

Towards a Logic-Based Unifying Framework for Computing

ROBERT KOWALSKI, Imperial College London
FARIBA SADRI, Imperial College London

Computer Science today lacks a unified view of Computing. Instead, the main subareas, most notably artificial intelligence, database systems and programming languages, offer different and often competing approaches to knowledge representation, problem-solving and computation.

In this paper we propose a unifying logic-based, framework for Computing, inspired by artificial intelligence, but scaled down for practical database and programming applications. Computation in the framework is viewed as the task of generating a sequence of state transitions, with the purpose of making an agent's goals all true. States are represented by sets of atomic sentences (or facts), representing the values of program variables, tuples in a coordination language, facts in relational databases, or Herbrand models. In the model-theoretic semantics, the entire sequence of states and events are combined into a single model-theoretic structure, by associating time stamps with facts. But in the operational semantics, facts are updated destructively, without time stamps. We show that the model generated by destructive updates is identical to the model generated by reasoning with facts containing time stamps. We also extend the model with intentional predicates and composite event predicates defined by logic programs containing conditions in first-order logic (FOL). We extend the notions of local stratification and weak stratification to generate the associated model.

Keywords: state transition system, reactive systems, composite events, model generation, frame problem, FOL-stratification, FOL-perfect model

1. INTRODUCTION

Computing today, as a scientific discipline, lacks a unifying framework. It consists, instead, of diverse techniques in such various areas as artificial intelligence, databases and programming. Logic programming was an early attempt to provide a unifying framework for computing, based on the use of logic for knowledge representation and problem-solving. Arguably, this attempt had only limited success, because it failed to address adequately the fundamental role of state transition systems in computing.

In this paper, we present a candidate unifying framework for computing, based on the use of logic for representing state transition systems. Although the approach has its origins in research about representing and reasoning about states, actions and events in artificial intelligence, it has been scaled-down to make it more like conventional programming languages and database and information management systems.

Earlier versions of the work presented in this paper have been presented in [Kowalski and Sadri 2009; Kowalski and Sadri 2010; Kowalski and Sadri 2011; Kowalski and Sadri 2012a; Kowalski and Sadri 2012b]. In those papers, we referred to the framework as LPS, to highlight its focus on providing a Logic-based approach to Production Systems. In this paper, for the sake of continuity, we retain the name LPS, although the intended applications of the approach have been extended considerably. These applications include its use as an agent programming language, active database language, and a language for programming concurrent systems. It can also be used for teleo-reactive computing [Nillson 2001; Kowalski and Sadri 2012a], composite (or complex) event processing, and complex processes.

An LPS framework $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$ represents the goals and beliefs of a single agent embedded in a global environment. The reactive rules \mathbf{R} , which represent the agent's maintenance goals, have the logical form of material implications $\forall X [\textit{antecedent} \rightarrow \exists Y \textit{consequent}]$. The logic program \mathbf{L} represents the agent's view of the changing state and the state transforming events. The domain theory \mathbf{D} ,

which also has the form of a logic program, is a causal theory, which specifies the effect of concurrent sets of events on the state of the environment.

The global environment and the agent’s own internal state are combined in a separate, single state S , represented by a set of atomic sentences, called *facts* or *fluents*. The state is like a relational database, but also like a set of program variables or tuples in a coordination language [Carriero and Gelernter 1989].

State transitions are generated by sets of concurrently occurring events. These events can include both the simultaneous actions of a single agent, the simultaneous actions of several agents, or external events of other origin. Their preconditions and postconditions are specified by the domain theory D . The domain theory D is similar to the situation calculus [McCarthy and Hayes 1969] or event calculus [Kowalski and Sergot 1986], but instead of using frame axioms to update states, it performs destructive updates on the current state, deleting facts that are terminated by the set of all events occurring in the state transition and adding facts that are initiated by the set of events.

The logic program $L = L_{int} \cup L_{events} \cup L_{timeless} \cup L_{temp}$ performs a supporting role.

L_{int} defines intensional predicates in terms of extensional predicates,
 L_{events} defines composite events in terms of state conditions and simpler events,
 $L_{timeless}$ defines time independent predicates, and
 L_{temp} defines temporal relationships, such as successor and inequality relations between time points.

The logic program $S \cup L_{int} \cup L_{timeless}$ can be viewed as a deductive database state. The logic program $L_{events} \cup L_{temp} \cup L_{timeless}$ superimposes a collection of composite events on top of the sequence of successive states and simple events. These composite events can be viewed as state-connecting paths, as in transaction logic [Bonner and Kifer 1993].

The reactive rules R , logic programs L and domain theory D can contain state conditions that are formulas of first-order logic (FOL), which are like FOL queries to a relational or deductive database. The semantics of these FOL conditions is given by generalising negative literals to FOL conditions in the definitions of local stratification [Przymusinski 1987] and weak stratification [Przymusinska and Przymusinski 1988].

Although the operational semantics employs destructive updates and destructive assignment to maintain only a current state, the model-theoretic semantics is defined relative to the entire sequence of states, events and paths combined into a single model in which facts and events are time-stamped. In the model-theoretic semantics, the *computational task* is to generate a model that makes the reactive rules R all true. The logic program L contributes to the definition of the model by adding intensional facts and composite events to the sequence of time-stamped extensional facts and simple events.

In this paper, we show that, given the same initial state and the same sequence of state-transforming events, the model generated by LPS using destructive updates is identical to the “natural” model generated by using a frame axiom. We define a generalization of weak stratification in order to specify this natural model.

The view of computation in LPS as model generation needs to be distinguished from the use of model checking for proving program properties. In a sense, the reactive rules are program properties that are represented explicitly in the program and are used operationally in the attempt to generate a model that makes them true. In [Kowalski and Sadri 2012a], we showed how to derive a program property

that is not represented explicitly. The derivation uses ordinary deduction, presented as a proof tree. It would be interesting to investigate the use of model checking techniques for the same purpose.

The paper is organised as follows: Section 2 introduces the framework informally by means of examples, and aims to give a flavour of the breadth of its applications. Then sections 3-6 investigate the framework more formally. Section 3 defines the language, section 4 introduces the model theory, section 5 presents the operational semantics, section 6 gives further details of the model theory, and section 7 discusses soundness and completeness. Section 8 shows that destructive updates in LPS generate the same models as the frame axiom. In sections 9 and 10, we discuss related and future work.

Compared with earlier papers, the main contributions of this paper are its more rigorous treatment of the semantics of reactive rules and logic programs with FOL conditions, and its demonstration of the relationship between the frame axiom and destructive updates. We also present a preliminary approach to the treatment of concurrent events.

2. EXAMPLES

2.1 Emergencies

The following example is a variant of an example in [Hausmann et al. 2012]. In this example, a reactive agent monitors a building for outbreaks of fire. The agent receives inputs from a heat sensor and a smoke detector. If these inputs are sufficiently close together in time, then the agent recognises a possible fire, and attempts to deal with it. There are two alternative ways of dealing with the possible fire. One alternative is to activate local fire suppression devices and then to call for a security guard to inspect the area. The other alternative is simply to call the fire department.

The example illustrates several features of LPS, including reactivity, composite events, composite processes and non-determinism. The representation is an LPS framework $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$ consisting of a main program \mathbf{R} , which contains a single reactive rule, and a logic program \mathbf{L} . However, S , L_{int} and the domain theory \mathbf{D} are all empty. In an elaboration of the example, $S \cup L_{int} \cup L_{timeless}$ could include such additional information as the geography of the building and the location, identity and capabilities of the security guards. Here the variables T and T_i all represent time points. The predicates *heat-sensed* and *smoke-detected* represent input events taking place over short periods of time $[Tf_1, Tf_2]$ and $[Ts_1, Ts_2]$, respectively.

We use the forward arrow \rightarrow for the implication symbol in reactive rules, and the backward arrow \leftarrow in logic programming clauses. As in Prolog, identifiers beginning with an upper case letter denote variables, and numbers or identifiers beginning with a lower case letter denote constants.

$$\begin{array}{l}
 \mathbf{R} \quad \text{heat-sensed}(\text{Area}, Tf_1, Tf_2) \wedge \text{smoke-detected}(\text{Area}, Ts_1, Ts_2) \wedge \\
 \quad |Tf_1 - Ts_1| \leq 60 \text{ sec} \wedge \max(Tf_2, Ts_2, T) \\
 \quad \rightarrow \text{fire-response}(\text{Area}, T, T_1, T_2) \wedge T < T_1 \\
 \\
 L_{events} \quad \text{fire-response}(\text{Area}, T, T_1, T_4) \\
 \quad \leftarrow \text{activate-fire-suppression}(\text{Area}, T_1, T_2) \wedge T < T_1 \leq T + 5 \text{ sec} \wedge \\
 \quad \quad \text{send-security-guard}(\text{Guard}, \text{Area}, T_3, T_4) \wedge T_2 < T_3 \leq T_2 + 10 \text{ sec}
 \end{array}$$

$$\begin{aligned} & \text{fire-response}(\text{Area}, T, T_1, T_2) \\ & \leftarrow \text{call-fire-department}(\text{Area}, T_1, T_2) \wedge T < T_1 \leq T + 120 \text{ sec} \end{aligned}$$

All variables in a reactive rule are universally quantified with scope the entire rule, except for variables that are only in the consequent of a rule. These are existentially quantified with scope the consequent of the rule. Analogously, all variables in a logic programming clause that are not explicitly quantified are universally quantified with scope the entire clause. However, variables that occur only in the body of a clause and are not explicitly quantified can also be considered to be existentially quantified with scope the body of the clause.

Notice that in this example the antecedent of the rule represents an unnamed composite event and the consequent represents a named composite action, *fire-response*. This composite action can be regarded as consisting of two alternative plans, each of which is represented by a clause in *Levents*. The first plan consists of two actions, and the second plan consists of a single action.

For simplicity in this example, the events *heat-sensed*, *smoke-detected*, *activate-fire-suppression*, *send-security-guard* and *call-fire-department* are all simple events, which are either directly observed in the environment or directly performed as simple actions. In general, events that are not defined in *Levents* are simple events.

Both plans are temporally constrained. In practice, the first plan might be preferred and tried before the second. If any part of the first plan fails, then the second plan can be tried. Moreover, even if the first plan fails, it can be retried as long as the temporal constraints can be satisfied. If both plans fail and cannot be retried, then the reactive rule cannot be made true. This can be avoided by adding additional alternative plans for the consequent of the rule. Notice that the temporal constraints in the logic program ensure that, if the first plan takes too long, then the second plan, which involves calling the fire department, can still be tried.

In this example, as in many others, there are potentially many different models that can be generated to make the reactive rule true, some of which are preferable to others. In theory, LPS could be augmented with a control component that decides what actions to perform with a view towards optimising the utility of the resulting model. However, in practice, it is probably sufficient for the programmer simply to order the alternatives, taking both time constraints and preferred outcomes into account.

We will see later that LPS is incomplete because it can only generate models that make the consequents of reactive rules true when their antecedents become true. It cannot preventatively make a reactive rule true by making its antecedent false, and it cannot proactively make its consequent true in anticipation of its antecedent becoming true in the future.

We will also see later that the explicit representation of time is necessary for the model-theoretic semantics. Moreover, it facilitates the representation and processing of temporal constraints on the timing of fluents and events. However, we will also see that it can be hidden in an external syntax.

2.2 Dialogue

In this example, an agent “me” attempts to make a reactive rule true, by generating an output sentence whenever it receives an input sentence from the agent “you”. The predicate *sentence*(*Agent*, *T*₁, *T*₂) represents a composite event taking place from time *T*₁ to time *T*₂.

R $sentence(you, T_1, T_2) \rightarrow sentence(me, T_3, T_4) \wedge T_2 < T_3 \wedge T_3 \leq T_2 + 3 \text{ sec}$

The temporal constraint in the consequent of the rule indicates that the composite output action $sentence(me, T_3, T_4)$ needs to start within 3 seconds of the completed input.

The utterance by an agent A of a word W is treated as a simple (or atomic) event that takes place over an interval of time, and is represented by an atomic sentence $word(A, W, T_1, T_2)$. The interval is represented by its start and end times, T_1 and T_2 , respectively.

We assume that the language of LPS is sorted (or typed). In this example, the language would contain disjoint sorts for *agents*, *words*, and *time points*; and the different arguments of the predicate $word$ would be restricted to terms of the appropriate sort.

The following sequence of input events represents the utterance by agent “you” of the stream of words “what is your name”. For simplicity, time points are represented by positive integers:

$word(you, what, 1, 2)$ $word(you, is, 2, 3)$
 $word(you, your, 3, 4)$ $word(you, name, 4, 5)$

Composite events of uttering nouns, noun phrases, sentences and other parts of speech are represented by means of logic programs in L_{events} . These programs are similar to the logical representation of definite clause grammars [Pereira and Warren 1980] in Prolog, and used in [Kowalski 1979] to illustrate parsing as reasoning:

L_{events} $adjective(Agent, T_1, T_2) \leftarrow word(Agent, my, T_1, T_2)$
 $adjective(Agent, T_1, T_2) \leftarrow word(Agent, your, T_1, T_2)$

$noun(Agent, T_1, T_2) \leftarrow word(Agent, name, T_1, T_2)$
 $noun(Agent, T_1, T_2) \leftarrow word(Agent, fariba, T_1, T_2)$
 $noun(Agent, T_1, T_2) \leftarrow word(Agent, what, T_1, T_2)$
 $verb(Agent, T_1, T_2) \leftarrow word(Agent, is, T_1, T_2)$

$sentence(Agent, T_1, T_3) \leftarrow noun-phrase(Agent, T_1, T_2) \wedge verb-phrase(Agent, T_2, T_3)$
 $noun-phrase(Agent, T_1, T_3) \leftarrow adjective(Agent, T_1, T_2) \wedge noun(Agent, T_2, T_3)$
 $noun-phrase(Agent, T_1, T_2) \leftarrow noun(Agent, T_1, T_2)$
 $verb-phrase(Agent, T_1, T_3) \leftarrow verb(Agent, T_1, T_2) \wedge noun-phrase(Agent, T_2, T_3)$
 $verb-phrase(Agent, T_1, T_2) \leftarrow verb(Agent, T_1, T_2)$

In this example, S and L_{int} are empty. Notice that the logic program L_{events} does not distinguish between recognising sentences and generating them.

The operational semantics of LPS, which we define later in paper, reasons forwards from the antecedents of rules to their consequents, and evaluates simple events and FOL state conditions in the rules in their temporal order. It decomposes composite events in the consequents of rules top-down into conditions and simpler events. But it is neutral with respect to the evaluation of composite events in the antecedents of rules, and it is neutral with respect to whether state conditions are evaluated top-down (backwards) or bottom-up (forwards).

