Two Kinds of Rules: *Goal Rules* and *Belief Rules*

Robert Kowalski[0000-0002-1341-8583]

Imperial College London
r.kowalski@imperial.ac.uk

Abstract. It is generally agreed that there are two kinds of if-then rules: logic-based rules, in which *if-then* has a logical semantics, and reactive rules, in which *if-then* represents change of state without a logical semantics. I will argue that reactive rules have an implicit logical semantics, as *goals* that need to be satisfied by generating a model that makes the goals true.

The logical semantics of reactive rules can be made explicit by making change of state explicit, and by understanding the rules as meaning that if some conditions are true at a time, then some actions are performed at a future time. This is the approach taken by the modal logic MetateM. The same approach can be used with a non-modal logic, such as the situation calculus or event calculus. However, all these logics use frame axioms to reason that if a fact is true in a state, then it remains true in the next state, unless it is terminated by the change of state.

Reasoning by means of frame axioms is intolerably inefficient, compared with destructive change of state in conventional reactive systems. I will argue that this inefficiency can be avoided in logic-based systems by using destructive change of state, with frame axioms becoming an emergent property.

But giving a logical semantics to reactive rules leaves open the relationship of reactive rules with ordinary logic-based rules. I will argue that ordinary logic-base rules can be understood as *beliefs* that help an agent satisfy its goals.

Keywords: reactive rules, logic programs, goals and beliefs, LPS

1 Introduction

The distinction between two kinds of rules is widely recognized in the AI and Semantic Web communities: *logic-based rules*, which have a logical semantics, and *reactive rules*, which represent change of state without a logical semantics [15]. Despite this dissimilarity, the two kinds of rules are often confused with one another. Even when a clear distinction is made between them, their relationship can remain unclear.

In [16-21], we presented a language, LPS (Logic-based Production System), which combines logic programs, representing an agent's *beliefs* with reactive rules, representing an agent's *goals*. LPS gives a logical semantics to reactive rules, by treating them as sentences that need to be satisfied by generating a model that makes them true. This semantics of reactive rules in LPS is like the semantics of programs in MetateM [4, 12]. However, MetateM does not have a separate representation for logic programs. LPS, on the other hand, uses the model-theoretic semantics of logic programs to define the space of candidate models that can be generated to make reactive rules true. The resulting combination of reactive rules and logic programs in LPS is similar in spirit to the homogeneous Event-Condition-Action Logic Programming language (ECA-LP) [29].

The distinction in LPS between *goal rules*, which are reactive rules, and *belief rules*, which are logic programs, is similar to other distinctions made in other areas. It is related, for example, to the distinctions between:

- Integrity constraints and derivation rules in database systems [26];
- Regulative and constitutive norms in normative systems [5];
- Operative rules and structural rules in the operation of an enterprise [27];
- Reactive systems and transformational systems in "software engineering, programming languages, and system and hardware design" [13].
- Beliefs and desires in the belief-desire-intention (BDI) model of intelligent agents [7, 10, 11].

History. The origins of LPS go back to our work on combining integrity constraints with logic programming rules in deductive databases [32]. This contributed to our subsequent work on abductive logic programming (ALP) [14], using logic programs to generate abductive explanations and using integrity constraints to eliminate unacceptable candidate explanations. Later, we embedded ALP into an agent cycle, using integrity constraints both to generate reactive behaviour and to serve as higher-level maintenance goals [16]. We developed LPS, as a scaled-down variant of ALP agents, to combine the functionality of both production rules and logic programming rules, together with destructive change of state for the sake of efficiency [17].

During these developments, we were aware of MetateM, but we did not understand how it related to our work. We were also aware of the BDI agent model [7]. However, in both cases, I was dissuaded by their use of modal logic, which seemed unnecessarily complicated and conceptually questionable.

In the case of MetateM, I eventually realized that its use of modal logic for representing time is a relatively minor difference from our use of explicit timestamps. Moreover, I was encouraged to discover that we came to the same conclusion of viewing computation as model generation, coming from opposite directions.

The relationship between LPS and BDI is more complicated. On the one hand, BDI overlaps with LPS and ALP with its distinction and focus on goals and beliefs. But on the other hand, the early BDI agent systems were specified in multi-modal logics, with separate modal operators for goals, beliefs and intentions. However, the practical implementations of BDI agents bore little resemblance to their logical specifications.

Following the lead of AgentSpeak [31], most BDI agent implementation languages abandoned their modal specifications and their logical semantics, and they represented beliefs, desires, and intentions as "data structures" [10, 11, 31]. In AgentSpeak, the main data structure is a *plan*, which has the rule-like form: *triggering event: beliefs* ← *goals or actions*. The arrow is written in the backward direction because of its resemblance to a logic program rule. However, unlike a logic program rule, the invocation of a plan can be "both data-directed (using addition/deletion of beliefs) and goal-directed (using addition/deletion of goals)" [31]. As a consequence, plans in BDI languages of the AgentSpeak variety do not have a logical semantics.

