Combining Logic Programming and Imperative Programming 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 theorem that is a logical consequence of the program treated as a set of axioms, or by showing that the goal is true in an intended 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 (if at all) when the goal is solved.

In this paper, we present the language LPS (Logic Production Systems) [14-21], which combines the logic programming (LP) and imperative programming notions of computation.

Computation in LPS follows the imperative paradigm of solving goals by generating a sequence of states and events, to make the goals true. However, unlike states in imperative programming languages, which are collections of "variables", states in LPS are sets of facts (called *fluents*) that change with time. In this respect, states in LPS are like relations in a relational database.

Change of state in LPS also follows the imperative paradigm of destructive updates, maintaining only a single current state. However, whereas imperative programs update states by means of assignment statements, LPS uses logic programs that define *causal laws* to add new facts to the current state and to delete old facts. Logic programs in LPS can also be used to define intensional predicates in terms of extensional predicates. These definitions are like view definitions in relational databases.

In general, logic programs are sets of clauses of the form *conclusion if conditions* where the *conclusion* is a simple atomic formula, and the *conditions* can be an arbitrary formula of first-order logic with aggregation operators. However, in the current implementation of LPS in SWISH [33], *conditions* are restricted to conjunctions of atomic formulas and their negations.

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.
```

The first clause defines the initial state (at "time" 1), in which the fluent *lightOn* is *true*. The clause is shorthand for the sentence *holds(lightOn, 1)*, written in the syntax of the event calculus [22].

The second and third clauses define observations of the external event *switch*, which occurs both in the transition from the state at time 1 to the next state at time 2, and in the transition from the state at time 3 to the next state at time 4. The clauses are shorthand for the sentences *happens(switch, 1, 2)* and *happens(switch, 3, 4)* in event calculus syntax.

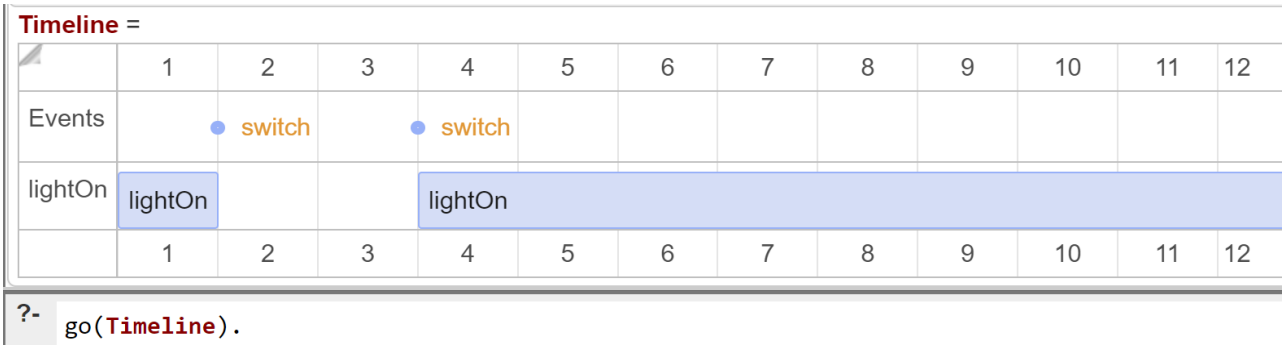
The fourth clause defines the intensional fluent *lightOff* in terms of the extensional fluent *lightOn*. The clause is shorthand for the sentence:

holds(lightOff, T) if not holds(lightOn, T).

The fifth and sixth clauses are causal laws, which specify, in effect, that an occurrence of a *switch* event (whether it is externally observed or internally generated as an action) turns the light on if it is off, and turns it off if it is on. The two clauses are shorthand for the sentences:

initiates(switch, lightOn, T+1) if holds(lightOff, T).
terminates(switch, lightOn, T+1) if holds(lightOn, T).

Given the six clauses above, the SWISH [33] implementation of LPS includes a visualisation of the computation, displaying the history of states and events:



Logically, this history determines a model that *satisfies* the program, by making all the sentences in the program true. It makes extensional fluents true or false, by using causal laws. It makes intensional fluents true or false (as ramifications of changes to extensional fluents), by using intensional predicate definitions.