Given the above sequence of input events, one way of satisfying the top-level goal, represented by the reactive rule, is to generate the following sequence of actions:

<i>word(me, my, 6, 7)</i>	<i>word(me, name, 7, 8)</i>
<i>word(me, is, 8, 9)</i>	<i>word(me, fariba, 9, 10)</i>

Assuming that there are no further sentences uttered by “you”, the reactive rule is true in the resulting Herbrand model:

<i>{word(you, what, 1, 2)</i>	<i>noun(you, 1, 2)</i>	<i>noun-phrase(you, 1, 2)</i>
<i>word(you, is, 2, 3)</i>	<i>verb(you, 2, 3)</i>	<i>noun-phrase(you, 3, 5)</i>
<i>word(you, your, 3, 4)</i>	<i>adjective(you, 3, 4)</i>	<i>noun-phrase(you, 3, 4)</i>
<i>word(you, name, 4, 5)</i>	<i>noun(you, 4, 5)</i>	<i>verb-phrase(you, 2, 5)</i>
<i>sentence(you, 1, 3)</i>	<i>sentence(you, 1, 5)</i>	
<i>word(me, my, 6, 7)</i>	<i>adjective(me, 6, 7)</i>	<i>noun-phrase(me, 6, 8)</i>
<i>word(me, name, 7, 8)</i>	<i>noun(me, 7, 8)</i>	<i>noun-phrase(me, 7, 8)</i>
<i>word(me, is, 8, 9)</i>	<i>verb(me, 8, 9)</i>	<i>noun-phrase(me, 9, 10)</i>
<i>word(me, fariba, 9, 10)</i>	<i>noun(me, 9, 10)</i>	<i>verb-phrase(me, 8, 10)</i>
<i>sentence(me, 7, 9)</i>	<i>sentence(me, 7, 10)</i>	<i>sentence(me, 6, 10) ∪ Temp</i>

Here *Temp* is the extension of the temporal inequality relation defined by L_{temp} . Notice that the grammar would need to be refined to avoid concluding *sentence(you, 1, 3)*, *sentence(me, 7, 9)*, *sentence(me, 7, 10)*, which are intuitively unintended.

In some of the earlier versions of LPS, we treated events as instantaneous and states as having duration. However, in the remainder of this paper, we do the opposite and treat states as instantaneous and events as having duration. The different treatments are different ways of ensuring that the truth value of a fluent does not change within a state, but changes only in the transition from one state to the next. The general approach of LPS is compatible with both treatments of time.

2.3 Blocks world

In the previous examples, there are no internal states. Instead, events come and go without leaving any trace. This is not typical of programs in LPS.

The blocks world illustrates the more typical case of a program with an internal state. In this example, the state can be viewed as the extensional part of a deductive database, in which *on(Block, Place, T)* is an extensional predicate, and *clear(Place, T)* is an intensional predicate. We assume that the language is order-sorted, with sorts for *blocks*, *places* and *time points*. The sort *places* includes the sort *blocks* and contains in addition the constant *table*.

Assuming, for simplicity, that the table is always clear, the predicate *clear(Place, T)* is defined by two clauses in L_{int} :

$$L_{int} \quad \begin{array}{l} \text{clear}(\text{table}, T) \\ \text{clear}(\text{Block}, T) \leftarrow \neg \exists X \text{on}(X, \text{Block}, T) \end{array}$$

We treat negative literals in the body of a clause as a special case of an FOL condition. Operationally, an FOL condition is a query to predicates defined at a lower stratum than the stratum of the predicate defined in the head of the clause. Clauses satisfying this condition (which we call FOL-stratification and define formally later) are a natural generalisation of locally stratified logic programs. In

this example, there are two strata: The lower stratum corresponds to the extensional part of the database, and the upper stratum corresponds to the intensional part. More generally, as we will see later, the intensional predicates can be spread over several strata.

In general, events, including the agent's own actions and external events, update only the extensional part of a database state. The intensional fluents are updated implicitly, as a result of updates to the extensional part.

In general, the set of simple events taking place from T_1 to T_2 generate a state transition from the state whose fluents all hold at time T_1 into the next state whose fluents all hold at time T_2 . In this example, the updates are associated with the simple event $move(Block, Place, T_1, T_2)$, defined by the domain theory \mathbf{D} . This is similar to the treatment of events in the event calculus. We assume that the arguments of $move$ are typed, so that it is not possible to move the *table*.

As in the event calculus, it is convenient to represent time-stamped fluents and events by means of meta-predicates, writing for example $holds(p, t)$ instead of $p(t)$ and $happens(e, t_1, t_2)$ instead of $e(t_1, t_2)$. We also combine both notations interchangeably.

In general \mathbf{D} consists of two parts, D_{post} which is concerned with the post-conditions of events and D_{pre} which is concerned with the preconditions of events. In this example D_{post} specifies the fluents that are initiated and terminated by the simple event $move(Block, Place)$, and is given below. D_{pre} will be described later.

$$D_{post} \quad \begin{aligned} &initiated(on(Block, Place), T_1, T_2) \leftarrow happens(move(Block, Place), T_1, T_2) \\ &terminated(on(Block, Support), T_1, T_2) \leftarrow happens(move(Block, Place), T_1, T_2) \wedge \\ &\quad on(Block, Support, T_1) \end{aligned}$$

The task of putting a block on a place can be triggered by an input event requesting that the block be put on the place, represented by a reactive rule:

$$\mathbf{R} \quad request(on(Block, Place), T_1, T_2) \rightarrow make-on(Block, Place, T_3, T_4) \wedge T_2 \leq T_3$$

Here $request(on(Block, Place), T_1, T_2)$ is a simple, external event, which does not initiate or terminate any fluents. Alternatively, it could initiate a fluent that stores a record of the request.

In a more realistic example, additional conditions might be included in the antecedent of the rule, for example to check that the request is authorised, and additional constraints might be included in the consequent of the rule to constrain the amount of time that can elapse between the request and its fulfilment.

The predicate $make-on$ can be defined as a composite action/event, using an auxiliary composite action/event $make-clear$:

$$L_{events} \quad \begin{aligned} &make-on(Block, Place, T, T) \leftarrow on(Block, Place, T) \\ &make-on(Block, Place, T_1, T_4) \leftarrow \neg on(Block, Place, T_1) \wedge \\ &\quad make-clear(Block, TB_1, TB_2) \wedge make-clear(Place, TP_1, TP_2) \wedge \\ &\quad move(Block, Place, T_3, T_4) \wedge T_1 \leq TB_1 \wedge T_1 \leq TP_1 \wedge TB_2 \leq T_3 \wedge TP_2 \leq T_3 \\ &make-clear(Place, T, T) \leftarrow clear(Place, T) \\ &make-clear(Place, T_1, T_4) \leftarrow Place \neq table \wedge on(Block, Place, T_1) \wedge \\ &\quad make-clear(Block, T_1, T_2) \wedge move(Block, table, T_3, T_4) \wedge T_2 < T_3 \end{aligned}$$

Notice that the two $make-clear$ events in the body of the second $make-on$ clause are partially ordered. They can be performed in any order, as well as at the same time,

for example by using two hands. Notice, moreover, that, it might be desirable to add an extra condition to the body of the clause, constraining the *make-clear* events so that one of them starts as soon as the condition $\neg on(Block, Place, T_1)$ is verified.

The clauses in L_{events} are written in a teleo-reactive style, and have a base case corresponding to the goal of the composite action. Teleo-reactive programs facilitate both the re-execution of plan subgoals when plans fail, and the omission of plan subgoals when the environment opportunistically solves these subgoals instead. These features of teleo-reactive programs apply also to LPS programs more generally. We present an example later in this subsection.

In an earlier version of LPS [Kowalski and Sadri 2011] we allowed “planning clauses”, which in the case of *make-clear* could have the form:

$$\begin{aligned} clear(Place, T_3) \leftarrow & on(Block, Place, T_1) \wedge clear(Block, T_1) \wedge \\ & move(Block, table, T_2, T_3) \wedge T_1 < T_2 \end{aligned}$$

Such planning clauses allow a higher level of knowledge representation than the version of LPS presented in [Kowalski and Sadri 2012a; Kowalski and Sadri 2012b] and in this paper. However, the behaviour of programs containing planning clauses is more complicated and harder to understand.

It is possible to transform planning clauses into composite event definitions and vice versa. In any case, it is important to appreciate that LPS programs do not perform planning from first principles, but simply execute pre-existing, explicitly represented conditional plans. This is similar to plans in practical BDI agent languages.

In earlier papers, we allowed only one action to be executed at a time. In this paper, we allow concurrent actions and other events. In the blocks world, in particular, we can allow several agents to pick up blocks concurrently, in which case an action by one agent becomes an external event for another agent.

In the simplest case, concurrent events are independent, and their combined effects are simply the collection of their individual effects. However, in other cases, concurrent events can have combined effects that are different from the effects of their individual events. For example, two concurrent events of picking up two ends of a table have different combined effects from the individual events of picking up only one end of the table.

In this example, we consider the case of competing events, which are prevented from occurring concurrently by means of integrity constraints in D_{pre} . These integrity constraints have the form of logic programming clauses with head *false*; but as we will see later, they play a different role in the semantics from other logic programming clauses.

$$\begin{aligned} D_{pre} \quad & false \leftarrow happens(move(Block, Place), T_1, T_2) \wedge \\ & \neg [clear(Block, T_1) \wedge clear(Place, T_1)] \\ & false \leftarrow happens(move(Block, Place), T_1, T_2) \wedge Block = Place \\ & false \leftarrow happens(move(Block, Place_1), T_1, T_2) \wedge \\ & \quad happens(move(Block, Place_2), T_1, T_2) \wedge Place_1 \neq Place_2 \\ & false \leftarrow happens(move(Block_1, Place), T_1, T_2) \wedge \\ & \quad happens(move(Block_2, Place), T_1, T_2) \wedge Block_1 \neq Block_2 \wedge Place \neq table \\ & false \leftarrow happens(move(Block_1, Block_2), T_1, T_2) \wedge \\ & \quad happens(move(Block_2, Place), T_1, T_2) \end{aligned}$$

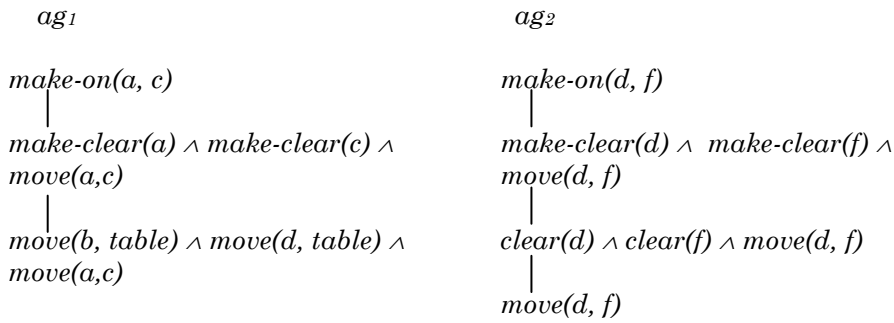
The integrity constraints in D_{pre} also represent the preconditions of individual events, as exemplified by the first two constraints above.

Notice that, given three candidate events, say $move(a, b)$, $move(c, d)$ and $move(d, e)$ each of which is possible in isolation, but not in combination, there are two possible sets of two concurrent events, $\{move(a, b), move(c, d)\}$, $\{move(a, b), move(d, e)\}$, three possible singleton sets $\{move(a, b)\}$, $\{move(c, d)\}$, $\{move(d, e)\}$ and the empty set $\{\}$.

As a concrete example, consider two agents ag_1 and ag_2 with initial goals: $make-on(a, c, T_1, T_2) \wedge T_1 > 0$, and $make-on(d, f, T_3, T_4) \wedge T_3 > 0$, respectively. Let the initial state be as pictured:

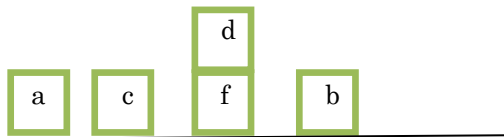


The two agents can attempt to achieve their individual goals, first by incrementally generating a tree of subgoals, each. Here we ignore the time parameters for ease of reading:



The tree has other branches, but we assume either that they have already been explored and failed, or that they can be explored in the future.

Having generated these branches, suppose agent ag_1 selects candidate actions $\{move(b, table), move(d, table)\}$ and ag_2 selects candidate action $\{move(d, f)\}$ for execution at the same time – other variations are also possible. The three actions are not possible concurrently. Suppose the successful actions are $move(b, table)$ and $move(d, f)$, i.e. one action is successful for each agent. The state will be transformed into the following:



The initial goal of agent ag_2 has been achieved. However, only the first subgoal of the current plan of agent ag_1 has been achieved. Agent ag_1 can continue with its current plan, in which case it will unnecessarily retry the action $move(d, table)$. Or it

can take advantage of the opportunistic solution by agent ag_2 of the higher-level subgoal $make-clear(c)$ in the higher-level, more abstract plan $make-clear(a) \wedge make-clear(c) \wedge move(a,c)$, and solve its initial goal, by performing the action $move(a,c)$ in one step.

The model-theoretic semantics of LPS defines computation as the task of generating a model that makes an agent's reactive rules and initial goals all true. It is not concerned with optimising the generated model. In the example above, the chosen solution is a model that contains the fewest actions. In other applications, it may be desirable to generate a model that maximizes concurrency, as in the following example.

2.4 The Dining Philosophers

Although the dining philosophers' problem was originally proposed as a problem of concurrent programming, the problem can also be viewed in both database and AI knowledge representation and problem-solving terms:

In the initial state, five philosophers sit around a circular table with a bowl of spaghetti in the middle of the table and five forks, one to the left and one to the right of each philosopher. Each philosopher alternates between thinking and eating. In order to eat, a philosopher needs two adjacent forks. A philosopher can pick up the two adjacent forks if they are both available.

The setting of the problem is similar to that of the blocks world. The forks are like blocks, having a fork is like there being an object on a block, and a fork being available is like a block being clear. Similar constraints apply to picking up a fork as apply to picking up a block: A fork cannot be picked up and moved simultaneously to two different locations (philosophers).

The solution of the dining philosophers' problem presented below is similar to the solution in C-Linda [Carriero and Gelernter 1989]. Here we assume that the five philosophers are represented by five separate processors or agents, each of which has its own local copy of the same framework $\langle R, L, D \rangle$, but shares a single copy of the global state. The global state acts as a coordination medium, which non-deterministically decides which possible sets of concurrent events actually occur.

The only extensional predicate in the representation is the predicate $available(Fork, T)$, and $L_{int} = \{\}$. In the initial state S_0 , all five forks are available. To facilitate updating the state destructively, the extensional fluents are represented without time parameters:

$$S_0 = \{available(fork(0)), available(fork(1)), available(fork(2)), available(fork(3)), available(fork(4))\}.$$

For simplicity the predicate $adjacent(fork(0), philosopher(0), fork(1))$ is treated as time-independent:

$$\begin{array}{ll} L_{timeless} & adjacent(fork(0), philosopher(0), fork(1)) \\ adjacent(fork(1), philosopher(1), fork(2)) & adjacent(fork(2), philosopher(2), fork(3)) \\ adjacent(fork(3), philosopher(3), fork(4)) & adjacent(fork(4), philosopher(4), fork(0)) \end{array}$$

The extensional fluents are like tuples in a relational database, or in a coordination language [Carriero and Gelernter 1989]. They are also like the values of "variables"

in conventional programming languages. For example, updating the state by adding the fluent $available(fork(f))$ is like executing an assignment statement $available(fork(f)) := true$, assigning the value $true$ to the “variable” $available(fork(f))$.

The solution in C-Linda uses four tickets. To eat, a philosopher needs not only two forks but also one ticket. This ensures that at least one philosopher is always able to eat. We have programmed this solution in LPS, but here we present a solution using a simple action of picking up two adjacent forks simultaneously.

Each philosopher, $philosopher(i)$, $0 \leq i \leq 4$, can perform four simple actions, of *thinking*, *picking up forks*, *eating*, and *putting down forks*. These can be combined into a single composite action, *dine*:

$$\begin{aligned}
L_{events} \quad & dine(philosopher(i), T_1, T_6) \leftarrow think(philosopher(i), T_1, T_2) \wedge \\
& adjacent(F_1, philosopher(i), F_2) \wedge \\
& pickup-forks(F_1, philosopher(i), F_2, T_3, T_4) \wedge T_2 \leq T_3 \wedge \\
& eat(philosopher(i), T_4, T_5) \wedge putdown-forks(F_1, philosopher(i), F_2, T_5, T_6)
\end{aligned}$$

Notice that the temporal constraints here allow a lapse of time between thinking and picking up forks, because forks may not be available as soon as thinking is completed. But no such lapse of time takes place between picking up forks and eating and between eating and putting down forks.