In this paper, I will explore:

- How the logical semantics of LPS and MetateM can be used to give a logical semantics to other reactive rule languages;
- How the use of destructive change of state in LPS can be used to improve the efficiency of other logic-based languages;
- How the incorporation of logic programs in LPS can be used to improve the level of abstraction in other reactive rule languages; and
- How the combination of goal rules and belief rules in LPS can be viewed as a scaled-down model of human thinking.

The paper is deliberately written in an informal style, to make it more accessible to readers without an extensive background in logic and computing. More rigorous treatments can be found in the references.

2 Reactive Systems

Reactive systems include a wide range of programming, database and AI systems [28]. Not all these systems are rule-based, but they all treat computation as generating a sequence of states to maintain a desired relationship with a changing environment. Production systems are possibly the simplest example of such a reactive rule system.

Production systems. A *production system* [9] consists of:

- A *working memory*, represented as a set of facts;
- *Production rules* of the form *if conditions then actions*;
- An *"inference engine"*, which uses "forward chaining" to match conditions of rules with facts in the working memory, and derives candidate actions;
- *Conflict resolution*, which selects candidate actions for execution; and
- *Action execution*, which adds, deletes or modifies facts, updating the current state of the working memory.

As Paul Thagard remarks in his Introduction to Cognitive Science [35], production rules "are very similar to" logical conditionals, "but they have different representational and computational properties". In this paper, I will show how LPS reformulates production rules as logical conditionals and combines them with logic programs.

MetateM. To a first approximation, computation with reactive rules in LPS is like computation in MetateM. Computation in MetateM uses *reactive rules*, to maintain a *current history of states* and a current collection of *commitments*, where:

- The *current history of states*, including the present state and all past states, is represented by a set of facts.
- *Reactive rules* have the form *if antecedent then consequent*.
- *Computation* uses forward reasoning with reactive rules to generate a *complete history* of states.
- *Forward reasoning* identifies any instances of the *antecedents* of rules that are true in the current history. For each such rule, it derives the *consequent* of the rule as a *commitment* to be made true in the next state or in the future.
- Given the complete collection of all previous and new commitments, a *choice* is made between different actions needed to make the commitments true.
- The chosen commitments and any external events are executed, updating the current history by *adding facts* that are true in the next state.

The *logical semantics* of the reactive kernel of LPS, KELPS [20], is similar to the logical semantics of MetateM. In both cases, reactive rules represent *goals*, and computation generates a complete history to try to make the goals true.

Reactive rules in MetateM generalize production system rules. Whereas antecedents of rules in production systems can only express conditions about the current state of the working memory, antecedents in MetateM can refer to any states or events in the current history. Whereas consequents of rules in production systems can only refer to actions to be performed immediately, consequents in MetateM can express commitments about any states or actions in the future.

As a consequence of this greater generality, MetateM needs to store the complete history of previous states, and not only the current state. Moreover, it needs to store the complete collection of all present and future commitments, whereas production systems do not need to store any commitments at all.[1]

MetateM is a modal temporal logic, with a possible world semantics, in which individual states are models, and the history of all individual states is itself a model. Unlike the sequence of states in production systems, the history is constructed by adding facts, without deleting any facts.

By treating computation as model generation, MetateM provides a simple and elegant, logical interpretation of *conflict resolution*, as a choice between alternative ways of making the goals true. This choice creates a natural opportunity to employ decision theory and game theory, to aid in the choice of alternatives [3].

Example. Given the production rules:

```
If hungry then eat.  If sleepy then sleep.
```

and a current state in which hungry and sleepy are both true, a production system would need to perform conflict resolution, to decide whether to eat or sleep (and to do so immediately), assuming it is not possible to eat and sleep at the same time. This means

[1] However, in practice, production systems (and BDI implementations) often add "goal facts" to working memory, both to represent future commitments and to simulate goal reduction rules.

that the production rules cannot be treated as logical implications, because taken literally there is no model in which both rules are true.

In MetateM, the same rules would be written in the form:

If hungry then ◊ eat. If sleepy then ◊ sleep.

where ◊ is a temporal modal operator, which means "at some time in the future".

Moreover, it is also necessary to state explicitly the *constraint* that eat and sleep cannot both be true at the same time. This constraint can be expressed by the rule:

If ● true then not eat or not sleep.

where ● means "at the last time", and ● true is always true in the previous state, and therefore not eat or not sleep is true in all states.