In this example, the occurrence of the *switch* event from time 1 to 2 terminates the truth of the extensional fluent *lightOn*, so that it is no longer true at time 2. As a consequence, according to both negation as failure (NAF) and the classical meaning of negation, *not lightOn* becomes true at time 2, and consequently *lightOff* also becomes true at time 2.

The sentences *not lightOn* and *lightOff* remain true at time 3, simply because they are not made false by the occurrence of any terminating events. Similarly, the fluent *lightOn* that becomes true at time 4 remains true indefinitely, until some terminating *switch* event occurs.

In general, computation in LPS satisfies the following *causal theory*:

holds(Fluent, T+1) if happens(Event, T, T+1), initiates(Event, Fluent, T+1).
holds(Fluent, T+1) if holds(Fluent, T), not there exists Event such that
[happens(Event, T, T+1), terminates(Event, Fluent, T+1)].

Here the second sentence is a *frame axiom,* which asserts that a fluent that holds at a time *T* continues to hold at the next time *T+1*, unless an event that terminates the fluent occurs between *T* and *T+1*.

It is important to appreciate that LPS does not reason explicitly with frame axioms. Forward reasoning with the frame axiom would entail the computational cost of reasoning

that, for every fluent that holds at a time T and that is not terminated by an event that occurs between T and T+1, the fluent continues to hold at time T+1. Backward reasoning is only marginally better. Backward reasoning, to determine whether a fluent holds at a given time, entails the cost of chaining backwards in time until the time the fluent was initiated, checking along the way that the fluent was not terminated in between times. Both kinds of reasoning are intolerably inefficient compared with destructive change of state.

Instead, in LPS, the frame axiom is an emergent property that is true in the model determined by using destructive change of state.[1] On the other hand, the logical interpretation of destructive change of state in LPS provides a logically pure alternative to the logically impure use of assert and retract in Prolog, which is the way Prolog programmers avoid the inefficiencies of the frame axiom in practice.

Unlike models in modal logic, which are collections of possible worlds connected by accessibility relations, models in LPS are single models in which fluents are time stamped with the times at which they hold, and events are time stamped with the times when they happen. In this example, the Herbrand model, which consists of all the facts that are true in the model, is:

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

In addition to logic programs, which can be regarded as the *beliefs* of an intelligent agent, LPS also includes reactive rules of the form *if antecedent then consequent* and constraints of the form *false if conditions*, which can be regarded as the agent's *goals.* Computation in LPS attempts to satisfy the agent's goals, by performing action events which, together with the agent's beliefs, generate a model that makes its goals true.

For example, the reactive rule *if lightOff then switch*
which 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*]].

represents the goal of switching the light whenever the light is off.

An LPS agent uses its beliefs to determine when the *antecedent* of a rule becomes true, and then it generates actions to make the *consequent* of the rule true. If time is unending, then the model determined by the resulting history of states and events can be infinite, and the computational process might never terminate.

The following timeline displays the initial portion of the infinite model generated by the computation in this example:

[1] This is similar to the way in which the associativity of addition and multiplication is an emergent property that is true in the model generated by a constructive definition of addition and multiplication. But associativity is not used to generate the model, nor to compute sums and products of numbers.

Timeline =

	1	2	3	4	5	6	7	8	9	10	11	12
Events		switch		switch								
lightOn	lightOn		lightOn		lightOn							
Actions			switch		switch							
	1	2	3	4	5	6	7	8	9	10	11	12

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

Here the initial state and external *switch* events are the same as before. However, instead of the intentional fluent *lightOff* persisting from state 2 to state 3, the reactive rule recognises that *lightOff* is true at time 2, and generates the goal of performing a *switch* action in the future. The *switch* action can be performed at any time after time 2. However, in practice, LPS generates models in which goals are satisfied as soon as possible. So, in this case, it performs the action immediately, from time 2 to 3.