The composite action *dine* can be triggered by means of a reactive rule and an event of it becoming time to eat:

$$\mathbf{R} \quad time-to-eat(philosopher(i), T_1, T_2) \rightarrow dine(philosopher(i), T_3, T_4) \wedge T_2 \leq T_3$$

For example, the five instances of the rule could be triggered by the five concurrent external events:

$$\begin{aligned}
ev_1 = \{ & time-to-eat(philosopher(0), 0, 1), & time-to-eat(philosopher(1), 0, 1), \\
& time-to-eat(philosopher(2), 0, 1), & time-to-eat(philosopher(3), 0, 1), \\
& time-to-eat(philosopher(4), 0, 1) \}
\end{aligned}$$

Additional temporal constraints could be imposed on the consequent of the rule, to try to prevent the philosophers from starving while waiting to eat.

The solution of the dining philosophers’ problem is shared between the reactive rules and logic programs used by the individual philosophers and the domain theory used to update the global state. For the philosophers’ part, it suffices for each philosopher to employ the one reactive rule in \mathbf{R} and the one logic programming clause in L_{events} . The domain theory \mathbf{D} defines the post-conditions and preconditions of the simple atomic actions. In this simple formulation of the problem, the only actions that change the state are the actions of picking up and putting down forks:

$$\begin{aligned}
D_{post} \quad & terminated(available(F), T_1, T_2) \leftarrow \\
& happens(pickup-forks(F_1, philosopher(I), F_2), T_1, T_2) \wedge (F = F_1 \vee F = F_2) \\
& initiated(available(F), T_1, T_2) \leftarrow \\
& happens(putdown-forks(F_1, philosopher(I), F_2), T_1, T_2) \wedge (F = F_1 \vee F = F_2)
\end{aligned}$$

$$\begin{aligned}
D_{pre} \quad & false \leftarrow happens(pickup-forks(F_1, philosopher(I), F_2), T_1, T_2) \wedge \\
& \neg [available(F_1, T_1) \wedge available(F_2, T_1)] \\
& false \leftarrow happens(pickup-forks(F_1, philosopher(I), F), T_1, T_2) \wedge \\
& happens(pickup-forks(F, philosopher(J), F_2), T_1, T_2)
\end{aligned}$$

The first constraint in D_{pre} ensures that a philosopher picks up forks only if they are available, and the second constraint ensures that two philosophers I and J do not pick up the same fork F simultaneously.

In general, when a simple event happens, then all fluents initiated by the event are added and all fluents terminated by the event are deleted in the state transition. In this sense, simple events are *atomic*, in that either all their effects succeed or all their effects fail at the same time. The domain theory D and its use for defining state transitions provides a declarative semantics for such forms of atomicity, with details concerning roll-back and critical sections relegated to the implementation.

The model-theoretic and operational semantics, defined later, are the semantics of a single agent, say agent 0 for example, interacting with a single current state. In the dining philosopher's problem, this single state is a global state shared with other philosophers. The domain theory is used to update this state.

In order to generate the external events for agent 0 and to correctly update the global state, it is necessary to simulate the other agents and to generate their candidate actions, which are external events for agent 0 . It is also necessary to arbitrate between competing candidate actions and to choose a set of concurrent actions that satisfy the integrity constraints in D_{pre} . This choice of concurrent actions is non-deterministic, but one such sequence of choices is shown below. In fact, this is the sequence generated by our Prolog prototype of LPS.

Here S_0, S_1, \dots, S_{12} is the sequence of states, and ev_1, \dots, ev_{12} is the sequence of events, where ev_i is the set of all events that take place concurrently in the transition from state S_{i-1} to state S_i :

$$\begin{aligned}
S_2 &= S_1 = S_0 \\
ev_2 &= \{\} \\
ev_3 &= \{think(philosopher(0)), think(philosopher(1)), think(philosopher(2)), \\
&\quad think(philosopher(3)), think(philosopher(4))\} \\
S_3 &= S_2 \\
ev_4 &= \{pickup-forks(fork(0), philosopher(0), fork(1)), \\
&\quad pickup-forks(fork(2), philosopher(2), fork(3))\} \\
S_4 &= \{available(fork(4))\} \\
ev_5 &= \{eat(philosopher(0)), eat(philosopher(2))\} \\
S_5 &= S_4 \\
ev_6 &= \{putdown-forks(fork(0), philosopher(0), fork(1)), \\
&\quad putdown-forks(fork(2), philosopher(2), fork(3))\} \\
S_6 &= S_2 = S_1 = S_0 \\
ev_7 &= \{pickup-forks(fork(1), philosopher(1), fork(2)), \\
&\quad pickup-forks(fork(3), philosopher(3), fork(4))\} \\
S_7 &= \{available(fork(0))\} \\
ev_8 &= \{eat(philosopher(1)), eat(philosopher(3))\} \\
S_8 &= S_7 \\
ev_9 &= \{putdown-forks(fork(1), philosopher(1), fork(2)), \\
&\quad putdown-forks(fork(3), philosopher(3), fork(4))\} \\
S_9 &= S_6 = S_2 = S_1 = S_0 \\
ev_{10} &= \{pickup-forks(fork(4), philosopher(4), fork(0))\} \\
S_{10} &= \{available(fork(1)), available(fork(2)), available(fork(3))\} \\
ev_{11} &= \{eat(philosopher(4))\} \\
S_{11} &= S_{10} \\
ev_{12} &= \{putdown-forks(fork(4), philosopher(4), fork(0))\}
\end{aligned}$$

$$S_{12} = S_9 = S_6 = S_2 = S_1 = S_0$$

In the model-theoretic semantics, all the events and states are combined into a single Herbrand model M by adding an extra temporal argument to fluents and events. The model M also contains the extension of the temporal inequality relation \leq and the composite actions:

$$\begin{array}{lll} dine(philosopher(0), 2, 6) & dine(philosopher(2), 2, 6) & dine(philosopher(1), 2, 9) \\ dine(philosopher(3), 2, 9) & dine(philosopher(4), 2, 12) & \end{array}$$

The operational semantics correctly generates a model M in which the reactive rule \mathbf{R} and the integrity constraints D_{pre} are all true. The logic program \mathbf{L} is used to help generate the model, and therefore is also true in M .

Note that even though the global state has information about the availability of all five forks, an individual agent may have access only to information about the availability of forks adjacent to it. In fact the agent can function simply by relying on the environment to tell it when its actions, including picking up forks, are successful.

3. THE LANGUAGE

An LPS program $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$ combines the reactive rules \mathbf{R} and logic programs \mathbf{L} of an individual agent with a domain theory \mathbf{D} that is used to perform updates on a current state S_i that is possibly shared with other agents. The shared state S_i arbitrates between the candidate actions of different agents, non-deterministically choosing a set ev_i of concurrent events that satisfies the integrity constraints in D_{pre} , and using D_{post} to transform S_{i-1} into S_i .

The agent receives observations of the events ev_i and can query the current state S_i to generate candidate actions $acts_{i+1}$ with the purpose of making the reactive rules \mathbf{R} all true. The truth value of \mathbf{R} is determined with respect to a natural, minimal model associated with the entire sequence of states S_0, \dots, S_i, \dots and events ev_1, \dots, ev_i, \dots augmented with higher-level predicates defined by the logic program $\mathbf{L} = L_{int} \cup L_{events} \cup L_{timeless} \cup L_{temp}$.

The reactive rules \mathbf{R} , logic programs \mathbf{L} and \mathbf{D} can contain FOL conditions, which operationally query the extended current state. The model-theoretic semantics for FOL conditions generalises the perfect model semantics for negative conditions. It generalises local stratification, by restricting the strata of predicates in FOL conditions in the body of a clause to ones that are lower than the stratum of the predicate occurring in the head. It generalises the perfect model semantics, by generating submodels whose predicates belong to lower strata before using them to generate models whose predicates belong to higher strata.

States S_i are sets of atomic sentences (or simple fluents), which can be viewed as the extensional component of a deductive database. The domain theory \mathbf{D} updates only these extensional predicates. The logic program L_{int} implicitly updates intensional predicates as ramifications of changes to the extensional predicates. The logic program L_{events} both recognises and constructs composite events.

In the operational semantics, fluents are represented without explicit time. Updates associated with a set ev_i of events are performed destructively, using the domain theory \mathbf{D} to delete fluents that are terminated by ev_i and to add fluents that are initiated by ev_i . Fluents that are neither initiated nor terminated simply persist without reasoning that they persist, and without copying them explicitly from one state into the next.

In a possible world semantics, as employed in modal logic or TR logic, states are also represented by sets of fluents, but they are linked by an accessibility relation associated with the state transforming events. However, in the model-theoretic semantics of LPS, fluents and events are time-stamped and combined in a single, non-modal model-theoretic structure.

In the version of LPS presented in this paper, we assume that time is linear and discrete, and that the succession of time points is represented by the relation $succ(s, t)$ defined in L_{temp} (with the aid of $L_{timeless}$) where s and t are real numbers. For simplicity, we assume that there is a state transition from S_{i-1} to S_i associated with each instance $succ(t_{i-1}, t_i)$, that all fluents in S_i hold at time t_i , and that all events in ev_i occur from time t_{i-1} to t_i . The set ev_i can be empty. In other papers, we have made the opposite assumption, associating time points with simple events and time intervals with states. The two conventions are mostly interchangeable.

To distinguish between a fluent p without a time stamp, needed for destructive updates in the operational semantics, and the same fluent with a time stamp t , needed for the model-theoretic semantics, we either add an extra argument $p(t)$ to p , or we treat p as a term and include it in a meta-predicate such as $holds(p, t)$. To distinguish between a state S_i whose fluents are all without time stamps, and the same state in which all the fluents have the same time stamp t_i , we write S_i^* . Similarly, for a simple or composite event e without a time stamp, either we write $e(t_1, t_2)$, or we include e as a term within a meta-predicate such as $happens(e, t_1, t_2)$.

For the concurrent occurrence of an unstamped set ev_i of simple events, occurring from time t_{i-1} to time t_i we write ev_i^* for the same set of events with their time stamps.

3.1 Vocabulary

We assume an order-sorted language in which the constants and variables of the language are assigned sorts that may be hierarchically ordered. The argument places of function symbols and predicate symbols are correspondingly assigned sorts, so that formulas are well-formed only if the argument places are filled by terms of the allowed sort.

The predicate symbols of the language are partitioned into (disjoint) sets representing fluents, events, auxiliary predicates and meta-predicates:

Fluent predicate symbols are partitioned into *extensional predicates*, which represent facts in the states S_i , and *intensional predicates* defined in L_{int} .

Event predicates are analogously partitioned into simple event predicates and composite event predicates. Simple events can represent either externally generated events or internally generated actions. *Composite event predicates* are defined in L_{events} .

Auxiliary predicates consist of time-independent predicates, such as *max* and *min* and others used for arithmetic, defined in $L_{timeless}$, as well as temporal predicates, such as *succ*, defined in L_{temp} .

The *meta-predicates* consist of

- The predicates *initiated* and *terminated* defined by the domain theory \mathbf{D} , and used for performing state transitions; and
- The predicates *holds* and *happens*.

Fluents and events occur as terms when they are arguments of the meta-predicates.

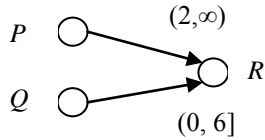
States S_i are not represented explicitly in the language, but are represented implicitly by the set of all the extensional facts that are true at time t_i . These extensional facts represent the kernel of the state S_i . However, conceptually, the set $S_i^* \cup L_{int} \cup L_{timeless}$, which is the kernel extended by the intensional and time-independent predicates, can be thought of as the time-stamped *extended state* at time t_i . Similarly, the sets evi of events are not represented explicitly, but are represented implicitly by the set of simple events that occur from time t_{i-1} to t_i .

3.2 Internal and external syntax

The explicit representation of time, defined by L_{temp} , underpins the model-theoretic semantics of the language. However, at the potential expense of restricting the expressive power of the language, the language can also be expressed in an external syntax in which time is implicit. For example, in [Kowalski and Sadri 2011], temporal ordering is indicated by the order in which formulas are written and by the use of special logical connectives. This is similar to the syntax of TR Logic [Bonner and Kifer 1993], in which $P \otimes Q$ means “do P and then do Q ”. In LPS, depending on the context, this translates into an internal syntax with an explicit representation of time, for example as $P(T_1, T_2) \wedge Q(T_2, T_3)$ or as $P(T_1, T_2) \wedge Q(T_3, T_4) \wedge T_2 < T_3$, if P and Q are events.

The internal syntax is also compatible with a modal external syntax. For example, $P \wedge \diamond Q$ can similarly be translated into $P(T_1, T_2) \wedge Q(T_3, T_4) \wedge T_2 < T_3$, if P and Q are events, and $P \wedge \circ Q$ can be translated into $P(T_1, T_2) \wedge Q(T_2, T_3)$. However in modal logic, events and actions are not represented by formulas, but by parameters of modal operators, and P and Q would be restricted to fluents. If P and Q are fluents, then $P \wedge \diamond Q$ can be translated into $P(T_1) \wedge Q(T_2) \wedge T_1 < T_2$, and $P \wedge \circ Q$ can be translated into $P(T_1) \wedge Q(T_2) \wedge succ(T_1, T_2)$.

Graphical notations for representing partial ordering are also possible. For example, a notation such as:



could be used to represent $P(T_1) \wedge Q(T_2) \wedge R(T_3) \wedge T_1 + 2 < T_3 \wedge T_2 < T_3 \leq T_2 + 6$.

In the remainder of the paper, we use the internal syntax with explicit time, because it clarifies the model-theoretic semantics, and because it is neutral with respect to external syntax.

Note that the internal syntax of LPS employs a relational syntax in which function symbols are used only for constructing names of composite objects. It would also be possible to employ an external syntax in which function symbols are used with equality for representing functional relations, as in $colour(a) = red$ or $colour(a) = colour(b)$. In the internal syntax, these would be translated into $colour(a, red)$ and $colour(a, X) \wedge colour(b, X)$ respectively, together with integrity constraints to represent the fact that objects have only one colour. In practice, such integrity constraints might be emergent properties that would not have to be checked explicitly.

Notice that functional external syntax could be combined with infix notation. For example, $T_3 \leq T_2 + 6$ could be regarded as shorthand for $T_3 \leq T \wedge +(T_2, 6, T)$.

3.3 Reactive Rules

Reactive rules (or simply *rules*) in \mathbf{R} are sentences of the logical form:

$$\forall X [antecedent(X) \rightarrow \exists Y consequent(X, Y)]$$

where X is the set (or tuple) of all unbound variables, including time variables, that occur in $antecedent(X)$, and Y is the set (or tuple) of all unbound variables, including time variables, that occur only in $consequent(X, Y)$. In addition to the variables in X and Y , the rule can contain other variables that are bound in FOL conditions. For notational convenience, we write $consequent(X, Y)$ even though $consequent(X, Y)$ need not contain all the variables that occur in $antecedent(X)$. Because of these restrictions on the quantification of variables, we can omit the quantifiers $\forall X$ and $\exists Y$. More formally:

Definition 3.1 (Reactive rule). A reactive rule is a sentence of the form:

$$antecedent(X) \rightarrow consequent(X, Y)$$

where both $antecedent(X)$ and $consequent(X, Y)$ are a conjunction¹ of FOL state conditions, event atoms and temporal constraints.