To satisfy the rules, MetateM needs to reason with "frame axioms", which express that if a fact is true at a time and the fact is not terminated by an action or external event that occurs at the time, then the fact continues to be true at the next time. For example:

If hungry and not eat then ○ hungry.
If sleepy and not sleep then ○ sleepy.

where ○ is a modal operator, which means "in the next state".

Given a current state in which hungry and sleepy are both true, there are infinitely many models that satisfy the goals, depending on how far into the future the commitments ◊ eat and ◊ sleep are made true. Among other decision-making strategies, the MetateM interpreter chooses models that make commitments true as soon as possible.

Brief comparison with reactive rules and constraints in LPS. Reactive rules in LPS are like rules in MetateM. In both cases, antecedents of reactive rules can refer to any states or events in the current history, and consequents can express commitments about any states or actions in the future.

The semantics of reactive rules in LPS is also like the semantics of rules in MetateM. In both cases, reactive rules are treated as goals, which need to be made true in the complete history of all states and state-changing events. But, whereas MetateM is a modal temporal logic, in which a complete history is a collection of models, one for each state, LPS is a non-modal logic in which a complete history is a single model in which facts whose truth values can change with the passage of time are timestamped.

In LPS, because all times in the consequent of a reactive rule are later than or equal to the latest time in the antecedent, and because the conditions in both the antecedent and consequent can be written in temporal order, time variables can often be suppressed, as in:

if hungry then eat. if sleepy then sleep.

Written with explicit time variables, these sentences become:

If hungry at any time T1 then eat from some time T2, where T1 ≤ T2.
If sleepy at any time T1 then sleep from some time T2, where T1 ≤ T2.

Time variables can also be suppressed if a constraint has only one time variable. For example, using LPS syntax:

false if sleep, eat.

which means: For all times T, it is not the case that: eat from T and sleep from T.

In general, reactive rules generate candidate actions, and constraints eliminate candidate actions. This combination of reactive rule goals and constraint goals is like the combination of obligations and prohibitions in deontic logic. The possibility that obligations can be violated, resulting in a suboptimal situation can be understood as some models being better than others [22].

The same combination of reactive rules and constraints is also like the combination of liveness and safety properties in model checking of concurrent and distributed systems. In model checking, liveness and safety need to be verified as *emergent properties* of a program. However, in model generation, as in MetateM and LPS (and potentially in production systems), liveness and safety are built into the program, and execution of the program is designed to ensure that liveness and safety are true by design.

In the next section, we will explore the relationship between the use of frame axioms to generate state transitions, as in MetateM, with the use of destructive updates, as in production systems and LPS. In the following sections, we will:

- present KELPS, which is the reactive kernel of LPS,
- show how logic programs can be used in conjunction with reactive rules, and
- discuss the potential of LPS as a scaled-down model of human thinking.

3 Representing and Reasoning about Change of State

In production systems and many other reactive rule languages, actions are normally bookkeeping operations that add, delete or modify facts in working memory. In logic-based systems, actions and other events represent events in the problem domain.

Frame axioms. In logic-based systems, the standard approach for dealing with change is to use a causal theory [33], in which times, states or situations are reified and are represented explicitly as arguments of predicates. Many of these approaches distinguish between *fluents*, which are facts that hold true at time points (or over time intervals), and *events*, which occur between time points. Events include both *external events*, generated by the environment, and *actions*, generated by the system. Change of state can be defined by two meta-level axioms, the second of which is a *frame axiom*:

```
If the events that occur between T and T+1 initiate a fluent,
then the fluent is true at T+1.

If a fluent is true at T
and the fluent is not terminated by the events that occur between T and T+1,
then the fluent is true at T+1.
```

To apply these axioms in a specific domain, the initiation and termination predicates need to be defined by *causal laws*, such as:

```
eat initiates eating.        eat terminates hungry.
sleep initiates sleeping.    sleep terminates sleepy.
```

Forward reasoning with such axioms means that, when a state transition occurs from time (or state) T to T+1, all the fluents that are initiated and all the fluents that are already true at time T, but not terminated, are added to the new state with the new timestamp T+1. If a fluent is terminated, it is not deleted, but simply not added to the new state with a new timestamp. This approach respects the logic of change of state, but it does so at the cost of copying every fluent that is not terminated, from one state to the next. This is certainly not practical for even a moderately large number of fluents.

Destructive change of state. Most practical computer languages, including production systems, reason with change of state by storing only a single current state, which can be regarded as a database of facts that are true in that state. These facts are stored without timestamps, and are updated destructively, by adding any facts that are initiated and deleting any facts that are terminated. Any facts that are not terminated are left unchanged, without reasoning that the facts are unchanged. This practical approach is very efficient, but it raises the problem of how to reconcile such destructive change of state with logic-based representations that employ frame axioms.