Whereas, without the reactive rule, the second *switch* external event turned the light on, now the same external event turns the light off. So, again, the reactive rule is triggered and turns the light back on, 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* of the clause, 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:

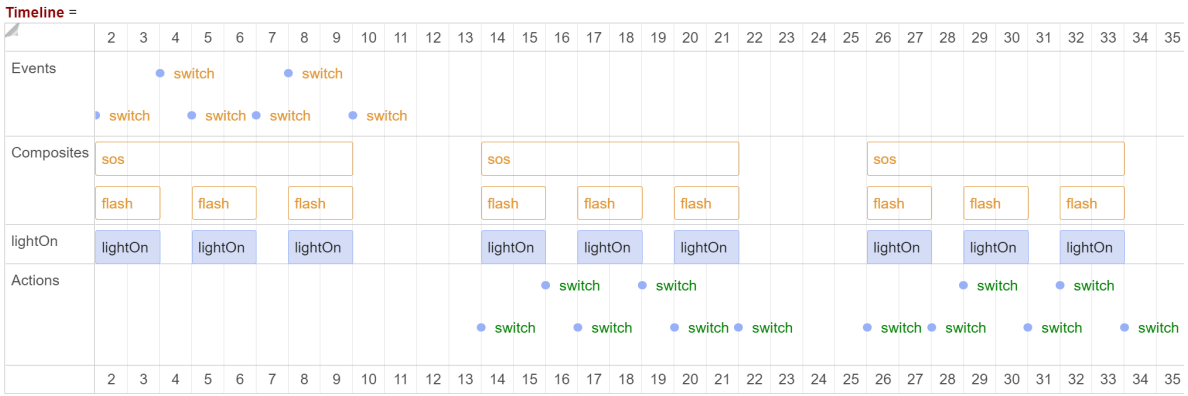
sos from T1 to T4 if lightOff at T1, flash from T1 to T2,
flash from T2 to T3, flash from T3 to T4.
flash if switch to T, switch from T+1.[2]

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

[2] This clause is shorthand for the sentence
happens(flash, T1, T2) if happens(switch, T1, T), happens(switch, T+1, T2).

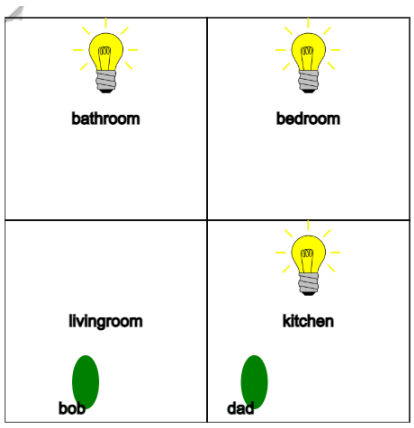
if sos to T then sos from T+3.

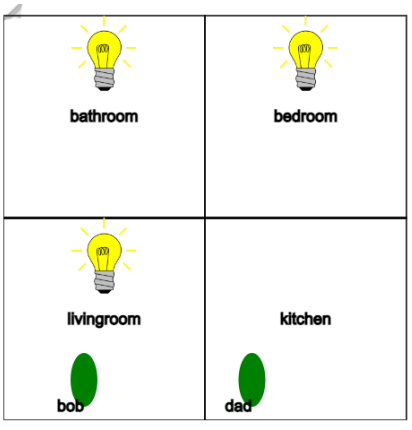
LPS uses the complex event definition both to recognise an *sos* and to generate an *sos* in response. Moreover, once it has responded to an *sos* it has recognised, it then recognises its own *sos* and responds to it by generating another *sos*, *ad infinitum* [24].