- An *FOL state condition* (or simply a *state condition*) is an FOL formula in the vocabulary of the fluent and time-independent predicates, containing at most a single time variable, which is unbound.

Operationally, the evaluation of a state condition can be understood as a query to the current extended state $S_i^* \cup L_{int} \cup L_{timeless}$, where the time parameter refers to the current time t_i .

- An *event atom* is an atomic formula whose predicate symbol is a simple or composite event. Similarly an *action atom* is an event atom whose predicate symbol is an action.

¹ In [Kowalski and Sadri 2012b] we allowed consequents of rules to be disjunctions of such conjunctions. This is because we focused on programs consisting of reactive rules without logic programs. In this paper, we obtain a similar effect by allowing consequents to contain predicates defined by non-deterministic logic programs. Alternatively, it would be easy to extend the language of this paper to include the disjunctive consequents of the earlier paper.

- A temporal constraint is an atomic formula of the form $\text{succ}(\text{time}_1, \text{time}_2)$, $\text{time}_1 < \text{time}_2$ or $\text{time}_1 \leq \text{time}_2$ where time_1 and time_2 represent time points, one of which is a variable, and the other of which is a variable or a constant.

The temporal constraints contain only variables that occur in the state conditions and event atoms of the rule, and all the time parameters that occur in the antecedent are constrained directly or indirectly in the consequent to be earlier than or equal to the time parameters that occur only in the consequent.

Notice that, although fluents can occur as subformulas of FOL conditions, events can occur only as atomic conjuncts. This makes it impossible to represent, for example, the condition that no event of a certain kind occurs within a certain interval of time. This restriction simplifies the operational semantics, so that any non-temporal condition in a reactive rule can be treated as a query to the extended current state $S_i^* \cup L_{\text{int}} \cup L_{\text{timeless}}$ augmented with ev_i^* . The restriction is not necessary for the model-theoretic semantics, and can be removed at the expense of complicating the operational semantics.

3.4 Goal clauses

In both the model-theoretic and operational semantics, whenever the antecedent of a reactive rule becomes true, the consequent of the rule becomes a goal to be made true in the future.

In addition to satisfying such derived goals, it may also be required to make an initial set of goals true in the future. For this purpose, and because the operational semantics maintains a goal state containing goal clauses, we define goal clauses here more generally:

Definition 3.2 (Goal clause). A goal clause is an existentially quantified conjunction of FOL state conditions, event atoms and temporal constraints. All the variables in the temporal constraints occur in the state conditions and event atoms of the goal clause.

Note that a goal clause can also be regarded as a reactive rule with an empty (or true) antecedent.

3.5 Logic programs

The logic programs $L = L_{\text{int}} \cup L_{\text{events}} \cup L_{\text{timeless}} \cup L_{\text{temp}}$ play a supporting role to the reactive rules of an LPS program $\langle R, L, D \rangle$. Like the antecedents and consequents of rules, they can also contain non-atomic FOL conditions, as in the extended logic programs of [Lloyd and Topor 1984].

Definition 3.3 (Extended logic program). An extended logic program is a set P of sentences (or *clauses*) of the form:

$$\text{head}(X) \leftarrow \text{body}(X, Y)$$

where X is the set of all variables that occur in $\text{head}(X)$, and Y is set of all unbound variables that occur only in $\text{body}(X, Y)$. The *head* of the clause $\text{head}(X)$ is an atomic formula and the *body* of the clause $\text{body}(X, Y)$ is a (possibly empty) conjunction of

conditions, which are atomic and non-atomic FOL formulas.² An extended logic program whose body is a (possibly empty) conjunction of atomic formulas is a *Horn clause program*.

Clauses are implicitly quantified in the form:

$$\forall X [head(X) \leftarrow \exists Y body(X, Y)]$$

which is more often written in the logically equivalent form:

$$\forall X \forall Y [head(X) \leftarrow body(X, Y)]$$

These quantifiers are normally left implicit, because they can always be reconstructed unambiguously. Also, we drop the qualification “extended” and call extended logic programs simply “logic programs”.

Lloyd and Topor reduce logic programs with FOL conditions in their bodies to normal logic programs whose bodies are *literals*, namely conjunctions of atomic formulas and negations of atomic formulas. In contrast, we evaluate FOL conditions using the standard definition of truth for formulas of first-order logic. However, for this purpose, we need to ensure that the predicates of non-atomic FOL conditions are fully defined before the conditions are evaluated. For this purpose, we employ a simple generalisation of local stratification, called *FOL-stratification*. Later we will generalize this to weak FOL-stratification.

FOL-stratification is exemplified by the clause that defines the subset relation \subseteq in terms of the membership relation \in :

$$U \subseteq V \leftarrow \forall Z [Z \in U \rightarrow Z \in V]$$

Given a set of atomic sentences *Memb* defining the extension of the predicate \in , the FOL condition in the body of the clause can be viewed as a query to the extensional database *Memb*. The results of the query are used to define the predicate \subseteq in the head of the clause.

The relationship between the two predicates \subseteq and \in in the clause can be viewed in terms of stratification: The predicate \in is defined in a lower stratum, and the predicate \subseteq is defined in a higher stratum.

Loosely speaking, a logic program is FOL-stratified if there is a well-ordering of the ground atoms of the language into distinct strata, such that, for every ground instance *C* of a clause in the program, the non-atomic FOL conditions in the body of *C* are defined in lower strata than the stratum of the head of *C*, and atomic conditions in the body are defined in the same or lower strata than the head.³ In LPS, the component programs *L_{int}*, *L_{events}*, *L_{timeless}* and *L_{temp}* are all FOL-stratified in this sense.

For example, the logic program *L_{int}* can be FOL-stratified by assigning extensional and time independent atoms to a lower stratum, and intensional predicates to a

² Although a conjunction of FOL formulas is itself an FOL formula, we distinguish between atomic and non-atomic FOL formulas, because non-atomic formulas generalise negative literals in normal logic programming.

³ Replacing an FOL condition in an FOL-stratified program by one that is logically equivalent does not affect the FOL perfect model, if the resulting program is also FOL-stratified. However, it may change a program that is FOL-stratified into one that is not. For example, the program $p \leftarrow p$ is stratified, but the program $p \leftarrow \neg p$ is not.

higher stratum. As a consequence, FOL conditions in the bodies of clauses can be evaluated without using the intensional predicates. However, the stratification can be extended to multiple levels. For example, if the predicate *clear* is intensional, then the predicate *shallow* can be defined as an intensional predicate at a higher stratum than the stratum of *clear*.

$$\textit{shallow}(\textit{Place}, T) \leftarrow \forall \textit{Block} [\textit{on}(\textit{Block}, \textit{Place}, T) \rightarrow \textit{clear}(\textit{Block}, T)]$$

Similarly, the logic program $L_{\textit{events}}$ can be FOL-stratified by assigning fluent, simple event and time-independent predicates to a lower-stratum than composite event predicates. FOL conditions involving the lower-stratum predicates can be viewed as queries to the extended current state $S_i^* \cup L_{\textit{int}} \cup L_{\textit{timeless}}$ augmented with ev_i^* .

We will define FOL-stratification and its semantics more formally later.

Definition 3.4 (The logic programs L). $L_{\textit{timeless}}$ is an FOL-stratified logic program containing only predicates without time parameters.

$L_{\textit{temp}}$ is an FOL-stratified logic program defining the temporal predicates *succ*, \leq and $<$, as a discrete total ordering.

$L_{\textit{int}}$ is an FOL-stratified logic program, consisting of clauses of the form $\textit{head}(X, T) \leftarrow \textit{body}(X, Y, T)$ in which the predicate in the *head* is an intensional predicate, and the predicates in the *body* are intensional, extensional or time-independent predicates. The intensional predicates are assigned to higher strata than the extensional and time-independent predicates. Each clause in $L_{\textit{int}}$ contains exactly one time parameter T that is a variable.

$L_{\textit{events}}$ is an FOL-stratified logic program, consisting of clauses of the form $\textit{head}(X, T_1, T_2) \leftarrow \textit{body}(X, Y, T_1, T_2)$ in which the predicate in the *head* is a composite event predicate, and the predicates in the *body* are composite event, simple event, fluent, time-independent or temporal predicates. The composite event predicates are all assigned to a higher stratum than the simple event, fluent and time-independent predicates. T_1 and T_2 represent the interval over which the composite event takes place, and are constrained to be, respectively, the earliest and latest time variables occurring in a fluent or event atom in $\textit{body}(X, Y, T_1, T_2)$ ⁴.

The time variables in temporal constraints must all occur in the *head* or unbound in fluent or event atoms in the *body*.

Note that the body of a clause in $L_{\textit{events}}$ is equivalent in form both to a goal clause, and to an antecedent or consequent of a reactive rule.

3.6 The domain theory D

The domain theory D of an LPS program $\langle R, L, D \rangle$ has two components $D = D_{\textit{post}} \cup D_{\textit{pre}}$. The first component $D_{\textit{post}}$ is an FOL-stratified logic program that specifies the extensional fluents that are initiated and terminated by simple events. The second component $D_{\textit{pre}}$ is a set of integrity constraints restricting the occurrence and co-occurrence of simple events.

⁴ Note that X might include other time parameters, as in the emergency example of section 2.

Definition 3.5 (Domain theory D). D_{post} is a set of clauses of the form:

$$head(T_2) \leftarrow body(T_1, T_2)$$

and D_{pre} is a set of integrity constraints of the form:

$$false \leftarrow body(T_1, T_2)$$

$head(T_2)$ is an atom of the form $initiated(P, T_2)$ or $terminated(P, T_2)$, where P is an extensional fluent. $body(T_1, T_2)$ is an FOL formula containing only simple event predicates with time parameters T_1 and T_2 , fluent predicates with time parameter T_1 , and time-independent predicates.

In the operational semantics, $body(T_1, T_2)$ is a query to the augmented current state $S_i^* \cup L_{int} \cup L_{timeless} \cup ev_i^*$ at time t_i . An answer to the query is a ground instantiation of the free variables in $body(T_1, T_2)$, leaving bound variables to be treated according to the classical semantics of universal and existential quantifiers.

The FOL-stratification of D_{post} consists of two strata: The predicates defined by $S_i^* \cup L_{int} \cup L_{timeless} \cup ev_i^*$ constitute the lower stratum, and the meta-predicates $initiated(P, T_2)$ and $terminated(P, T_2)$, constitute the higher stratum

3.7 The environment

An LPS framework $\langle R, L, D \rangle$ represents the goals R and beliefs L of an individual agent embedded in an environment, which consists of a current state S_i and current set ev_i of events. For simplicity, the current state includes both the agent's own local state and the whole of the external, global state. Similarly, the current events include both the agent's own local actions and all external, global events.

In a multi-agent system, the global components are shared among all the agents, but the local components are encapsulated. In a closed system, consisting of a single agent with no external environment, the entire state is internal, and all events are internal actions.

D_{post} uses the events ev_i to update the state S_i , and D_{pre} ensures that the set ev_i of events is possible. D_{post} updates both the local and global components of the state, using both local and global events. In the case of purely internal actions, these updates are performed entirely by the agent itself. In the case of concurrent events whose effects depend on both internal actions and external events, the updates are performed by the external environment.

The computational task for LPS is defined in terms of making the goals $R \cup G_0$ of an individual agent *true* in a model that is determined by the combined local and global components of the environment. This ensures that, in a multi-agent setting, all the agents have the same consistent (and co-ordinated) view of the shared components of the environment. However, it does not mean that all the agents have unrestricted access to all the information in the environment. An agent's access to the environment is restricted inherently by its vocabulary. Moreover, the external environment might also impose further restrictions of its own.

In a more refined formalization of LPS, it might be desirable to decompose the environment into separate local and global components. However, to simplify the treatment in this paper, we combine the local and global components of the environment into a single entity.

4. THE MODEL-THEORETIC SEMANTICS OF LPS

In this section, we present two alternative semantics: The first involves an event theory ET that uses a frame axiom to express that any fluent that is not terminated by a state transition persists from one state to the next. The second uses destructive state updates. In section 8, we show that the two semantics generate the same intended models.

4.1 The event theory ET

The event theory ET is a logic program that, given $D_{post} \cup L \cup S_0^* \cup ev^*$, defines when an extensional fluent P holds at a time point $T > 0$:

Definition 4.1 (Event theory ET). The event theory ET consists of the two clauses:

$$\begin{aligned} holds(P, T_2) &\leftarrow initiated(P, T_1, T_2) \\ holds(P, T_2) &\leftarrow holds(P, T_1) \wedge succ(T_1, T_2) \wedge \neg terminated(P, T_1, T_2) \end{aligned}$$

ET is a hybrid of the situation calculus and the event calculus. The second clause is a frame axiom in the spirit of the situation calculus. However, because states are not represented explicitly in LPS, the ontology of ET is that of the event calculus.

With the aid of ET , we can give a simple characterization of the computational task for LPS:

Definition 4.2 (Computational task according to ET). Given an LPS program $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$, an initial state S_0 and initial set of goal clauses G_0 , the computational task is to generate, for every set ext_i of external events, where $i \geq 1$, a set $acts_{i+1}$ of actions, such that:

$$ET \cup D_{post} \cup L \cup S_0^* \cup ev^* \text{ entails } \mathbf{R} \cup G_0 \cup D_{pre}$$

where

$$\begin{aligned} ev^* &= ev_1^* \cup ev_2^* \cup \dots \cup ev_i^* \cup \dots \\ ev_i &= ext_i \cup acts_i, \text{ for } i \geq 1, \text{ and } act_1 = \{\}. \end{aligned}$$

The notion of entailment here is deliberately unspecified, and many different notions of entailment have been proposed for similar event theories, mainly to give them a non-monotonic semantics. The semantics that we will define later is also non-monotonic, and is expressed in terms of the truth of $\mathbf{R} \cup G_0 \cup D_{pre}$ in a uniquely determined, intended model of $ET \cup D_{post} \cup L \cup S_0^* \cup ev^*$.

But, independent of the definition of entailment, reasoning with ET is computationally infeasible. It is not practical either to reason forwards with frame axioms, duplicating facts that hold from one state to the next, or to reason backwards, to determine whether a fact holds in a given state by determining whether it held in previous states. As a consequence, frame axioms are rarely used in practical applications, and destructive assignment or destructive updates are generally used instead.

The computational infeasibility of reasoning with the frame axiom(s) has received hardly any attention. For example, [Shanahan 1987], in *Solving the Frame Problem*, explicitly excludes consideration of “implementation issues” on page 7. In this paper,

we consider computational feasibility to be one of the core issues related not only to implementation, but also to the semantics of state transition systems.

In contrast with the use of the frame axiom to reason about change of state in *ET*, computation in LPS is performed by using destructive change of state.

To avoid repetition, we use the following notation in definition 4.3 and elsewhere for the sequences of time stamped events and states:

$$\begin{aligned} ev^* &= ev_1^* \cup ev_2^* \cup \dots \cup ev_i^* \cup \dots \\ ev_i &= ext_i \cup acts_i, \text{ for } i \geq 1, \text{ where } act_1 = \{\} \\ S^* &= S_0^* \cup S_1^* \cup \dots \cup S_i^* \cup \dots \end{aligned}$$

Definition 4.3 (Informal specification of the computational task). Given an LPS program $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$, an initial state S_0 and initial set of goal clauses G_0 , the *computational task* is to generate, for every set ext_i of external events where $i \geq 1$, a set $acts_{i+1}$ of actions such that $\mathbf{R} \cup G_0 \cup D_{pre}$ is true in the intended model of:

$$\mathbf{L} \cup S^* \cup ev^*$$

where for $i \geq 1$, S_i is obtained from S_{i-1} by deleting all the fluents in S_{i-1} terminated by ev_i and adding all the fluents initiated by ev_i as determined by \mathbf{D} .