Reconciling frame axioms with destructive change of state. The solution of the problem can be found in the way that the real world exists only in its current state, changing state by destroying its past and unfolding its future. But in its totality, the real world is the complete history of all its states and events, past, present and future. The frame axiom is true in this history, but it is not used to generate change of state.

The implementation of LPS employs a similar approach: It maintains a single current state in which facts are represented without explicit times or states. As a result, facts that are not terminated by a state transition can be left unchanged without using frame axioms to reason that they are unchanged. However, in the complete history, fluents and events are timestamped, and the frame axiom is true as an emergent property.

The LPS approach can also be used to implement and justify destructive change of state in other logic-based languages, such as MetateM. Conversely, the LPS approach might be used to give a logical semantics to suitably modified production systems and other reactive rule languages. For this purpose, facts need to be associated with the times at which they hold, and reactive rules need to be understood as rewritten with explicit times. In addition, constraints need to be written explicitly, and they need to be enforced to prevent untimely choice of actions. Moreover, conflict resolution needs to be restricted to strategies that have a logical interpretation, such as specifying preferences between different models.

4 The Reactive Kernel of LPS (KELPS)

The kernel of LPS (KELPS), which consists of a current state, reactive rules, constraints and causal laws, is a reactive system language in its own right [20]. It combines the efficiency of destructive change of state in production systems with the expressive power and logical semantics of MetateM.

Compared with production systems, the greater expressive power of KELPS and MetateM comes from the more general antecedents and consequents of their reactive rules. KELPS deals with these more general consequents, in the same way as MetateM, by maintaining a separate collection of all present and future commitments.

The problem of dealing with more general antecedents is more difficult. MetateM deals with this problem by maintaining the entire current history of states and events, to identify any instances of antecedents that are true in the current history. But this current history is not available in KELPS, which maintains only the current state. KELPS deals with the problem by partially evaluating the antecedents of rules in the current state, and generating the partially evaluated rules as additional rules to be evaluated further in future current states.

For example, consider a rule expressing that if an item is ordered, and the item is not delivered two days later, then the order is cancelled.

```
If order(Item, OrderId, Day1),  Day1 + 2 = Day2, not delivered(Item, Day2)
then cancel(OrderId, Day2).
```

When an event, say order(newBed, ord0017, 01/04/2026) occurs, LPS unifies the event with the first condition of the rule, evaluates the second condition of the rule and adds the *resolvent* as a new reactive rule to be monitored in the future:

```
If not delivered(newBed, 03/04/2026) then cancel(ord0017, 03/04/2026).
```

In general, KELPS uses *causal laws* and *constraints* to maintain a *current state*, a current collection of *commitments*, and a current collection of *reactive rules*, where:

- The *current state* is represented by a set of facts, called *fluents*, which are represented without timestamps.
- *Reactive rules* have the form *if antecedent then consequent*, where all variables (including time variables) in the *antecedent* are universally quantified with scope the entire rule, and all variables in the *consequent* that are not in the *antecedent* are existentially quantified with scope the consequent of the

rule. All times in the *consequent* are later than or equal to the latest time in the *antecedent*.

- *Causal laws* represent the postconditions of events. They have the form *events initiate fluents* and *events terminate fluents*, where all variables are universally quantified, and all variables in the *fluents* occur in the *events*.[2]
- *Constraints* represent the preconditions of actions. They have the form *false if actions and conditions*, where all variables are universally quantified with scope the entire constraint.
- At any time that any events occur, including any actions that may have been chosen to be performed at that time, the causal laws are used to update the current state, *adding any fluents* that are initiated by the events and *deleting any fluents* that are terminated by the events.
- *Forward reasoning* identifies any instances of conditions (including events) in the *antecedents* of rules that are true at the current time. For each such rule, it derives the *resolvent* as a *new reactive rule* to be made true in the future. If there are no remaining antecedents in the rule, it derives the instantiated *consequent* of the rule as an additional commitment.
- Given the updated collection of all commitments, a *choice* of actions is made, selecting the actions from among alternative ways of making the commitments true. The choice needs to satisfy all the constraints.

In the *logical semantics* of KELPS, reactive rules and constraints are *goals*, and computation attempts to make these goals true in the complete history of all states and events. For this purpose, all fluents, external events and actions in the history are represented with timestamps.

In generating the history, the causal laws, the external events and the fluents belonging to the initial state all serve as *beliefs*, which determine the search space of histories that can be generated. In addition, as soon as an action is performed and becomes true, it also becomes a belief, as do all the fluents that then belong to the resulting state.

In full LPS, beliefs can be defined more generally by logic programs.

