Combining Prolog and Imperative Computing in LPS

Robert Kowalski[1], Fariba Sadri[1], Miguel Calejo[2] and Jacinto Davila[3]

[1] Imperial College London, [2]logicalcontracts.com, Lisbon,
[3]Contratos Lógicos. C.A. and Universidad de Los Andes

Logic programs and imperative programs employ different notions of computation. Logic programs compute by proving that a goal is a logical consequence of the program, or by showing that the goal is true in a model defined by the program. Imperative programs compute by starting from an initial state, executing actions to transition from one state to the next, and terminating in a final state when the goal is solved.

Prolog has a “pure” logic programming (LP) kernel, extended with language features that do not have an obvious logical interpretation. Arguably, the most important of these features is the ability to assert and retract clauses, which allows Prolog to simulate destructive change of state in imperative programming languages.

In this paper, we present the language LPS (Logic Production Systems) [8-15], which combines the LP and imperative programming notions of computation and gives a restricted form of assert and retract in Prolog a logical interpretation. Computation in LPS follows the imperative paradigm of solving goals by generating a sequence of states and events, to make the goals true. States are sets of facts (called *fluents*) that change with time. Events include both external events and internally generated actions. State transitions are made by destructively asserting and retracting fluents using *causal laws* written in the LP form:

event initiates fluent if conditions.
event terminates fluent if conditions.

Operationally, when an event happens, then the causal laws are “executed” by retracting all fluents whose terminating conditions hold in the current state, and asserting all fluents whose initiating conditions hold. All other fluents persist without change. This operational behaviour can be specified by a *Causal Theory*, using LP clauses (where the integers *T* and *T+1* denote consecutive discrete times or states) such as:

```
holds(Fluent, T+1) if happens(Event, T, T+1), initiates(Event, Fluent, T+1).
holds(Fluent, T+1) if holds(Fluent, T),
                      not (happens(Event, T, T+1), terminates(Event, Fluent, T+1).
```

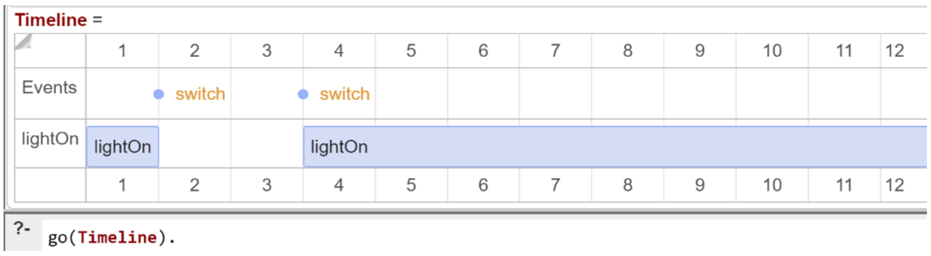
It is important to appreciate that LPS does not use this Causal Theory to reason about persistence. The Causal Theory is an emergent property that is true in the time-stamped model that is implicitly generated by the LPS computation.

As a simple example, consider the following LP clauses in LPS syntax:

```
initially lightOn.     observe switch from 1 to 2.  observe switch from 3 to 4.
lightOff if not lightOn.
switch  initiates     lightOn    if  lightOff.
switch  terminates   lightOn   if  lightOn.
```

In general, causal laws directly initiate and terminate only extensional fluents, such as *lightOn*. Intentional fluents, such as *lightOf*, are initiated and terminated only as ramifications of changes to the extensional fluents.

The SWISH [20] implementation includes a visualisation of the model generated by the computation, displaying the extensional fluent *lightOn*:



```
?- go(Timeline).
```

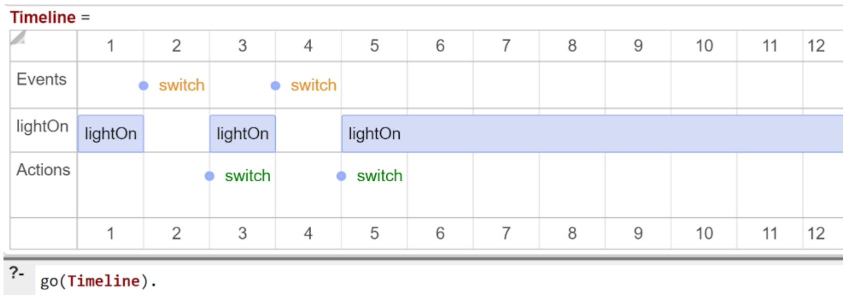
Unlike possible worlds in modal logic, models in LPS are single models in which fluents and events indicate the times (or states) for which they hold. In this example, the model can be written as a standard Herbrand interpretation:

{happens(switch, 1,2), happens(switch, 3,4), holds(lightOff, 2), holds(lightOn, 3), terminates(switch, 1,2), lighOn, 2), initiates(switch, 3, 4), lightOn, 4), holds(lightOn, 1), holds(lightOn, 4), holds(lightOn, 5), holds(lightOn, 6), ….}

In addition to causal laws, which destructively update the current state, LPS includes reactive rules of the form *if antecedent then consequent*. Logically, these are goals that need to be satisfied by generating a model that makes them true. For example:

if lightOff then switch.

LPS uses the LP component of LPS to determine when the *antecedent* of the rule becomes true, and then generates actions and executes the causal laws to make the *consequent* of the rule true. As in this example, computation can be a non-terminating process, if time is infinite and an external event can occur at any time. In this case, an initial portion of the infinite model generated by the computation can be visualised as:



```
?- go(Timeline).
```

The reactive rule above is shorthand for the sentence:

For all T1 [if holds(lightOff, T1) then
there exists T2 such that [happens(switch, T2, T2+1) and T1 ≤ T2]].

Here the switch action can be executed any time after the light is on. However, in practice, LPS generates models in which goals are satisfied as soon as possible.
In general, both the *antecedent* and *consequent* of a reactive rule can be a conjunction of (possibly negated) timeless predicates, such as the inequality relation ≤, and (possibly negated) fluents and events. All variables in the *antecedent* are universally quantified with scope the entire rule. All other variables are existentially quantified with scope the *consequent* of the rule. All times in the *consequent* are later than or at the same time as the latest time in the *antecedent*.

The *antecedents* and *consequents* of reactive rules can also include *complex events* defined by LP clauses of the form:

complex-event if conditions.

where the *conditions* have the same syntax as the antecedents and consequents of reactive rules. The start time of the *complex-event* is the earliest time in the *conditions*, and the end time of the *complex-event* is the latest time in the *conditions*.

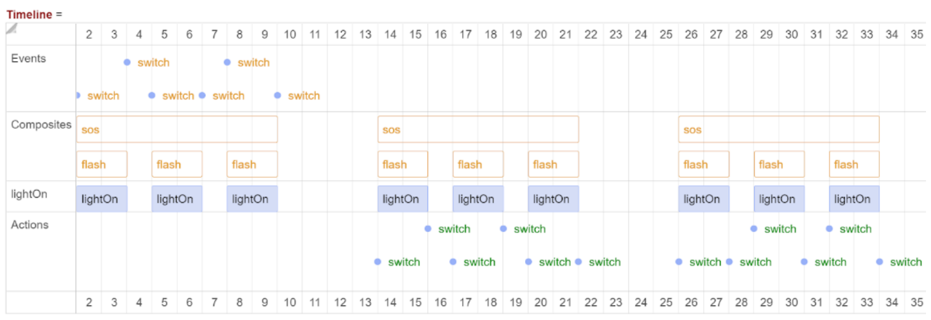
For example, the following two LP clauses define a complex event, *sos*, which is a simplified distress signal of a light flashing three times in succession. Each flash of light is on for two time steps and is separated from the next flash by one time step. We have not yet defined a shorthand, time-free syntax for such clauses:

sos from T1 to T4 if lightOff at T1, flash from T1 to T2,
flash from T2 to T3, flash from T3 to T4.
flash if switch to T, switch from T+1.

Given this definition and replacing the reactive rule above by the rule:

if sos to T then sos from T+3.

LPS uses the complex event definition both to recognise an *sos* and to generate an *sos* in response. Moreover, once it has responded to an *sos* it has recognised, it then recognises its own *sos* and responds to it by generating another *sos*, *ad infinitum* [17]:



In addition to the timeline visualisations, the SWISH implementation of LPS includes animations which display only one state at time. See for example dad chasing around the house to turn off the lights that bob turns on [16].

The preceding examples illustrate the main features of LPS (except for constraints of the from *false if conditions*, which restrict the actions an agent can perform). The examples show that reactive rules are the driving force representing an agent's goals, supported by Prolog-like logic programs, representing the agent’s beliefs.