In this specification, the computational task is shared between an agent attempting to execute a collection of *candidate-acts_i* to make $\mathbf{R} \cup G_0$ true and the environment, maintaining D_{pre} , by arbitrating between the agent's *candidate-acts_i* and other candidate actions of other agents. The result of this arbitration is a set of events, $ev_i = ext_i \cup acts_i$, selected by the environment, where $acts_i$ is the subset of *candidate-acts_i* that have succeeded, and ext_i is the set of all other successful events.

This specification is incomplete. It needs to be augmented with a definition of the intended model, and with a more precise statement of the definition of S_i in terms of S_{i-1} . We present the augmented specification in the next subsection.

4.2 An abstract specification of the computational task

In general, a logic program P can be viewed as an intensional definition of the predicates that occur in the heads of clauses in P in terms of the predicates that occur in the bodies of clauses in P . These head predicates can also be represented extensionally as a set $sem(P)$ of ground atoms. The set $sem(P)$ has a dual interpretation: syntactically as a set of sentences, and semantically as a model-theoretic structure. Viewed in semantic terms, $sem(P)$ is a *Herbrand interpretation*, which represents the set of all the ground atoms that are true in the interpretation.

In the simplest case, when P is a set of Horn clauses, there exists an extensional representation $sem(P) = min(P)$ of P that is also minimal (with respect to set inclusion) [van Emden and Kowalski 1976]. This case does not cater for logic programs that contain non-atomic FOL conditions. Later we define two semantics that do cater for such FOL conditions. Both semantics are defined for the case in which non-atomic FOL conditions in the body of a clause are defined in lower strata than the stratum of the head of the clause. The first semantics generalizes local stratification, and the second semantics generalizes weak stratification.

The following definition expands the specification of the computational task in 4.3, and presents it in an abstract form that is independent of the semantics sem .

The only assumption is that *sem* associates a unique Herbrand interpretation with every logic program in the class of programs under consideration.⁵

Definition 4.4 (Abstract specification of the computational task). Let *sem* be a mapping from logic programs *P* to sets of ground atoms that are instances of the heads of clauses in *P*. Given an LPS program $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$, an initial state S_0 and initial set of goal clauses G_0 , the *computational task* is to generate, for every set ext_i of external events, where $i \geq 1$, a set $acts_{i+1}$ of actions, such that $\mathbf{R} \cup G_0 \cup D_{pre}$ is true in the Herbrand interpretation:

$$sem(\mathbf{L} \cup S^* \cup ev^*), \text{ where for } i \geq 1:$$

$$S_i = (S_{i-1} - \{p \mid terminated(p, t_{i-1}, t_i) \in sem(D_{post} \cup L_{int} \cup L_{timeless} \cup S_{i-1}^* \cup ev_i^*)\}) \\ \cup \{p \mid initiated(p, t_{i-1}, t_i) \in sem(D_{post} \cup L_{int} \cup L_{timeless} \cup S_{i-1}^* \cup ev_i^*)\}.$$

In this definition, the clauses in D_{post} can be viewed as querying the augmented current state $L_{int} \cup L_{timeless} \cup S_{i-1}^* \cup ev_i^*$ to determine the fluents that have been initiated and the fluents that have been terminated by the set ev_i of events that take place in the transition from S_{i-1} to S_i .

The definition appeals to the notion of truth in a Herbrand interpretation. This notion requires, in turn, the concept of the Herbrand universe:

Definition 4.5 (Herbrand universe and Herbrand base). Given the vocabulary of a sorted language, the *Herbrand universe* is the set of all well-sorted ground terms that can be constructed from the vocabulary. The *Herbrand base* is the set of all well-sorted ground atoms that can be constructed from the vocabulary.

Definition 4.6 (Herbrand interpretation and Herbrand model). A *Herbrand interpretation* is a subset of the Herbrand base. A *Herbrand model* M of a set S of sentences is a Herbrand interpretation such that every sentence s in S is true in M .

The truth value of a sentence s in a Herbrand interpretation M depends not only upon M , but also upon the Herbrand universe U :

Definition 4.7 (Truth). If $s \in H$ is an atomic sentence, then s is true in M if and only if $s \in M$, and $\neg s$ is true in M if and only if $s \notin H$.

For any formula $s(X)$ with free variables X , $\forall X s(X)$ is true in M if and only if $s(x)$ is true in M for every $x \in U$, and $\exists X s(X)$ is true in M if and only if $s(x)$ is true in M for some $x \in U$.

The truth values of all other FOL sentences of the language are defined as usual in classical FOL. In particular, the negation $\neg s$ of a sentence is true in M if and only if s is not true in M . Thus negation is classical negation, but also has the flavour of negation as failure, because $\neg s$ is true in M if and only if s fails to be true in M .

It is possible to extend the definition of truth to include sentences with aggregation operators, which construct such objects as the set of all terms that satisfy a given formula, the number of such terms, or their sum. This extension is necessary for defining the postconditions of concurrent events whose effects are

⁵ This includes the stable model semantics [Gelfond and Lifschitz 1988], if we take *sem* to be the intersection of all stable models, and if we allow the mapping to be partial in case there are no models.

cumulative (like pushing a block in different directions). We do not explore this possibility further in this paper.

4.3 The simplified case in which L and D_{post} are sets of Horn clauses

The abstract semantics of definition 4.4 specializes naturally to the simplified case in which L and D_{post} are Horn clause programs. In this case, the semantic mapping sem is given by the minimal model. In general, every set of Horn clauses has a minimal Herbrand model:

Definition 4.8 (Minimal model). Given a set of Horn clauses P with Herbrand base H , the minimal model $min(P)$ of P is the smallest set $M \subseteq H$ such that, for every ground instance $head \leftarrow body$ of a clause in P , $head \in M$ if $body$ is true in M .

Notice that with this definition, the minimal model of P is equivalent to the minimal model of $ground(P)$, the set of all ground instances of P .

Since the body of a Horn clause is a conjunction of atomic formulas, the condition that $body$ is true in M is equivalent to the condition that $atom \in M$ for every atomic condition $atom$ in $body$. As we will see in section 6, the formulation in terms of the truth of $body$ in M has the advantage that it also applies to FOL-stratified clauses whose bodies contain FOL conditions belonging to strata that are lower than the stratum of the head of the clause.

Here we instantiate the semantics of LPS for the simplified case where L and D_{post} are Horn clause programs:

Definition 4.9 (Computational task for Horn clauses). Given an LPS program $\langle R, L, D \rangle$, in which L and D_{post} are sets of Horn clauses, the computational task is as given in definition 4.4 with $sem = min$.

Later in the paper, we instantiate the abstract semantics for more general logic programs containing non-atomic FOL conditions. In the meanwhile, we use the more abstract semantics sem , to evaluate conditions in the operational semantics.

5. THE OPERATIONAL SEMANTICS

The operational semantics (OS) can be thought of as a potentially non-terminating cycle, in which external events and internal actions are merged, the state is destructively updated, and the agent thinks and decides what to do next. Thinking can be interrupted to observe changes in the environment, and to perform actions.

The cycle is relatively abstract, and is compatible with many different implementations. In particular, although the OS is defined for programs written with an explicit representation of time, it can also be implemented, as in [Kowalski and Sadri 2011], directly for programs written in an external syntax in which temporal order is indicated by the order in which conditions and events are written.

The cycle is also only semi-constructive. Extended states can contain a countably infinite number of ground atoms, and an FOL query can have a countably infinite number of answers. In practice, these infinities can be avoided, for example by eliminating function symbols, as in Datalog. None the less, it simplifies the treatment if we do not impose any theoretically unnecessary restrictions.

We assume that the i -th cycle coincides with the i -th state S_i , and that states are instantaneous, holding at the time point t_i . The set of events ev_i takes place from time t_{i-1} to time t_i , transforming the state S_{i-1} into the state S_i . This is equivalent to assuming that the state S_{i-1} holds fixed between times t_{i-1} and t_i , and that the events ev_i take place instantaneously at time t_i .

As we will see, actions are selected for possible execution at the end of the cycle, but are combined with external events at the beginning of the next cycle. In the case of conflict between the selected actions and external events, the environment determines which sets of concurrent events actually occur.

5.1 Goal States

In addition to maintaining the current state S_i , the OS maintains a *goal state* G_i , which is a set (or conjunction) of goal trees. Every node in a goal tree is a goal clause representing an alternative way of solving the top-level goal clause at the root of the tree. This top-level goal clause is either an initial goal clause, or an instance of the consequent of a reactive rule introduced when the antecedent of the rule becomes true. To solve the computational task, all the goal trees must eventually be reduced to *true*.

Definition 5.1 (Goal state). A *goal state* is a set (or conjunction) of goal trees.

A *goal tree for a goal clause* C_0 is a set (or disjunction) of goal clauses organized as nodes in a tree. The root of the tree is the goal clause C_0 . Every child node C_i is obtained from its parent node C_{i-1} by goal-reduction in steps 2.1 and 2.2 of cycle.

A *branch* of a goal tree is a sequence of nodes $C_0, C_1, \dots, C_n, n \geq 0$, starting with the root node C_0 , and such that every node C_i is a child of the previous node C_{i-1} .

The top-level goal clause C_0 of a goal tree is *reduced to true* if and only if there is a branch C_0, C_1, \dots, C_n of the tree with $C_n = true$. In this case we also say that the goal tree is reduced to *true*.

The top-level goal clause C_0 of a goal tree is *reduced to false* if and only if every branch C_0, C_1, \dots, C_n of the tree contains a goal clause $C_n = false$. In this case we also say that the goal tree is reduced to *false*.

Logically, a goal state is the (possibly infinite) conjunction of its goal trees, and a goal tree is the disjunction of the (finitely many) goal clauses that are its nodes. An empty goal state is logically equivalent to *true*, and a goal tree that is reduced to *false* is logically equivalent to *false*. *Operationally*, each goal tree is a separate *thread*, independent of other goal trees.

To simplify the OS, we will assume that composite events in the antecedents of reactive rules have been pre-processed, by performing backward reasoning (with *Levents*) in advance, reducing composite events to conjunctions of simple events, FOL conditions and temporal constraints. In the general case, this could give rise to an infinite set of reactive rules.

Although a practical implementation can work only with finite sets, in theory the OS can handle such infinite sets. At the expense of complicating the OS, composite event definitions could also be executed in the forward direction, as in many of the integrity checking methods developed for deductive databases. Alternatively, backward reasoning could be used at “run time” to reduce composite event predicates to simpler event predicates. For simplicity, we ignore these (and other) possibilities in this paper.

In addition to maintaining a goal state, the OS maintains a current set of reactive rules R_i . A new rule is added to R_i when a conjunct in the *antecedent* of a rule becomes true. The new rule represents an instance of the rest of the original rule that needs to be true in the future.

5.2 The OS Cycle

Given an LPS program $\langle R, L, D \rangle$, the i -th iteration of the OS cycle uses the set ev_i of combined external events ext_i and actions act_i to transform the state S_{i-1} , rules R_{i-1} and goal state G_{i-1} at time t_{i-1} into S_i , R_i and G_i at time t_i , generating a possibly empty set *candidate-acts* $_{i+1}$ of candidate actions to be merged with external events and executed at the beginning of the next cycle. Initially $i = 1$, $R_0 = R$, $act_1 = \{\}$, and G_0 , if it is not empty, consists of a one-node tree rooted at an initial goal clause C_0 .

To be faithful to the model-theoretic semantics, it is not possible to restrict the amount of time that can be spent on step 0 of the cycle, which updates the state, and on step 1, which processes the antecedents of reactive rules. In a practical system, it would be necessary to ensure that these steps can be performed in a timely manner, before the next time in the succession of time points.

Assuming that this assurance can be given for steps 0 and 1, it is also necessary, to restrict the amount of time spent on goal reduction in step for the same reasons, 2. This can be done in a number of different ways. If the time of the next iteration of the cycle is known in advance, then the number of goal-reduction steps can simply be restricted so that the time is not exceeded. Alternatively, the number of goal reduction steps can be limited by a maximum amount Max , and this amount could be decreased every time goal reduction is performed. We have implemented this latter approach in our Prolog prototype, and it is the one we assume here.

With these assumptions, the i -th iteration of the cycle consists of the following steps:

Step 0. Update the current state. The environment arbitrates among the various candidate actions submitted by different agents, together with any other externally generated events, returning a set $ev_i = ext_i \cup acts_i$ that includes both external events ext_i and a subset $acts_i$ of the submitted candidate actions *candidate-acts* $_i$, such that D_{pre} is true in $sem(S_{i-1}^* \cup L_{int} \cup L_{timeless} \cup ev_i^*)$.

State S_{i-1} is transformed into S_i , by deleting any fluents p such that $terminated(p, t_{i-1}, t_i) \in sem(D_{post} \cup L_{int} \cup L_{timeless} \cup S_{i-1}^* \cup ev_i^*)$ and adding any fluents p such that $initiated(p, t_{i-1}, t_i) \in sem(D_{post} \cup L_{int} \cup L_{timeless} \cup S_{i-1}^* \cup ev_i^*)$.

Let $G_i = G_{i-1}$, $R_i = R_{i-1}$ and *candidate-acts* $_{i+1} = \{\}$.

Step 1. Process antecedents of rules. For every reactive rule in R_i , construct every parsing of the rule into the form:

$$early\text{-}antecedents \wedge other\text{-}antecedents \rightarrow consequent$$

where *early-antecedents* is a conjunction of state conditions and simple events such that all the time parameters in *early-antecedents* can be unified with the current time t_i , without making any temporal constraints in *other-antecedents* false, and without constraining any of the time parameters in state conditions or events in *other-antecedents* to be equal to or earlier than t_i .

For each such parsing, and each ground instance *early-antecedents* σ that is true in $sem(L_{int} \cup L_{timeless} \cup S_i^* \cup ev_i^*)$, generate the corresponding “resolvent”:

other-antecedents $\sigma \rightarrow$ consequent σ

simplify the temporal constraints in the resolvent, and add the simplified resolvent as a new reactive rule to R_i .

For simplification, it is sufficient to delete any temporal constraints that are true in $sem(L_{temp} \cup L_{timeless})$. If after simplification, *other-antecedents* σ is an empty conjunction (equivalent to *true*), then the simplified resolvent is deleted from R_i and added to G_i as a new top-level goal, starting a new goal tree (or thread).

Step 2. Process goal clauses. If the number of steps that can be performed has reached *Max*, or if there are no new steps that can be performed in this iteration of the cycle, then this iteration of the cycle terminates.

Otherwise, while the number of goal-reduction steps performed so far has not reached *Max*, and there are new steps that can be performed in this iteration of the cycle, choose any goal clause C in G_i and perform one of the steps 2.1, 2.2 or 2.3.

Step 2.1. Reduce a composite event. Select a composite event atom in C , unify the composite event atom with the head of some clause in L_{events} and update G_i by adding the resolvent to G_i as a child of C . Note that there are no restrictions on the time parameters in this step. This allows the goal-reduction of composite events to look-ahead into the future, which is a kind of forward planning.