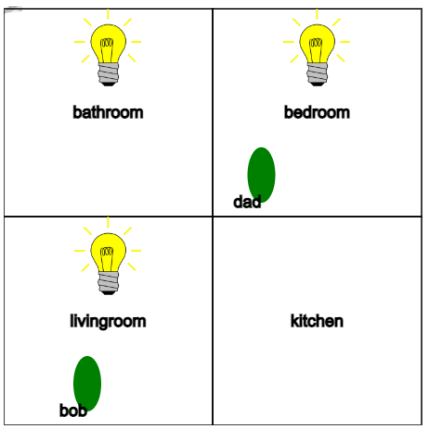


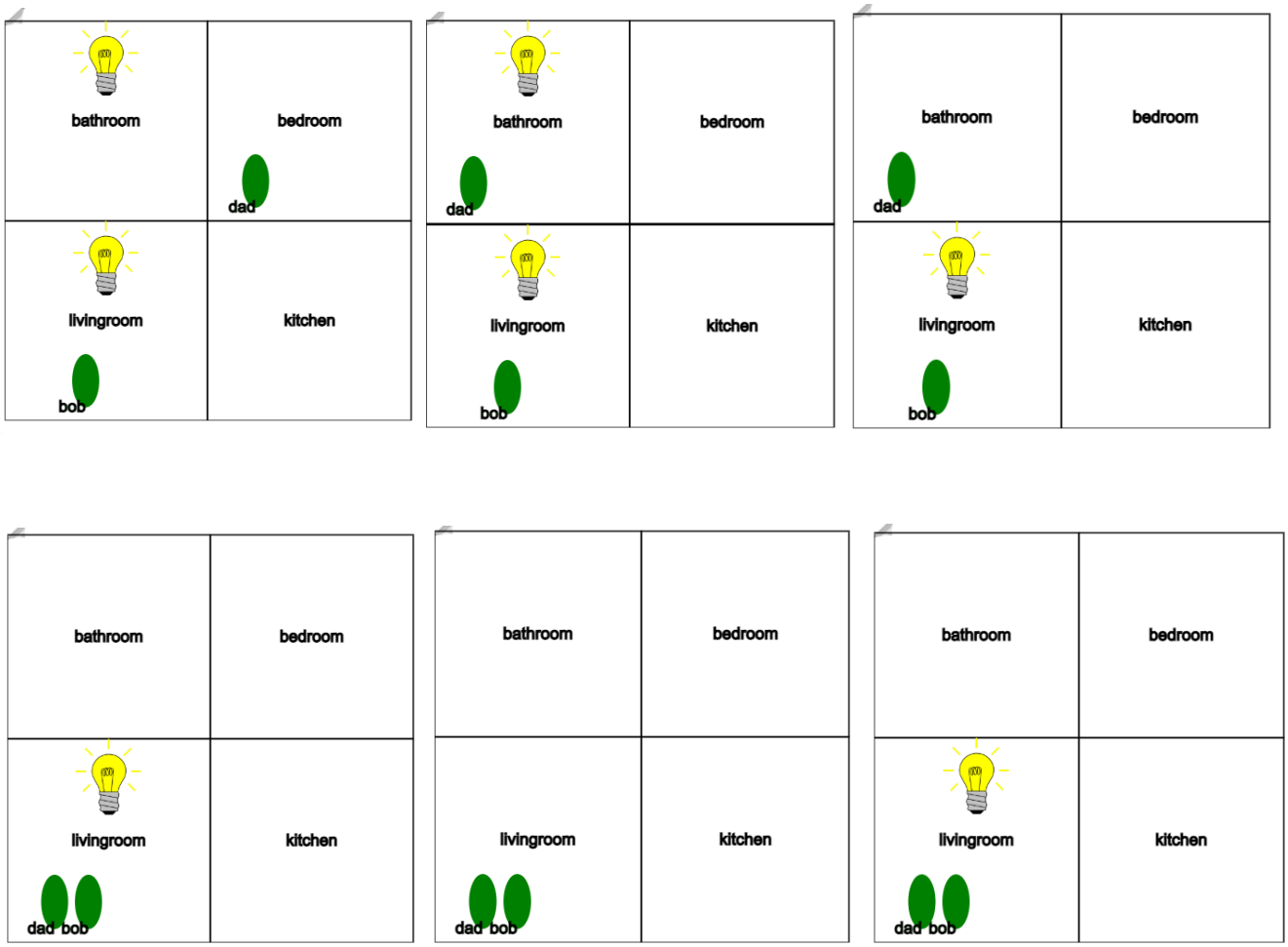
In addition to the timeline visualisations, the SWISH implementation of LPS includes animations which display one state at time. See for example dad chasing around the house to turn off the lights that bob turns on [23]. Here the screenshots of the animation (up to time 9) should be read left to right. Notice that here *switch* has three argument places, to indicate, the agent, place, and result.