5 Combining Goals and Beliefs in LPS

Logic programming (LP). A logic program consists of:

- A set of *facts*[3], represented by atomic sentences; and
- A set of *rules* of the form *conclusion if conditions*, where the *conclusion* is an atomic formula, possibly containing variables, and the *conditions* are a

[2] Note that *initiate* and *terminate* are meta-predicates, which express a relationship between ordinary object-level predicates.

[3] In this paper, facts do not contain variables. So, for example sleep(P) initiates sleeping(P), where P is a variable, would not be a fact, but would be a rule without any conditions. This contrasts with the usual convention in LP that rules without conditions are called "facts".

conjunction of atomic formulas and negations of atomic formulas. All variables are universally quantified, with scope the entire rule.[4]

Negative conditions are interpreted, by *negation as failure*, where not p means that p cannot be shown. Negation as failure can be understood as exploiting the *closed world assumption* that the logic program contains all the information about its subject matter. For example, it is natural to assume that a railway timetable contains all the information about train journeys within its geographical area. So, if it cannot be shown that there is a train connection between two places at a time, then there is no train connection between the two places at the time.

In LPS, negation needs to be stratified (or locally stratified [30]). This excludes rules, such as p if not p, where the conclusion of the rule depends negatively on itself. Logic programs conforming to this restriction have a unique minimal model, whose true facts coincide with the facts that are true in all the models of the program. This model can be understood as the intended model defined by the logic program.

Logic programs in LPS. In LPS, logic programs are used to define:

- *Extensional fluents*, which are defined by facts and *intensional fluents*, which are defined by rules.
- *Causal laws*, in which the fluents that are initiated and terminated by events depend on fluents that are true in the current state.
- *Composite events*, which include multiple, simpler events, combined in logical and temporal relationships, and which include plans of actions.
- *Timeless auxiliary predicates*, including arithmetic predicates, which do not depend upon time.

Fluents. In LPS, there are two kinds of fluents: *Extensional fluents*, which represent "ground truths" or concrete facts, are defined by LP facts. *Intensional fluents*, which represent abstract concepts or other consequences of the concrete facts, are defined by LP rules. Extensional fluents are stored explicitly in the current state and are updated when they are initiated and terminated by events. Intensional fluents are not stored explicitly but are derived as ramifications of changes to the extensional fluents.

For example, the balance in a bank account might be stored in an extensional fluent, and the interest in the account might be represented by an intensional fluent:

```
balance(bobAccount, 100).
InterestRate(Account, 0) if balance(Account, Amount), Amount < 1000.
```

If the balance in bobAccount changes due to any bank transfers, then any necessary changes to the interest rate will take place automatically.

[4] Or, equivalently, all variables in the conclusion are universally quantified with scope the entire rule, and all variables in the conditions that are not in the conclusion are existentially quantified with scope the conditions of the rule.

Ironically, the same kind of automatic updating of an intensional fluent as the result of updating an extensional fluent is called *reactivity* in “reactive programming” [2].

Causal laws. Causal laws with variables in KELPS are actually LP rules without any conditions. They are not expressive enough for realistic examples. Here is an example that shows the use of conditions in causal laws:

```
    transferTo(Account, Amount) initiates balance(Account, NewBalance)
    if balance(Account, OldBalance), NewBalance = OldBalance + Amount.

    transferTo(Account, Amount) terminates balance(Account, OldBalance).
```

Composite events. Similarly to the way that intensional fluents provide an abstract, higher-level view of concrete, extensional fluents, composite events provide an abstract, higher-level view of simple events, including plans of actions.

For example, in formal grammar, a non-terminal symbol can be understood as a composite event, defined in terms of terminal symbols and other non-terminals. Terminal symbols are simple events, which cannot be decomposed into other events. For example, here saying is a composite event, and say is a simple event:

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

saying(Agent, verbphrase3) from T1 to T3
if saying(Agent, verb) from T1 to T2, saying(Agent, nounphrase) from T2 to T3.

saying(Agent, noun) from T1 to T2    if  say(Agent, donna) from T1 to T2.
saying(Agent, noun) from T1 to T2    if   say(Agent, logic) from T1 to T2.
saying(Agent, verb)  from T1 to T2    if   say(Agent, likes) from T1 to T2.
```

The definition of saying in terms of say can be used both to recognize sentences and to generate sentences. A silly example can be found and executed online at https://demo.logicalcontracts.com/example/turingTest.pl

Important examples of the use of logic programs to define composite events are Golog [25], Openlog [8] and Transaction Logic [6]. Golog and Openlog both use frame axioms, Golog uses the situation calculus, whereas Openlog employs a procedural syntax with a semantics and interpreter based on abductive logic programming. Transaction Logic uses destructive change of state with a possible world semantics.

The dining philosophers. The problem of the dining philosophers illustrates the combination of logic programs, constraints and reactive rules to solve a classic problem in concurrent systems. The solution in LPS presented here can be found and executed online at https://demo.logicalcontracts.com/p/new%20dining%20philosophers.pl. Similar solutions can be implemented in other reactive rule languages, such as MetateM, suitably extended with logic programs, as suggested above.

There are five philosophers, sitting in a circle around a table with one fork to the left and one fork to the right of each philosopher. For simplicity, being a philosopher and sitting at the table with a pair of adjacent forks are both represented by timeless predicates. They could equally well be represented by fluents that are true initially and never terminated. Here there is only one fluent, representing that a fork is on the table, available to be picked up by one or other of the two adjacent philosophers:

```
philosopher(donna).  philosopher(bob).  philosopher(dania).
philosopher(tania).    philosopher(nina).

adjacent(fork1, donna, fork2).   adjacent(fork2, bob, fork3).
adjacent(fork3, dania, fork4).    adjacent(fork4, tania, fork5).
adjacent(fork5, nina, fork1).

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

The goal is represented by a reactive rule expressing that all philosophers must eventually dine, and the constraints that a fork cannot be picked up if it is not available and that two philosophers cannot pick up the same fork at the same time:

```
if   philosopher(P)   then dine(P).

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

Dining is defined by an LP rule as a composite event (or plan) of picking up two adjacent forks simultaneously, eating, and putting down the two forks simultaneously:

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

The causal laws defining change of state, namely that putting down a fork initiates its availability, and picking up a fork terminates its availability, are also expressed in LP:

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

LPS solves the goal by generating the following sequence of actions:

```
Time 1 to 2: nina and dania pick up their adjacent forks.
Time 2 to 3: nina and dania eat.
```

Time 3 to 4: nina and dania put down their forks.
Time 4 to 5: tania and donna pick up their adjacent forks.
Time 5 to 6: tania and donna eat.
Time 6 to 7: tania and donna put down their forks.
Time 7 to 8: bob picks up his adjacent forks.
Time 8 to 9: bob eats.
Time 9 to 10: bob puts down his forks.

Reasoning with logic programs. In general, logic programs can be used to reason either *forwards (or bottom-up)* to derive new facts from existing facts, or *backwards (or top-down)* to reduce goals to subgoals. Most Datalog systems reason forwards, and all Prolog systems reason backwards. Like Prolog, LPS also reasons backwards, both to evaluate the truth of conditions in the antecedents of reactive rules, and to reduce goals to subgoals in the consequents of reactive rules.

However, the logical semantics of LPS is compatible with other reasoning strategies. In particular, the integrity checking method of [32] could be used to trigger reactive rules by reasoning forwards from updates and deriving facts or new rules whose conclusions unify with conditions of reactive rules.

6 Combining Goals and Beliefs in Human Thinking

I have argued that the distinction between goals and beliefs is a useful way to look at rule-based computer systems. It is also possible to argue that a similar combination of goals and beliefs can serve as a cognitive model of human thinking. The plausibility of the argument relies in part on the success of production systems in such cognitive models as Soar [24] and Act-R [1].

The argument is complicated by the fact that it is easy to confuse different kinds of rules. The most famous example is probably the confusion about the meaning of conditionals in the Wason selection task.

The Wason selection task. In the standard version of the task, participants are given four cards lying on a table, with numbers showing on one side of the cards and letters showing on the other side. Participants are also given the rule:

if a card has a vowel on one side, then it has an even number on the other side.

The task is to determine which cards need to be turned over to test whether the rule is true or false. Typically, only 10 % of the participants reason correctly with the conditional as a material implication in classical logic [36].

Cognitive psychologists have proposed a wide variety of explanations for human performance on the Wason task, including the explanation that human reasoning is performed by domain-specific methods as opposed to general-purpose, logical reasoning. Arguably, a better explanation is that conditionals can represent goals or beliefs, and that different logical reasoning methods apply, depending on whether a conditional is interpreted as a goal or as a belief [23, 34].

Stenning and van Lambalgen [34], in particular, argue that the response of most participants in the selection task is consistent with their interpreting the conditional descriptively as a logic program rule: Participants correctly turn over the card showing a vowel, to make sure that the card has an even number on the other side. They also turn over the card showing an even number, which is consistent with the closed world assumption, but unnecessary with the open world assumption of classical logic. Moreover, they do not turn over the card showing an odd number, because this involves reasoning with negation in ways that are not common with logic program rules, even though it is necessary when reasoning with material implications in classical logic.

Stenning and van Lambalgen argue that most participants in the selection task reason in accordance with classical logic if the conditional has a natural interpretation as a prescription. For example, given a scenario involving people drinking in a bar, most participants in the selection task will interpret the following conditional prescriptively; and they will reason with it correctly as a material implication:

If a person is drinking alcohol in a bar, then the person is over eighteen.

They will not only check whether a person drinking alcohol is over eighteen, but they will also check whether a person who is under eighteen is drinking alcohol. They will not check what a person over eighteen is drinking. Nor will they check the age of a person drinking tap water.

LPS as a cognitive model. These arguments about the two interpretations of conditionals in psychological experiments support the view of LPS as a cognitive model, which combines beliefs, which are descriptive, with goals, which are prescriptive. However, they also show that LPS does not fully support the complete range of human thinking associated with satisfying goals as material implications.

LPS and other reactive rule languages can only make goals of the form if p then q true *reactively*, by making q true when p becomes true. They cannot make goals true *proactively*, by making q true before p becomes true. Nor can they make goals true *preventatively*, by making p false. For example, consider the goal:

If you leave home then the front door is locked.

Given LP rules that initiate the fact that the door is locked by generating an action of locking the front door, LPS will make the goal true reactively, by performing the action of locking the door when leaving home. However, LPS will not attempt to make the goal true proactively, by locking the door before leaving home. Nor will it attempt to make the goal true preventatively, by not leaving home.

LPS has been deliberately scaled down to eliminate these kinds of reasoning, to make it more efficient for routine computational activities. However, routine computation and other kinds of more intelligent reasoning are possible in abductive logic programming (ALP) agents [17].

ALP agents. ALP agents perform abductive reasoning, to generate hypotheses to make goals true in a model determined by the agents' beliefs. Abductive reasoning in ALP includes both the generation of hypothetical actions, as in LPS, as well as the generation of hypothetical events, to explain observations of facts that are true in the environment.

Beliefs in ALP are used reason backwards to reduce goals to subgoals. They are also used to reason forwards to derive logical consequences, both from given facts and from hypothetical facts. In particular, more generally than in LPS, beliefs are used to reason forwards from hypothetical actions to determine their logical consequences, and to determine whether they might violate constraints or have other noteworthy side effects. This use of forward reasoning, together with estimates of the probability of circumstances that are outside the agent's control, can help an agent to make better decisions and obtain better solutions for its goals.

7 Conclusions

The web-based implementation of LPS on SWISH [37] at https://demo.logicalcontracts.com/ includes a variety of runnable and editable examples. These include the dining philosophers, the prisoner's dilemma, rock-paper-scissors, sorting, map colouring, toy blocks worlds, Conway's game of life, self-driving cars and bank account transactions. The implementation includes a declarative Prolog-based sublanguage for associating images with fluents, animating the history computed by the program.

The implementation is still only a proof of concept, which needs to be developed further to make it viable for realistic applications. In particular, further work is needed to improve the strategy for choosing between alternatives, when there is a choice of different actions to satisfy the same goals.

Improving the current implementation is only one direction for future work. It might also be useful to explore other directions, including how to use some of the features of LPS in other computer languages. For example, the way in which LPS (and MetateM) gives a logical semantics to reactive rules could also be used to give a logical semantics to suitably modified versions of other reactive rule languages. Moreover, the combination of goals and beliefs in LPS might also suggest ways in which reactive languages can be extended to include logic program beliefs, and ways in which LP languages might be extended to include constraint goals and reactive rule goals.

Acknowledgments. Although the opinions expressed in this paper are my own, Fariba Sadri was an equal contributor to all the technical work. Miguel Calejo and Jacinto Dávila also made important contributions. Fariba, Veronica Dahl, Dov Gabbay, Daniel Harris, Aidan Hogan, Ken Satoh and Mario Wenzel made helpful comments on an early draft of the paper. The implementation of LPS at https://demo.logicalcontracts.com/ was supported by an EPSRC Pathways to Impact research grant.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

References

1. Anderson, J.R.: The architecture of cognition. Psychology Press (2013)
2. Bainomugisha, E., Carreton, A.L., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: A survey on reactive programming. ACM Computing Surveys (CSUR) (2013)
3. Baron, J.: Thinking and Deciding. Cambridge University Press (2023)