Step 2.2. Reduce a conjunction of state conditions and simple events. Select a parsing of C of the form:

early-consequents \wedge other-consequents

where *early-consequents* is a conjunction of state conditions and simple events such that all the time parameters in *early-consequents* can be unified with the current time t_i , without making any temporal constraints in *other-consequents* false, and without constraining any of the time parameters in state conditions or events in *other-consequents* to be equal to or earlier than t_i .

If there is a ground instance *early-consequents* σ that is *true* in $sem(L_{int} \cup L_{timeless} \cup S_i^* \cup ev_i^*)$, then choose one such instance, generate the “resolvent” *other-consequents* σ , simplify the temporal constraints, and update G_i by adding the simplified resolvent to G_i as a child of C .

If after simplification, the resolvent is an empty conjunction (equivalent to *true*), then the entire goal tree containing the goal clause can be deleted, because the top-level goal clause in the tree is then also *true*.

Step 2.3. Choose a conjunction of simple actions for attempted execution. Select a parsing of C of the form:

actions \wedge other-consequents

where *actions* is a conjunction of simple actions $happens(a, T_1, T_2)$ such that all the time parameters T_1, T_2 can be unified with the times t_i and t_{i+1} respectively, without making any temporal constraints in *other-consequents* false, and without constraining any of the time parameters in state conditions or events in *other-consequents* to be equal to or earlier than t_i .

Add all of the simple actions $happens(a, T_1, T_2)$ in *actions* to *candidate-acts_{i+1}*. Using the successful execution of these actions to resolve these and other *action* subgoals takes place in step 2.2 of the next iteration of the cycle.

Notes:

1. Steps 1 and 2 of the OS are the operational semantics of a single agent, possibly interacting with other agents, both by observing changes in the environment and by performing actions. In the multi-agent case, step 0 is global to all the agents, and as a simplifying assumption the different agent cycles are all synchronized, so that all of the agents try to perform their actions at the same time. Step 0 non-deterministically selects a single possible set of concurrent events and updates the current state for all the agents. In this way, the global state serves as a coordination medium, in the manner of the Linda programming paradigm [Carriero and Gelerter 1989] and the blackboard model [Hayes-Roth 1985].

In the special case of a single agent maintaining only an internal state without any interaction with the external environment, the internal state serves the same function as the global state. In such a case, the agent needs to take responsibility itself for ensuring that the collection of selected candidate actions $candidate-acts_{i+1} = acts_{i+1} = ev_{i+1}$ is possible as specified by D_{pre} .

2. Step 2.3 allows the possibility that the selected *actions* may contain variables other than time variables. This could be useful in the case of external actions where the variables can give feedback about the result of the action. For example, the variable X in the action $move-forward(X, T)$ might be instantiated by the environment indicating how far the action succeeded.

Alternatively, and in the case of internal actions, we can insist that only ground simple actions are selected for attempted execution. In the case of external actions, feedback from the environment can be given instead by means of the change of state resulting from the selected actions and other external events.

3. In steps 1, 2.2 and 2.3, different parsings amount to different ways of sequencing state conditions and simple events in the same conjunction. For example, the conjunction $p(T_1) \wedge q(T_2) \wedge r(T_3) \wedge T_1 \leq T_3 \wedge T_2 \leq T_3$ has the four correct parsings:

$p(T_1) \wedge q(T_2) \wedge r(T_3)$ at the same time
 $p(T_1) \wedge q(T_2)$ at the same time and before $r(T_3)$
 $p(T_1)$ before $q(T_2) \wedge r(T_3)$
 $q(T_2)$ before $p(T_1) \wedge r(T_3)$

The three parsings in which $r(T_3)$ is selected before $p(T_1)$ or $q(T_2)$ are incorrect and not allowed. Moreover, they are useless, because selecting $r(T_3)$ before $p(T_1)$ or $q(T_2)$ makes it impossible to evaluate $p(T_1)$ or $q(T_2)$ in the future.

4. If a goal clause becomes *false*, then there is no point in trying to solve other subgoals in the same goal clause. If an entire goal tree is reduced to *false*, then the top-level goal clause in the tree is *false*, the instance of the reactive rule that generated it is *false*, and the reactive rule itself is also *false*. In theory, the OS should terminate in failure. However, in practice, we may want to allow the OS to continue, trying to make all instances of the rules *true* in the future. Moreover, we

also have the option of providing a fail-safe, alternative way of solving any goal that is vulnerable to failure.

5. In the various repetitions of step 2 within a given iteration of a cycle, the OS can select any goal clause C in G_i . It can jump from one goal tree to another, attempting to solve different top-level goal clauses concurrently. Or it can focus on one goal tree at a time. Within a given goal tree, it can jump from one branch to another, trying alternative ways of solving the same top-level goal clause concurrently. Or it could focus on one way of solving a top-level goal clause, extending one branch of the goal tree at a time.

6. In different iterations of a cycle, in step 2, the OS can re-select the same goal clause C . In step 2.1, however, it may do so only to try to unify the selected composite event atom in C with the head of a clause in L_{events} not tried in previous iterations of the cycle. In step 2.2 it can re-try the same parsing *early-consequents* of conditions and simple event atoms, because the relevant part of the augmented current state $L_{int} \cup L_{timeless} \cup S_i^* \cup ev_i^*$ may have changed. For similar reasons, in step 2.3 it can re-try the same conjunction of actions, because actions that were not possible before may become possible in the new current state.

6. FOL-STRATIFICATION

The operational semantics appeals to the abstract semantics sem . In this section, we define FOL-stratification and instantiate sem to the FOL-perfect model semantics. FOL-stratification is a generalisation of local stratification in which the restriction on negative literals is generalised to non-atomic FOL formulas. The construction of FOL-perfect models similarly generalises the construction of perfect models.

As usual in logic programming, we treat a logic program P with variables as standing for the set $ground(P)$ of all its ground instances over the Herbrand universe. By a ground instance of a clause $head(X) \leftarrow body(X, Y)$ we mean a clause of the form $head(x) \leftarrow body(x, y)$, where x and y are sets of ground terms substituted for the sets of variables X and Y respectively. The variables in X and Y do not include any variables bound by quantifiers in FOL conditions in $body(X, Y)$.

Definition 6.1 (FOL-stratification). Let P be a ground logic program. Let $H = \cup_{0 \leq i \leq a} H_i$, be a partitioning and ordering of the Herbrand base H of P . For $A \in H$, let $stratum(A) = i$ if and only if $A \in H_i$. Then P is *FOL-stratified* with respect to H_i , $0 \leq i \leq a$, if and only if for every clause $head \leftarrow body$ in P and for every condition C in $body$:

- if C is an atomic condition, then $stratum(C) \leq stratum(head)$
- if C is a non-atomic FOL condition, then $stratum(A) < stratum(head)$
- for every atomic subformula A of C .

The definition of FOL-perfect model iteratively uses the perfect model of lower stratum predicates to evaluate FOL conditions in the bodies of clauses, *reducing* the clauses to Horn clauses, and generating the perfect model of the next higher stratum as the minimal model of the reduced clauses:

Definition 6.2 (FOL-perfect model). Let P be an FOL-stratified ground logic program with respect to H_i , $0 \leq i \leq a$. Let P_i be the set of all clauses $head \leftarrow body$ in P such that $stratum(head) = i$. Then P_0 is a set of Horn clauses.

The *FOL-perfect model* of P is defined by:

1. $perfect(P_0) = min(P_0)$.
2. $perfect(P_{i+1}) = min(P_{i+1} \cup perfect(P_i))$.
3. If β is a limit ordinal, then $perfect(P_\beta) = \cup_{0 \leq i < \beta} perfect(P_i)$.
4. $perfect(P) = perfect(P_a)$.

Case 2 of the definition appeals to the notion of the minimal model of a program P_{i+1} possibly containing FOL conditions defined by $perfect(P_i)$. The definition of the minimal model of a set of Horn clauses generalises naturally to this case:

Definition 6.3 (Minimal model of program with FOL conditions). Let $P = I \cup E$ be a ground logic program with Herbrand base H , where E is a set of ground atoms defining all the predicates in $H_E \subseteq H$ that occur in FOL conditions in the clauses I .

Then the minimal model $min(P)$ of P is the smallest set $M \subseteq H$ such that, for every clause $head \leftarrow body$ in P , $head \in M$ if $body$ is true in M , where an FOL condition all of whose predicates belong to H_E is true in M if and only if the condition is true in E .

Notice the definition exploits the dual interpretation of the Herbrand interpretation E both syntactically as a set of sentences in $P = I \cup E$, and semantically as determining the truth of FOL conditions whose predicates belong to H_E .

Notice also that the generation of $min(P)$ can be regarded as a two-stage process: First the FOL conditions whose predicates all belong to H_E are evaluated in E , resulting in a set of Horn clauses $reduct(P, E)$, then $min(P)$ is generated as the minimal model of $reduct(P, E)$. The reduct $reduct(P, E)$ generalises the treatment of negative literals in the Gelfond and Lifschitz [1988] reduct to FOL-conditions:

Definition 6.4 (Reduct). Let $P = I \cup E$ be a ground logic program with Herbrand base H , where E is a set of ground atoms defining all the predicates in $H_E \subseteq H$ that occur in FOL conditions in the clauses in I .

$reduct(P, E)$ is the set of Horn clauses generated from P by deleting all FOL conditions in the bodies of clauses in I that are true in E and deleting all clauses in I that have an FOL condition that is false in E .

Note that E is contained in $reduct(P, E)$. Thus case 2 of the definition of FOL-perfect model could be rewritten alternatively as:

2. $perfect(P_{i+1}) = min(reduct(P_{i+1}, perfect(P_i)))$

where min is the usual minimal model of a set of Horn clauses. We will see later that this alternative formulation of the definition has the advantage that it extends naturally to the case in which the program P is not statically FOL-stratified, but becomes FOL-stratified dynamically during the construction of the perfect model.

The definition of FOL-perfect model reduces to the definition of perfect model if all non-atomic FOL conditions are simply negative literals. As in the case of perfect

models of locally stratified programs, FOL-perfect models of FOL-stratified programs do not depend upon the stratification, always exist and are unique.

With the definitions of FOL-stratification and FOL-perfect model now in place, we can instantiate the abstract specification of the computational task, given in definition 4.4, to the case in which all the different components of an LPS framework $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$ are individually FOL-stratified.

Recall that definition 4.4 requires a specification of the semantics of the following combinations of the components of the framework:

$$\begin{aligned} & sem(\mathbf{L} \cup \mathbf{S}^* \cup ev^*) \text{ and} \\ & sem(D_{post} \cup L_{int} \cup L_{timeless} \cup S_{i-1}^* \cup ev_i^*), \text{ for all } i > 0 \end{aligned}$$

Thus we need to show that each of the combined programs

$$\begin{aligned} & \mathbf{L} \cup \mathbf{S}^* \cup ev^* \text{ and} \\ & D_{post} \cup L_{int} \cup L_{timeless} \cup S_{i-1}^* \cup ev_i^*, \text{ for all } i > 0, \end{aligned}$$

is also FOL-stratified.

In the case of the first program, it suffices to put the Herbrand base of $\mathbf{S}^* \cup ev^*$ in the lowest stratum followed by the stratifications of the head predicates of $L_{timeless}$, L_{int} , L_{temp} , and L_{events} , in that order.⁶

Similarly in the case of the programs $D_{post} \cup L_{int} \cup L_{timeless} \cup S_{i-1}^* \cup ev_i^*$, it suffices to put the Herbrand base of $S_{i-1}^* \cup ev_i^*$ in the lowest stratum followed by the stratifications of the head predicates of $L_{timeless}$, L_{int} and D_{post} , in that order.

With these stratifications, the computational task is well-defined:

Definition 6.5 (Computational task for FOL-stratified programs). Given an LPS program $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$, in which \mathbf{L} and D_{post} are FOL-stratified, the *computational task* is as given in definition 4.4 with $sem = perfect$.

7. SOUNDNESS AND COMPLETENESS

In this section we discuss the soundness and completeness of the operational semantics OS of LPS for the case $sem = perfect$.

THEOREM 7.1 (SOUNDNESS) Given an LPS program $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$, an initial state S_0 , and initial goal state G_0 , suppose for every set ext_i of external events, where $i \geq 1$, the OS generates a set $acts_{i+1}$ of actions.

Let \mathbf{S}^* and ev^* be the resulting sequences of states and events. Then $\mathbf{R} \cup G_0$ is *true* in $perfect(\mathbf{L} \cup \mathbf{S}^* \cup ev^*)$, if for every top-level goal clause C added in a goal state G_i , $i \geq 0$, there exists a goal state G_j such that $i \leq j$ and C is reduced to *true* in G_j .

Note that, in the special case where the sequence ev^* is finite, the theorem states that if the goal state eventually becomes *true*, then $\mathbf{R} \cup G_0$ is *true*. Note also that the theorem refers to $\mathbf{R} \cup G_0$, rather than to $\mathbf{R} \cup G_0 \cup D_{pre}$, because the requirement that D_{pre} be true in $perfect(\mathbf{L} \cup \mathbf{S}^* \cup ev^*)$ is covered by step 0 of the OS.

⁶ Note that other stratifications produce the same result: for example, the order $L_{timeless}$, L_{temp} , L_{int} , and L_{events} .

SKETCH OF PROOF. The statement of the theorem mimics the definition of truth for reactive rules and goal clauses. In particular, a sentence in the form of a reactive rule $\forall X [antecedent \rightarrow \exists Y consequent]$ is true in a model, if whenever an instance of the *antecedent* becomes true the corresponding instance of the *consequent* becomes true.

But whenever an instance of the *antecedent* becomes true, the corresponding instance of the *consequent* is added as a top-level goal clause C to the current goal state G_i . The fact that the corresponding instance of the *consequent* becomes true is equivalent to there existing a goal state G_j where $i \leq j$ and C is reduced to *true* in G_j .

The only-if half of the theorem also holds under certain conditions on the non-deterministic choices made in step 2 of the OS. In particular, the OS should perform every goal-reduction that is possible in step 2.2, to ensure that any sub-goals that are true in the model generated so far are recognized as being true by reducing them to true. Similarly, the bound *MAX* on the amount of time available for reducing composite events in step 2.1 should be large enough, to ensure that any subgoals that are true in the model generated so far are recognized as true.

As pointed out in [Kowalski and Sadri 2012b], the operational semantics is incomplete. In particular, it cannot (1) preventively make a reactive rule true by making its *antecedent* false, or (2) proactively make a reactive rule true by making its *consequent* true before its *antecedent* becomes true.

Examples of these two kinds of incompleteness include:

1. $attacks(X, you, T_1) \wedge \neg prepared\text{-}for\text{-}attack(you, T_1)$
 $\rightarrow surrender(you, T_2) \wedge T_1 < T_2 \leq T_1 + \delta$

The OS cannot make the rule true by performing actions to make *prepared-for-attack(you, T)* true and so $\neg prepared\text{-}for\text{-}attack(you, T)$ false.

2. $enter\text{-}bus(T_1) \rightarrow have\text{-}ticket(T_2) \wedge T_1 < T_2 \leq T_1 + \varepsilon$

The OS cannot make the rule true by performing actions to make *have-ticket(T₂)* true before *enter-bus(T₁)*.

We have investigated the completeness of LPS with respect to the generation of more restricted *supported models*. Informally speaking and ignoring composite events, a Herbrand model $M = perfect(L \cup S^* \cup ev^*)$ of a set of reactive rules R is supported if for every *action* in every act_i in M there is an instance of a reactive rule in R of the form:

$$antecedent \rightarrow early\text{-}consequents \wedge action \wedge other\text{-}consequents$$

such that $antecedent \wedge early\text{-}consequents$ is true in M . It is possible to show that, under certain conditions, the OS can generate all such supported models. However, we do not discuss this issue further in this paper.