```
10 if      location(bob, Place) at T, light(Place, off) at T
11 then    switch(bob, Place, on) from T.
12
13 if light(Place, on) at T, location(dad, Place) at T
14 then switch(dad, Place, off) from T.
15
16 if light(Place, on) at T, not location(dad, Place) at T
17 then goto(dad, Place).
18
```









The preceding examples illustrate the main features of LPS (except for constraints of the from *false if conditions*, which restrict the actions that an agent can perform).

LPS was inspired by trying to understand the difference and relationship between rules in LP and rules in production systems [25]. In part, we were motivated by the fact that both kinds of rules were being used in the 1980s for the practical purpose of implementing expert systems. But we were also motivated to a large extent by the more theoretical use of both LP [30] and production systems to model human thinking. In particular, we argued [13] that confusions between the two kinds of rules helps to explain psychological experiments, such as the Wason selection task [31], which seem to show that people do not reason logically with conditional sentences in natural language.

The fact that there are two kinds of rules (or conditionals) has been widely recognised in such diverse areas as human psychology, normative systems [3], such as law, database systems [32] and programming languages [11]. However, the relationship between the two kinds of rules has been harder to identify.

Our understanding of the relationship was influenced by Gallaire and Nicolas' [26] work on deductive databases in the 1970s. They distinguished between two kinds of general laws: general laws that are used (like logic programs) to derive implicit data from explicit data, and general laws that are used as integrity constraints, which are goals that the database must satisfy. This distinction inspired our work [29] on integrity checking for deductive databases, combining backward reasoning using LP rules with forward reasoning using integrity

constraints, triggered by database updates. This combination of forward and backward reasoning is reflected in the operational semantics of LPS today.

Conventional relational and deductive database systems can also be viewed as passive agents, whose beliefs are the database, and whose goals are its integrity constraints. They are passive because they merely check whether or not external updates preserve the integrity of the database. They cannot actively generate updates to ensure that integrity is maintained.

Our use of actions in LPS, to actively satisfy an agent's goals, was inspired by the generation of hypotheses in abductive logic programming (ALP) [12]. Whereas abduction is normally used to generate hypothetical external events to explain an agent's observations, abduction in LPS is used to generate hypothetical actions that an agent can perform to satisfy its goals.

An intelligent agent not only needs to perform actions, but it also needs to maintain a representation of the world, to help it determine what actions to perform. It is common in AI to represent such knowledge by means of frame axioms, such as those in the event calculus and in the causal theory presented earlier in this paper. But frame axioms are not practical for large scale computer applications. To develop LPS as a practical system, we decided to eliminate frame axioms, and to replace them by destructive change of state. But we were determined to do so within a logical framework, which would allow a wide range of alternative implementation options.

The solution of this last problem in the development of LPS was inhibited by our previous commitment to a theorem-proving view of computation in LP and ALP [9]. To employ destructive change of state within a theorem-proving approach, it would be necessary to destructively change the axioms of a theory in the middle of trying to prove a theorem. But this would also destroy the justification for arguing that the theorem is a logical consequence of the axioms.

We solved the problem by abandoning the theorem-proving view and by replacing it with the model-generation view presented in this and other papers. We were influenced and encouraged in this alternative approach by the model-generation view of computation in such LP languages as XSB Prolog [28], Transaction Logic [4] and Answer Set Programming [5], as well as by the use of model-generation in the modal temporal language MetaTem [2].

Other LP-based languages that are similar to LPS and that have similar motivations to LPS are CHR [8], DALI [7], Epilog [10] and EVOLP [1, 6].

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. Boella, G. and der Torre, L.V., 2005 Regulative and constitutive norms in the design of normative multiagent systems. In International Workshop on Computational Logic in Multi-Agent Systems (pp. 303-319). Springer, Berlin, Heidelberg.