4. Barringer, H., Fisher, M., Gabbay, D., Owens, R., Reynolds, M. (eds.): The Imperative Future: Principles of Executable Temporal Logic. John Wiley & Sons, Inc. (1996)
5. Boella, G., van der Torre, L.: Regulative and constitutive norms in normative multiagent systems. In: Ninth international conference principles of knowledge representation and reasoning, pp. 255-265. Whistler, Canada (2004)
6. Bonner, A.J., Kifer, M.: An overview of transaction logic. Theoretical Computer Science, 133(2), 205-265 (1994)
7. Bratman, M. E.: Intentions, Plans, and Practical Reason. Harvard University Press (1987)
8. Dávila, J.A.: Openlog: A logic programming language based on abduction. In: International Conference on Principles and Practice of Declarative Programming, pp. 278-293. Springer, Berlin Heidelberg (1999)
9. Davis, R., King, J.: An overview of production systems. Technical Report, STAN-CS-75-524, AIM-27, Stanford University (1975)
10. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A common semantic basis for BDI languages. In: International Workshop on Programming Multi-Agent Systems, pp. 124-139. Springer, Berlin Heidelberg (2007)
11. De Silva, L., Meneguzzi, F.R., Logan, B.: BDI agent architectures: A survey. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI) (2020)
12. Gabbay, D.: The declarative past and imperative future. In: Barringer, H. (ed.) Proceedings of the Colloquium on Temporal Logic and Specifications, pp. 409-448. Vol. 398 of LNCS, Springer Verlag (1989)
13. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Sci. Comput. Programming 8, 231-274 (1987)
14. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. Journal of logic and computation, 2(6), 719-770 (1992)
15. Kifer, M.: Rule interchange format: The framework. In: International Conference on Web Reasoning and Rule Systems, pp. 1-11. Springer, Berlin Heidelberg (2008)
16. Kowalski, R., Sadri, F.: Towards a unified agent architecture that combines rationality with reactivity. In: Proceedings of International Workshop on Logic in Databases, pp. 131-150. Springer-Verlag, LNCS 1154, (1996)
17. Kowalski, R., Sadri, F.: Abductive logic programming agents with destructive databases. Annals of Mathematics and Artificial Intelligence, 62(1), 129-158 (2011)
18. Kowalski, R., Sadri, F.: A logic-based framework for reactive systems. In: RuleML 2012, LNCS 7438, pp. 1-15. Springer, Heidelberg (2012)
19. Kowalski, R., Sadri, F.: Reactive computing as model generation. New Generation Computing, 33, 1, 33-67 (2015)
20. Kowalski, R., Sadri, F.: Programming in logic without logic programming. TPLP, 16, 269-295 (2016)
21. Kowalski, R., Sadri, F., Calejo, M., Dávila, J.: Combining logic programming and imperative programming in LPS. In: Prolog: The Next 50 Years, pp. 210-223. Cham: Springer Nature, Switzerland (2023)
22. Kowalski, R., Satoh, K.: Obligations as optimal goal satisfaction, Journal of Philosophical Logic, 47(4), 579-609 (2018)

23. Kowalski, R.: Computational Logic and Human Thinking: How to be Artificially Intelligent. Cambridge University Press (2011)
24. Laird, J.E., Newell, A., Rosenbloom, P.S.: Soar: An architecture for general intelligence. Artificial intelligence, 33(1), 1-64 (1987)
25. Levesque, H.J., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. The Journal of Logic Programming, 31(1), 59-83 (1997)
26. Nicolas, J.M., Gallaire, H.: Database: theory vs. interpretation. In: Gallaire, H., Minker, J. (eds.) Logic and Databases, Plenum, New York (1978)
27. Object Management Group: Semantics of Business Vocabulary and Business Rules SBVR™ 1.5 2019 https://www.omg.org/spec/SBVR
28. Paschke, A., Kozlenkov, A.: November. Rule-based event processing and reaction rules. In: International Workshop on Rules and Rule Markup Languages for the Semantic Web, pp. 53-66. Springer, Berlin Heidelberg (2009)
29. Paschke, A.: ECA-LP / ECA-RuleML: A homogeneous event-condition-action logic programming language, In: Int. Conf. on Rules and Rule Markup Languages for the Semantic Web (RuleML'06) (2006)
30. Przymusinski, T.: On the declarative semantics of stratified deductive databases and logic programs. In: Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, J. Minker (Ed.) 193 – 216. (1987)
31. Rao, A.S.: AgentSpeak (L): BDI agents speak out in a logical computable language. In: European workshop on modelling autonomous agents in a multi-agent world, pp. 42-55 Springer, Berlin Heidelberg (1996)
32. Sadri, F., Kowalski, R.: A theorem-proving approach to database integrity. In: Foundations of deductive databases and logic programming, pp. 313-362. Morgan Kaufmann (1988)
33. Shanahan, M.: Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia. MIT Press (1997)
34. Stenning, K., van Lambalgen, M.: Human Reasoning and Cognitive Science. MIT Press (2012)
35. Thagard, P.: Mind: Introduction to Cognitive Science. Second Edition. MIT Press (2005)
36. Wason, P.C.: Reasoning about a rule. The Quarterly Journal of Experimental Psychology, 20:3, 273-281 (1968)
37. Wielemaker, J., Riguzzi, F., Kowalski, R.A., Lager, T., Sadri, F., Calejo, M.: Using SWISH to realize interactive web-based tutorials for logic-based languages. Theory and Practice of Logic Programming, 19(2), 229-261 (2019)