8. SOLVING THE COMPUTATIONAL ASPECT OF THE FRAME PROBLEM

In this section, we show that the models obtained by destructive updates in LPS are identical to the models obtained by using the event theory ET :

$$\begin{aligned} holds(P, T_2) &\leftarrow initiated(P, T_1, T_2) \\ holds(P, T_2) &\leftarrow holds(P, T_1) \wedge succ(T_1, T_2) \wedge \neg terminated(P, T_1, T_2) \end{aligned}$$

We also define a generalization of FOL-stratification, which is needed to define the natural, intended model of $\mathbf{Q} = ET \cup D_{post} \cup L \cup S_0^* \cup ev^*$. We will then show that the intended model of \mathbf{Q} is identical to the FOL-perfect model of $L \cup S^* \cup ev^*$.

8.1 Weak stratification and weakly perfect models

The intended model of \mathbf{Q} is constructed by partitioning the Herbrand base H of \mathbf{Q} into strata associated with the succession of time points $t_0, \dots, t_i, t_{i+1}, \dots$ determined by $succ(t_i, t_{i+1}) \in perfect(L_{temp} \cup L_{timeless})$:

$$\begin{aligned} H_0 &= \{holds(p, t_0) \mid p \text{ is an extensional fluent}\} \cup \\ &\quad \{a \mid a \text{ is an atom with a time-independent predicate}\} \cup \\ &\quad \{a \mid a \text{ is an atom with a temporal predicate, including } succ\} \cup \\ &\quad \{happens(e, t, u) \mid e \text{ is a simple event, and } t \text{ and } u \text{ are time points}\} \\ \text{For } i \geq 0, H_{3i+1} &= \{holds(p, t_i) \mid p \text{ is an intensional fluent}\} \\ H_{3i+2} &= \{initiated(p, t, t_{i+1}) \mid p \text{ is an extensional fluent, and } t \text{ is a time point}\} \cup \\ &\quad \{terminated(p, t, t_{i+1}) \mid p \text{ is an extensional fluent, and } t \text{ is a time point}\} \\ H_{3i+3} &= \{holds(p, t_{i+1}) \mid p \text{ is an extensional fluent}\} \\ H_{\omega+1} &= \{happens(e, t, u) \mid e \text{ is a composite event, and } t \text{ and } u \text{ are time points}\} \end{aligned}$$

The sets H_0 and H_{3i+1} are themselves stratified: H_0 is partitioned into strata corresponding to the stratification of $L_{timeless} \cup L_{temp}$, and H_{3i+1} is partitioned into strata corresponding to the stratification of L_{int} .

Applied to the predicates in the heads of clauses in \mathbf{Q} , this stratification of H determines an associated stratification of $ground(\mathbf{Q})$:

$$\begin{aligned} Q_0 &= S_0^* \cup ground(L_{timeless}) \cup ground(L_{temp}) \cup ev^* \\ \text{For } i \geq 0, Q_{3i+1} &= \{holds(p, t_i) \leftarrow body \in ground(L_{int})\} \\ Q_{3i+2} &= \{initiated(p, t, t_{i+1}) \leftarrow body \in ground(D_{post})\} \cup \\ &\quad \{terminated(p, t, t_{i+1}) \leftarrow body \in ground(D_{post})\} \\ Q_{3i+3} &= \{holds(p, t_{i+1}) \leftarrow body \in ground(ET)\} \\ Q_{\omega+1} &= ground(L_{events}) \end{aligned}$$

Unfortunately, \mathbf{Q} is not FOL-stratified, because Q_{3i+3} contains unstratified instances of the frame axiom:

$$holds(p, t_{i+1}) \leftarrow holds(p, t_j) \wedge succ(t_j, t_{i+1}) \wedge \neg terminated(p, t_j, t_{i+1})$$

where $j > i+1$ and $holds(p, t_j)$ is at a higher stratum than $holds(p, t_{i+1})$. The problem and its solution are similar to those for the program [Apt and Bol 1994]:

Succ: $successor(X, s(X))$
Even: $even(0)$
 $even(Y) \leftarrow successor(X, Y) \wedge \neg even(X)$

The program has a natural stratification with all ground instances of *Succ* in the lowest stratum, and with $even(s(n))$ in the stratum one higher than the stratum of $even(n)$. However, the program is not locally stratified, because there are ground instances of the second clause in *Even*, for example $even(0) \leftarrow successor(s(0), 0) \wedge \neg even(s(0))$, where the negative condition is at a higher stratum than the head.

The problem is that the definition of local-stratification is too static, and does not take into account the dynamic stratification obtained by the use of the reduct in the construction of the perfect model. If we ignore the fact that the original program is not locally stratified, and attempt to generate its perfect model, we see that the second clause in *Even* is replaced in effect by the locally stratified clauses:

$$even(s(t)) \leftarrow \neg even(t) \quad \text{for all } t \text{ such that } successor(t, s(t)) \in min(Succ).$$

This dynamic variant of local stratification is called weak stratification, and it can also be applied more generally to programs with FOL conditions. The definition is virtually identical to the definition of perfect model, with *perfect* replaced by *weakly-perfect*, and without requiring the program to be statically stratified in advance:

Definition 8.1 (Weak FOL-stratification and weakly FOL-perfect model). Let P be a ground logic program. Let $H = \cup_{0 \leq i \leq a} H_i$, be a partitioning and ordering of the Herbrand base H of P . Let P_i be the set of all clauses $head \leftarrow body$ in P such that $stratum(head) = i$.

1. If P_0 is a set of Horn clauses all of whose conditions are in H_0 , then:

$$weakly-perfect(P_0) = min(P_0).$$

2. If P_i is weakly FOL-stratified, with intended model $weakly-perfect(P_i)$, and if $reduct(P_{i+1}, weakly-perfect(P_i))$ is a set of Horn clauses all of whose conditions are in H_{i+1} , then P_{i+1} is *weakly FOL stratified* and:

$$weakly-perfect(P_{i+1}) = min(reduct(P_{i+1}, weakly-perfect(P_i))).$$

3. If β is a limit ordinal, and for all $0 \leq i < \beta$, P_i is weakly FOL-stratified with intended model $weakly-perfect(P_i)$, then P_β is *weakly FOL-stratified* and:

$$weakly-perfect(P_\beta) = \cup_{0 \leq i < \beta} weakly-perfect(P_i).$$

4. $weakly-perfect(P) = weakly-perfect(P_a)$

It is possible to show that if a program is weakly FOL-stratified with respect to one stratification, then it is weakly FOL-stratified with respect to every other stratification, and consequently the weakly FOL-perfect model is unique.

In the case of the program Q , the use of the reduct in the construction of the weakly-perfect model eliminates the unstratified instances of Q_{3i+3} and replaces them by the stratified instances:

$holds(P, t_{i+1}) \leftarrow holds(P, t_i) \wedge \neg terminated(P, t_i, t_{i+1})$
for all t_i, t_{i+1} such that $succ(t_i, t_{i+1}) \in perfect(L_{temp} \cup L_{timeless})$.

8.2 The Frame Theorem

The following theorem links the two characterizations of the semantics of LPS.

THEOREM 8.2 (FRAME THEOREM).

$$perfect(L \cup S^* \cup ev^*) = weakly-perfect(ET \cup D_{post} \cup L \cup S_0^* \cup ev^*) \\
- \{head \mid head \leftarrow body \in ground(D_{post})\}.$$

Equivalently, $perfect(D_{post} \cup L \cup S^* \cup ev^*)$
 $= weakly-perfect(ET \cup D_{post} \cup L \cup S_0^* \cup ev^*)$

SKETCH OF PROOF. It suffices to show that

$$perfect(D_{post} \cup L_{int} \cup L_{timeless} \cup L_{temp} \cup S^* \cup ev^*) \\
= weakly-perfect(ET \cup D_{post} \cup L_{int} \cup L_{timeless} \cup L_{temp} \cup S_0^* \cup ev^*).$$

This is because, in both models, L_{events} is used only in the last strata to superimpose composite events on the underlying sequence of states and simple events.

Note that $ground(ET) = Q_3 \cup Q_6 \cup \dots \cup Q_{3i+3} \cup \dots$, where

$$Q_{3i+3} = \{holds(p, t_{i+1}) \leftarrow body \in ground(ET)\}$$

It suffices to show that for all $i > 0$,

$$perfect(D_{post} \cup L_{int} \cup L_{timeless} \cup L_{temp} \cup S_0^* \cup S_1^* \cup \dots \cup S_{i+1}^* \cup ev^*) \\
= weakly-perfect(D_{post} \cup L_{int} \cup L_{timeless} \cup L_{temp} \cup S_0^* \cup Q_3 \cup \dots \cup Q_{3i+3} \cup ev^*).$$

This can be proved by induction on i .

9. COMPARISON WITH OTHER WORK

LPS evolved from our attempts to reconcile and combine conflicting approaches to computing in such different areas as logic programming, production systems, active and deductive databases, agent programming languages, and the representation of causal theories in AI.

9.1 Deductive databases

Our attention was first drawn to the distinction between reactive rules and logic programs by the distinction made by [Nicolas and Gallaire 1978] between deduction rules and integrity constraints in deductive databases, both of which have a logical semantics. However, the exact nature of the relationship between their semantics was the subject of considerable debate in the early 1980s.

The two main views, to begin with, were the *consistency view* and the *theoremhood view*, both of which were defined relative to the completion of the database [Clark 1978]. In the consistency view, an integrity constraint is satisfied if it is

consistent with the completion of the database. In the theorem-hood view, it is satisfied if it is a theorem, logically entailed by the completion.

[Reiter 1988] also proposed an *epistemic view*, according to which integrity constraints are statements about what the database knows. However, [Reiter 1988] also showed that in many cases all three views are equivalent. For relational databases, in particular, the three views are also equivalent to the standard view that a database satisfies an integrity constraint if it is *true* in the database regarded as a Herbrand model. The semantics of LPS is an extension of this idea in two ways: First, it extends the idea of a database defined by a set of ground atomic sentences to the idea of the database defined by an FOL-stratified logic program. Second, it extends the notion that an integrity constraint is true in a model representing a database state to the idea that the integrity constraint is true in a model representing the entire collection of states and events.

The use of logic programs in LPS with FOL conditions was largely inspired by transaction logic [Bonner and Kifer 1993], which uses such programs to define database transactions. Although transaction logic focuses primarily on performing composite actions, reactive rules can also be programmed using transactions. Like LPS, transaction logic also employs destructive updates. But it employs a possible world semantics, in which the semantics of transactions is defined in terms of paths between possible worlds.

9.2 Abductive logic programming

The distinction between logic programs and integrity constraints also underpins abductive logic programming (ALP) [Kakas et al. 1998; Denecker and Kakas 2002]. In ALP, a program consists of a triple $\langle L, IC, A \rangle$, where L is a logic program, IC is a set of integrity constraints, and A is a set of “abducible” predicates, not defined by L . A goal G is an observation to explain or a state to achieve. The goal is solved by generating a set Δ of ground atoms in the vocabulary of the abducible predicates such that $L \cup \Delta$ entails G , and $L \cup \Delta$ satisfies the integrity constraints IC . Similarly to the case of deductive databases, different notions of entailment and integrity constraint satisfaction have been proposed.

LPS is a variant of ALP, in which the abducible predicates are restricted to simple actions, and observations are events and fluents that can be queried in the global state and do not require explanation. Moreover, the semantics is simplified so that entailment and integrity satisfaction are understood in the same way, as meaning that $G \cup IC$ is true in the FOL-perfect model of $L \cup \Delta$. The operational semantics of LPS is a variant of the IFF proof procedure [Fung and Kowalski 1997], originally developed for Kunen’s three valued completion semantics [Kunen 1987].

9.3 Logic programming semantics

The inclusion of FOL conditions in LPS is an important feature, motivated by their use in transaction logic to query databases states during the course of performing database updates. FOL-stratification and weak-FOL stratification with their associated models provide a natural setting for evaluating such FOL conditions. Moreover, weak FOL-stratification also provides a natural semantics for event theories such as *ET*. But stratification goes against current trends in logic programming, where the dominant approaches are the well-founded semantics [Van Gelder et al. 1991] and stable model semantics [Gelfond and Lifschitz 1988].

It may, of course, be possible to redo the semantics of FOL conditions in other approaches. In particular, there may be a natural way to represent alternative sets of possible concurrent events and the resulting states as alternative stable models. It may also be possible to modify the evaluation of FOL conditions to use a three-valued semantics.

On the other hand, the extension of stratification and perfect models to the case of FOL conditions may have other uses. For example, the application of Datalog to declarative networking [Loo et al 2009; Hellerstein 2010; Loo et al 2012] makes heavy use of stratification, and the extensions of stratification in this paper might also be useful in that domain.

9.3 Agent systems

LPS is a direct descendant of our work on ALP agents [Kowalski and Sadri 1999, Kowalski 2011], which embed ALP in the thinking component of a BDI-like agent [Rao and Georgeff 1995] cycle. In ALP agents, the logic program L represents the agent's beliefs, and the goals and integrity constraints $G \cup IC$ represents the agent's desires. The logic program L includes a deductive database that represents the agent's view of its environment. The database is updated by means of an event theory, which uses frame axioms. The ALP agent approach was developed further in the KGP agent model [Kakas et al. 2004; Mancarella et al. 2009]. In contrast, LPS and most practical agent systems employ a destructively updated database that represents the current state.

A number of other authors have also developed agent languages and systems within a logic programming context. For example in both DALI [Costantini and Tocchino 2004; Costantini and Tocchino 2006] and EVOLP [Brogi et al. 2002], events transform an initial agent logic program into a sequence of logic programs. The semantics of this evolutionary sequence is given by the associated sequence of models of the sequence of programs. In LPS, this sequence is represented by a single model by using time stamps.

In EVOLP, the sequence of logic programs is non-deterministic and allows for a directed graph of possible state evolutions, because stratification is not imposed. In LPS, non-determinism arises from the possibility of choosing different actions to generate state transitions, while still imposing stratification. Moreover, EVOLP allows rules to be updated, and not only fluents as in LPS.

FLUX [Thielscher 2005] is an agent language with several features similar to LPS, including the use of destructive assignment to update states. In FLUX, these states are not represented by atomic sentences as in LPS, but are reified as terms in a list-like structure.

[Thielscher 2010] provides a declarative semantics for AgentSpeak by defining its cycle and procedures by means of a meta-interpreter represented as a logic program. Like LPS, the resulting agent language incorporates a formal transition theory. However, unlike LPS, the language does not distinguish between different kinds of AgentSpeak procedures, according to their different functionalities. LPS, in contrast, distinguishes between reactive rules, and logic programs, representing different kinds of procedures in different ways. Somewhat closer to LPS is the agent architecture of [Hayashi et al. 2005; Hayashi et al 2009], which separates the representation of reactive rules and planning clauses. Planning is done by means of Hierarchical Task Networks, which are like logic programs that reduce composite events to simpler events in LPS.

[Eiter et al. 1999] define an extension of logic programming in which the clauses represent the conditions under which actions are permitted, forbidden, obliged or waived. All reasoning takes place and is completed within a single iteration of the agent cycle. In the LPS cycle, reasoning can be interrupted both to assimilate events and to generate actions.