LPS was inspired by trying to understand the difference and relationship between rules in LP and rules in production systems [18], as well as by complex actions in Transaction Logic [3], reactive rules in MetaTem [2] and agent plans in AgentSpeak [19]. It is also related to CHR [6], DALI [5], Epilog [7] and EVOLP [1, 4].

References

1. Alferes, J.J., Banti, F. and Brogi, A. 2006. An Event-Condition-Action Logic Programming Language. *10th European Conference on Logics in Artificial Intelligence.* In *JELIA 06:* Lecture Notes in Artificial Intelligence 4160, Springer-Verlag, 29- 42.
2. Barringer, H., Fisher, M., Gabbay, D., Owens, R. and Reynolds, M. 1996. *The imperative future: principles of executable temporal logic*. John Wiley & Sons, Inc.
3. Bonner, A. and Kifer, M. 1993.Transaction logic programming. In Warren D. S., (ed.), *Logic Programming: Proc. of the 10th International Conf.* 257-279.
4. Brogi, A., Leite, J. A. and Pereira, L. M. 2002. Evolving Logic Programs. In *8th European Conference on Logics in Artificial Intelligence (JELIA'02),* S. Flesca, S. Greco, N. Leone, G. Ianni (eds.), Springer-Verlag, LNCS 2424, Springer-Verlag, 50-61.
5. Costantini, S. and Tocchio, A. 2004. The DALI Logic Programming Agent-Oriented Language. In Alferes, J.J., Leite, J. (eds.) *JELIA 2004. LNCS (LNAI), Vol. 3229*, Springer, Heidelberg, 685–688.
6. Frühwirth, T. 2009. *Constraint Handling Rules.* Cambridge University Press.
7. Genesereth, M. 2013. Epilog for Javascript. http://logic. stanford.edu/epilog/javascript/epilog.js.
8. Kowalski, R. and Sadri, F. 1999. From Logic Programming Towards Multi-agent Systems. *Annals of Mathematics and Artificial Intelligence,* Vol. 25, 391-419.
9. Kowalski, R. and Sadri, F. 2009. Integrating Logic Programming and Production Systems in Abductive Logic Programming Agents. In *Proceedings of The Third International Conference on Web Reasoning and Rule Systems*, Chantilly, Virginia, USA.
10. Kowalski, R. and Sadri, F. 2010. An Agent Language with Destructive Assignment and Model-Theoretic Semantics. In Dix J., Leite J., Governatori G., Jamroga W. (eds.), *Proc. of the 11th International Workshop on Computational Logic in Multi-Agent Systems (CLIM*A), 200-218.
11. Kowalski, R. and Sadri, F. 2011. Abductive Logic Programming Agents with Destructive Databases. *Annals of Mathematics and Artificial Intelligence*, Vol. 62, No. 1, 129-158.
12. Kowalski, R. and Sadri, F. 2012. A Logic-Based Framework for Reactive Systems. In *Rules on the Web: Research and Applications, 2012* – RuleML 2012, Springer-Verlag. A. Bikakis and A. Giurca (Eds.), LNCS 7438, 1–15.
13. Kowalski, R. and Sadri, F. 2014: A Logical Characterization of a Reactive System Language. In *RuleML 2014*, A. Bikakis et al. (Eds.): RuleML 2014, LNCS 8620, Springer International Publishing Switzerland, 22-36
14. Kowalski, R. and Sadri, F. 2015. Model-theoretic and operational semantics for Reactive Computing. *New Generation Computing,* 33(1): 33-67.
15. Kowalski, R. and Sadri, F., 2016. Programming in logic without logic programming. *Theory and Practice of Logic Programming*, *16*(3), pp.269-295.
16. LE light: https://demo.logicalcontracts.com/example/badlight.pl last accessed 2022/09/07
17. LE sos: https://demo.logicalcontracts.com/p/new%20sos.pl last accessed 2022/09/07.
18. Newell, A. and Simon, H.A., 1972. *Human problem solving* (Vol. 104, No. 9). Englewood Cliffs, NJ: Prentice-hall.
19. Rao, A. 1996. AgentSpeak (L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, 42-55.
20. Wielemaker, J., Riguzzi, F., Kowalski, R.A., Lager, T., Sadri, F. and Calejo, M., 2019. Using SWISH to realize interactive web-based tutorials for logic-based languages. *Theory and Practice of Logic Programming, 19*(2), pp.229-261.