:** \( \text{transfer}(\text{fariba}, \text{bob}, 10) \)

**state at time 1:** \( \text{balance}(\text{bob}, 10) \quad \text{balance}(\text{fariba}, 90) \)

**events from time 1 to time 2:** \( \text{transfer}(\text{fariba}, \text{bob}, 20) \)

**state at time 2:** \( \text{balance}(\text{bob}, 30) \quad \text{balance}(\text{fariba}, 70) \)

**events from time 2 to time 3:**

**state at time 3:** \( \text{balance}(\text{bob}, 30) \quad \text{balance}(\text{fariba}, 70) \)

etc.

Frame axioms are emergent properties, not used for computation:

\[
\text{balance}(X, N, T2) \leftarrow \text{balance}(X, N, T1) \land \\
\neg \text{transfer}(X, Y, V, T1, T2) \land \neg \text{transfer}(Y, X, W, T1, T2)
\]
State transitions are described by a causal theory with or without timestamps

Postconditions:

\[ \text{balance}(X, N-V, T2) \leftarrow \text{transfer}(X, Y, V, T1, T2) \land \text{balance}(X, N, T1) \]
\[ \text{balance}(Y, M+V, T2) \leftarrow \text{transfer}(X, Y, V, T1, T2) \land \text{balance}(Y, M, T1) \]

Preconditions:

\[ \text{not \ [transfer}(X, Y, V, T1, T2) \land \text{balance}(X, N, T1) \land V > N] \]
\[ \text{not \ [transfer}(X, Y1, V1, T1, T2) \land \text{transfer}(X, Y2, V2, T1, T2) \land Y1 \neq Y2] \]

etc.
The Dining Philosophers
Dining philosophers (in the XSB/Studio implementation)

fluent(available(Fork)).
event(time_to_eat(Philosopher)).

action(think(Philosopher)).
action(pickup_forks(Fork1, Philosopher, Fork2)).
action(eat(Philosopher)).
action(putdown_forks(Fork1, Philosopher, Fork2)).

initial_state( [ available(fork(0)),
               available(fork(1)),
               available(fork(2)),
               available(fork(3)),
               available(fork(4)) ] ).

l_timeless(adjacent(fork(1),philosopher(1),fork(2)), []).
l_timeless(adjacent(fork(3),philosopher(3),fork(4)), []).
l_timeless(adjacent(fork(0),philosopher(0),fork(1)), []).
l_timeless(adjacent(fork(2),philosopher(2),fork(3)), []).
l_timeless(adjacent(fork(4),philosopher(4),fork(0)), []).
Dining philosophers (in the XSB/Studio implementation)

time_to_eat(philosopher(N),T1,T2) --->
  dine(philosopher(N),T3,T4),  tc(T2 <= T3).

observe([time_to_eat(philosopher(0)),
  time_to_eat(philosopher(1)),
  time_to_eat(philosopher(2)),
  time_to_eat(philosopher(3)),
  time_to_eat(philosopher(4))], 1).

% currently LPS stops when no further observations exist
observe([],T) :- T > 1, T < 12.

dine(philosopher(N),T1,T6) :-
  think(philosopher(N),T1,T2),
  adjacent(F1,philosopher(N),F2),
  pickup_forks(F1,philosopher(N),F2,T3,T4),  tc(T2 <= T3),
  eat(philosopher(N),T4,T5),
  putdown_forks(F1,philosopher(N),F2,T5,T6).
Dining philosophers (causal theory)

false :- pickup_forks(F1,philosopher(N),F2,T1,T2), not available(F1,T1).
false :- pickup_forks(F1,philosopher(N),F2,T1,T2), not available(F2,T1).
false :- pickup_forks(F1,philosopher(N),F,T1,T2), pickup_forks(F,philosopher(K),F2,T1,T2).

initiated available(F1) :-
    putdown_forks(F1,philosopher(N),F2,T1,T2).
initiated available(F2) :-
    putdown_forks(F1,philosopher(N),F2,T1,T2).

terminated available(F1) :-
    pickup_forks(F1,philosopher(N),F2,T1,T2).
terminated available(F2) :-
    pickup_forks(F1,philosopher(N),F2,T1,T2).
CLOUT (Computational Logic for Use in Teaching)

A six month project (October 2016 – March 2017) to develop an open-source, web-based prototype of LPS together with motivating, modifiable examples, to support computing in schools.

Collaborators are very welcome.
Conclusions

LPS gives a logical, model-theoretic semantics for practical programming and databases.

LPS is not a full-scale AI framework, but it can be extended.