4. Bonner, A. and Kifer, M. 1993.Transaction l.ogic programming. In Warren D. S., (ed.), Logic Programming: Proc. of the 10th International Conf. 257-279.
5. Brewka, G., Eiter, T. and Truszczyński, M., 2011. Answer set programming at a glance. Communications of the ACM, 54(12), pp.92-103.

6. 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.
7. 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.
8. Frühwirth, T. 2009. Constraint Handling Rules. Cambridge University Press.
9. Fung, T.H. and Kowalski, R., 1997. The IFF proof procedure for abductive logic programming. The Journal of logic programming, 33(2), pp.151-165.
10. Genesereth, M. 2013. Epilog for Javascript. http://logic. stanford.edu/epilog/javascript/epilog.js.
11. Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. Sci. Comput. Programming 8, 231-274.
12. Kakas, A., Kowalski, R., Toni, F. 1998. The Role of Logic Programming in Abduction. In: Gabbay, D., Hogger, C.J., Robinson, J.A. (eds.): Handbook of Logic in Artificial Intelligence and Programming 5, Oxford University Press, pp. 235—324.
13. Kowalski, R., 2010. Reasoning with conditionals in artificial intelligence. Cognition and conditionals: Probability and logic in human thinking, pp.254-282.
14. Kowalski, R. and Sadri, F. 1999. From Logic Programming Towards Multi-agent Systems. Annals of Mathematics and Artificial Intelligence, Vol. 25, 391-419.
15. 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.
16. 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 (CLIMA), 200-218.
17. 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.
18. 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.
19. 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
20. Kowalski, R. and Sadri, F. 2015. Model-theoretic and operational semantics for Reactive Computing. New Generation Computing, 33(1): 33-67.
21. Kowalski, R. and Sadri, F., 2016. Programming in logic without logic programming. Theory and Practice of Logic Programming, 16(3), pp.269-295.
22. Kowalski, R., Sergot, M. 2005.A Logic-based Calculus of Events. In: New Generation Computing, Vol. 4, No.1, 67--95 (1986). Also in: Inderjeet Mani, J. Pustejovsky, and R. Gaizauskas (eds.), The Language of Time: A Reader, Oxford University Press.
23. LE light: https://demo.logicalcontracts.com/example/badlight.pl last accessed 2022/11/28.
24. LE sos: https://demo.logicalcontracts.com/p/new%20sos.pl last accessed 2022/11/28.
25. Newell, A. and Simon, H.A., 1972. Human problem solving (Vol. 104, No. 9). Englewood Cliffs, NJ: Prentice-hall.
26. Nicolas, J.M. and Gallaire, H., 1978. Database: Theory vs. interpretation. In Logic and databases (pp. 33-54). Springer, Boston, MA.
27. Rao, A. 1996. AgentSpeak (L): BDI agents speak out in a logical computable language. In Agents Breaking Away, 42-55.
28. Rao, P., Sagonas, K., Swift, T., Warren, D.S. and Freire, J., 1997, July. XSB: A system for efficiently computing well-founded semantics. In International Conference on Logic Programming and Nonmonotonic Reasoning (pp. 430-440). Springer, Berlin, Heidelberg.
29. Sadri, F. and Kowalski, R., 1988. A theorem-proving approach to database integrity. In Foundations of deductive databases and logic programming (pp. 313-362). Morgan Kaufmann.

30. Stenning, K. and Van Lambalgen, M., 2012. Human reasoning and cognitive science. MIT Press.
31. Wason, P. C. 1968. Reasoning About a Rule. The Quarterly Journal of Experimental Psychology, 20:3, 273—281.
32. Widom, J. and Ceri, S. eds., 1995. Active database systems: Triggers and rules for advanced database processing. Morgan Kaufmann.
33. Wielemaker, J., Riguzzi, F., Kowalski, R.A., Lager, T., Sadri, F. and Calejo, M., 2019. Using SWISH to realise interactive web-based tutorials for logic-based languages. Theory and Practice of Logic Programming, 19(2), pp.229-261.