In contrast with approaches that map agent programs into logic programs, MetaTEM [Fischer 1994] maps agent programs into temporal modal logic sentences of the form “past or present conditions imply present or future conclusions”. As in LPS, computation attempts to generate a model in which such agent programs are *true*. In contrast with MetaTEM, which employs a possible world semantics and frame axioms for updating states, LPS uses a language with an explicit representation of time, an extension of the perfect model semantics, and an operational semantics with destructive updates.

9.4 Active databases

As pointed out by [Bailey et al. 1995], although they differ in their intended applications and research communities, agent systems and active databases employ similar approaches to programming reactive systems. For example, AgentSpeak [Rao 1996] employs agent programs that are plans consisting of a triggering event, a context, which specifies the conditions that should hold when the plan is triggered, and a body, which specifies the goals the agent should achieve or test, and the actions the agent should execute. Active databases employ similar event-condition-action (ECA) rules to react to events, test conditions and perform actions. Both agent system plans and ECA rules maintain a destructively updated database state, but lack a declarative semantics.

A number of researchers, working mainly in the deductive database area, have addressed the problem of developing a declarative, logic-based semantics for active databases. In the majority of these approaches ECA rules are mapped into logic programs to provide them with a logic programming semantics.

[Zaniolo 1993], for example, uses a situation calculus-like representation with frame axioms, and reduces ECA rules to logic programs. Statelog [Lausen et al. 1998] also uses a situation-calculus-like representation for the succession of database states. Like Zaniolo, Statelog represents ECA rules as logic programs, and gives them a semantics based on logic programming.

[Fernandes 1997] also views ECA rules in terms of change of state, but use the event calculus as the basis for an ECA language coupled with a deductive database. The event calculus is used to evaluate the condition part of the ECA rules and to provide a specification for the effects of executing the action part. The ECA language also allows the recognition of complex events from an event history.

ERA (Evolving Reactive Algebraic Programs) [Alferes et al. 2006] extends the dynamic logic programming system EVOLP [Brogi et al. 2002] by adding complex events and actions as well as external actions. ERA combines ECA and logic programming rules, and the firing of the ECA rules can generate actions that add or delete ECA or logic programming rules, as well as external actions. In the operational semantics the ECA and logic programming rules maintain their distinct characteristics, but in the declarative semantics the ECA rules are translated into logic programs. The declarative semantics is based on a variant of stable models developed for EVOLP.

[Caroprese et al.2006] also transform active integrity constraints into logic programs. They characterise the set of “founded” repairs for the database as the

stable model of the database augmented by the logic programming representation of the active integrity constraints. [Fraternali and Tanca 1995] also consider active databases but provide a logic-based *core* syntax for representing low-level, procedural features of active database rules. They provide procedural semantics for core rules and show how this can capture the procedural semantics of known active database systems.

9.5 Production systems

Arguably, production systems [Newell 1973], in which programs are expressed as condition-action rules, are the simplest example of a reactive system, and the earliest ancestor both of agent systems and active databases. It was the attempt to understand the difference and relationship between production rules and logic programming rules that eventually led to our development of LPS. Several other authors have made related attempts, with the aim of providing production rules with a declarative semantics. In the majority of these approaches production rules are mapped into logic programs.

[Raschid 1994] focuses on the use of production rules as reactive rules and as forward-reasoning logic rules. She first maps rules that add facts into logic programs, and rules that delete facts into integrity constraints. She then transforms the resulting combination of logic programs and integrity constraints into normal logic programs, and uses the fixed point semantics of logic programming to chain forward and simulate the production system cycle. [Baral and Lobo 1995] translate production rules into the situation calculus represented as a logic program with the stable model semantics. Their use of the situation calculus is similar to our characterization of the computational task using the event theory *ET*, and thus uses a frame axiom. [Dung and Mancarella 2002], on the other hand, use an argumentation theoretic framework to provide semantics for production rules extended with negation as failure.

Recently, there has been a revival of work on implementing production systems in logic programming terms. For example, [Damasio et al. 2010] use incremental Answer Set Programming (ASP) to realize different conflict resolution strategies for the RIF-PRD production system dialect. [Eiter et al 2012] simulate production systems in ACTHEX, an ASP framework with an interface to an external environment. The simulation does not use an explicit representation of state, but achieves state changes by updating and accessing the environment via action atoms and external atoms. [Rezk and Kifer 2012] combine production rules and ontologies, using transaction logic.

In comparison with other approaches that map reactive rules into logic programs, our approach has been to develop a semantics that respects the distinct natures of logic programs and reactive rules. In LPS, reactive rules and logic programs are both expressed in logical form, but logic programs represent beliefs that determine model theoretic structures, and reactive rules represent goals that are meant to be true in those models.

9.6 Causal theories in AI

ALP agents [Kowalski and Sadri 1999; Kowalski 2011] and the KGP agent model [Kakas et al. 2004] employ the event calculus to represent and reason about the relationship between fluents, actions and other events. Unlike the situation calculus, which reifies global states or situations, the event calculus reifies time

points and events. However, in both the situation calculus and event calculus, axioms are used to derive atomic sentences representing states of the world. Destructive updates are not possible, because it is not possible to change the axioms in the middle of a proof.

However, because ALP agents are embedded in an agent cycle, they can also directly observe the current state of the world, and thereby avoid the need to reason about it. In other words, the world can serve as its own representation, as advocated by [Brooks 1991]. This ability of an agent to observe the world instead of reasoning about its representation highlights the fact that the world is a semantic structure that gives meaning to an agent’s thoughts. Because the world is a semantic structure, it is not constrained by the restrictions of axioms that are not allowed to change during the course of a proof.

Adopting this view of the world as a semantic structure is compatible with a model-theoretic semantics of ALP, in which the logic program L and set of assumptions Δ determines a model of $L \cup \Delta$ that makes the goal and integrity constraints $G \cup IC$ all true. Moreover, it justifies the destructive updates and model-theoretic semantics of LPS. To the best of our knowledge, LPS is the only framework employing a causal theory that combines destructive updates with a logic-based semantics.

We do not claim that the use of destructive updates eliminates all need to reason with frame axioms. On the contrary, frame axioms are needed to prove certain properties of LPS programs. This is an area of work that we are currently investigating.

9.7 Parallelism and concurrency

LPS combines an AI approach to the representation of concurrent actions with a Linda-like use of a shared state as a coordination medium. The AI component comes from the use of the domain theory D , to reason about the combined effects of concurrent actions, in the spirit of [Reiter 1996]’s treatment of concurrent actions in the situation calculus and [Miller and Shanahan 2002]’s treatment in the event calculus.

Recently, [Khandelwal and Fox 2012] have extended Miller and Shanahan’s approach, to define the effects of multiple actions by using aggregate formulas in first-order logic. Our approach can be regarded as an approximation to theirs, and would benefit from a similar extension using aggregate formulas.

Unlike some other approaches that use message passing to handle concurrency, LPS uses a Linda-like shared state, which is similar also to the blackboard architecture used in AI. Our assumption that the environment non-deterministically decides which sets of possible concurrent events actually occur is similar to the use of a “supervisor” in [Dovier et al. 2012], to arbitrate between the conflicting actions of different agents in the pursuit of different goals. In addition, [Dovier et al. 2012] also provides communication primitives, to allow agents to resolve conflicts through negotiation. The semantics is defined in terms of state transitions, but does not provide an explicit treatment of reactivity.

Although the approach to concurrency behaves naturally with such challenging problems in concurrent programming as the dining philosophers’ problem, we need to investigate more deeply the relationship with the treatment of parallelism and concurrency in databases management systems and conventional programming languages more generally. In this respect, it is encouraging to note the recent developments [Hellerstein 2010] in the use of Datalog and the explicit

representation of time for programming distributed and parallel systems. Although frame axioms are represented explicitly in [Hellerstein 2010], they are not used in the implementation, using instead “traditional storage technology rather than re-deriving tuples each timestep”. Our frame theorem can be regarded as a justification for the use of such technology.

9.8 Reactive systems

An LPS framework $\langle \mathbf{R}, \mathbf{L}, \mathbf{D} \rangle$ is essentially a reactive system in which logic programs \mathbf{L} and causal theories \mathbf{D} play a supporting role to reactive rules \mathbf{R} . [Harel 1986] contrasts reactive systems with “transformational systems”, which transform inputs into outputs in a mathematically well-behaved manner. In contrast with transformational systems, reactive systems are “event-driven, continuously having to react to external and internal stimuli”. He further characterises them as being an extension of state transition systems, having the general form “when event α occurs in state A , if condition C is true at the time, the system transfers to state B ”. [Harel 2009] notes that StateCharts, a graphical language for reactive systems, is “the heart of the UML - what many people refer to as its driving behavioral kernel”.

As [Reisig 2012] puts it, an initialized, deterministic transition system is “a triple $C = (Q, I, F)$ where Q is a set (its elements are denoted as states), $I \subseteq Q$ (the initial states), and $F : Q \rightarrow Q$ (the next-state function)”. Transition systems can be extended to reactive systems in which the transition from one state to the next is “not conducted by the program, but by the outside world”.

But even in their simpler “initialized, deterministic” form, transition systems have been proposed as a general model of Computing. Reisig points out that in [Knuth 1973], the first volume of *The Art of Computer Programming*, Donald Knuth suggests their use as a general semantics for algorithms.

LPS can be viewed as an attempt to reconcile Harel’s two kinds of computational formalism, with reactive rules providing the main reactive component of the system, and logic programs providing structure for the “transformational part”. In addition, LPS also attempts to incorporate deductive database functionality and features of causal theories in AI.

10 Future work

LPS has its origins in AI knowledge representation and reasoning languages, but for the sake of efficiency and to focus on the features required for database and programming applications, the AI features have been deliberately restricted and simplified. For example, the abductive explanation of observations, which was one of the main motivations of ALP, has been deliberately left out. Similarly, the ability to perform preventative maintenance, which is a feature of the IFF proof procedure for ALP, has also been left out.

There are two complementary, directions for future work. One direction is to reintroduce into LPS some of the more powerful, but also more expensive features of ALP agents – for example the planning clauses in some of the earlier versions of LPS. Such features might also include more expressive integrity constraints, bearing in mind that reactive rules are just a species of integrity constraint in ALP.

The other direction is to further restrict the framework to make it more efficient or to specialize it for particular application domains – for example, by restricting the use of function symbols, as in Datalog. This direction also includes further

development of the operational semantics – for example to specify efficient strategies for executing composite events in the antecedents and consequents of rules.

There is also a third direction, which combines the other two, by adding more powerful features for particular classes of applications. This includes extending the syntax of FOL conditions to include the use of aggregation operators and more complex kinds of composite events.

The extension to include aggregation operators should not be too difficult, because it requires only extending the definition of the truth of a sentence in a Herbrand interpretation. Moreover, it can be implemented in a similar manner to the implementation of aggregate operators in relational database systems, Prolog and ASP.

The extension to include more complex composite events does not require any extension of the semantics, but requires only extending the current state to include a window of previous events. This window can then be queried along with other facts in the current state using arbitrary FOL conditions, augmented perhaps with aggregation operators.

We have implemented a prototype of LPS in LPA Prolog, which includes some of the details necessary for a more complete language. For example, the implementation uses a Prolog-like depth-first search to choose goal clauses for goal reduction. Some obvious additional improvements include the use of a constraint solver for handling temporal constraints and the use of a UML-like graphical external syntax.

ACKNOWLEDGEMENTS

We are grateful to Imperial College for EPSRC Pathways to Impact funding, which has supported the implementation of LPS. Many thanks also to David Kinny and Ken Satoh for helpful discussions, and to Ken Satoh and Luis Pereira for their comments on an earlier draft of this paper.

REFERENCES

- Alferes, J.J., Banti, F., Brogi A. 2006. An Event-Condition-Action Logic Programming Language. In: 10th European Conference on Logics in Artificial Intelligence, M. Fisher, W. van der Hoek, B. Konev and A. Lisitsa (eds.), JELIA06: Lecture Notes in Artificial Intelligence 4160, Springer-Verlag. 29- 42.
- Apt, K. R., & Bol, R. N. (1994). Logic programming and negation: A survey. *The Journal of Logic Programming*, 19, 9-71.
- Bailey, J., Georgeff, M., Kemp, D., Kinny, D., & Ramamohanarao, K. 1995. Active databases and agent systems—A comparison. *Rules in Database Systems*, 342-356.
- Baral, C., Lobo, J. 1995. Characterizing production systems using logic programming and situation calculus <http://www.cs.utep.edu/baral/papers/char-prod-systems.ps>
- Bonner, A., Kifer, M. 1993. Transaction logic programming. In Warren D. S., (ed.), *Logic Programming: Proc. of the 10th International Conf.*, 257-279.
- Brogi, A., Leite, J. A., 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.
- Brooks, R.A. 1991. Intelligence Without Representation, *Artificial Intelligence* 47, 139-159.

- Caroprese, L. Greco, S., Sirangelo, C., Zumpano, E. 2006. Declarative Semantics of Production Rules for Integrity Maintenance. In: 22nd International Conference on Logic Programming, Etalle, S., Truszczynski, M. (eds.), LNCS 4079, 26—40.
- Carriero, N. and Gelernter, D. 1989. Linda in Context. *Communications of the ACM*. Volume 32 Issue 4.
- Clark, K. 1978. Negation as Failure. In: *Readings in Nonmonotonic Reasoning*, Morgan Kaufmann, 311—325.
- Costantini, S., Tocchio, A. 2004. The DALI Logic Programming Agent-Oriented Language. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAD), vol. 3229, Springer, Heidelberg. 685—688.
- Costantini, S. and Tocchio, A. 2006. About Declarative Semantics of Logic-Based Agent Languages, Dalt 2005, LNAI 3904, Baldoni, M. et al (eds.), 106-123.
- Damáσιο, C., Alferes, J., & Leite, J. 2010. Declarative semantics for the rule interchange format production rule dialect. *The Semantic Web—ISWC 2010*, 798-813.
- Denecker, M. AND Kakas, A. C. 2002. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond*. Springer-Verlag, London, UK, 402—436.
- Dovier, A., Formisano, A. and Pontelli, E. 2012. Autonomous agents coordination: Action languages meet CLP) and Linda, *Theory and Practice of Logic Programming*.
- Dung, P.M., Mancarella, P. 2002. Production Systems with Negation as Failure. *IEEE Transactions on Knowledge and Data Engineering*, Volume 14 , Issue 2, 336—352.
- Eiter, T., Subrahmanian, V.S. and G Pick, G. 1999. Heterogeneous active agents, I. *AI Journal*, vol. 108, no. 1-2, 179-255.
- Eiter, T., Feier, C., & Fink, M. 2012. Simulating production rules using ACTHEX. *Correct Reasoning*, 211-228.
- van Emden, M. and Kowalski, R. 1976. The Semantics of Predicate Logic as a Programming Language, in *JACM*, Vol. 23, No. 4, 733-742.
- Fernandes, A.A.A., Williams, M.H., Paton, N. 1997. A Logic-Based Integration of Active and Deductive Databases. *New Generation Computing*, Volume 15, Number 2, 205—244.
- Fisher, M. 1994. A Survey of Concurrent METATEM - The Language and its Applications. *Lecture notes in computer science*, 827, Springer Verlag 480-505.
- Fraternali P., Tanca L. 1995. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions on Database Systems (TODS) Volume 20, Issue 4 (December 1995)*, 414—471.
- Fung, T.H. and Kowalski, R. 1997. The IFF Proof Procedure for Abductive Logic Programming. *J. of Logic Programming*.
- Gelfond, M., & Lifschitz, V. (1988, August). The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic programming (Vol. 161)*.
- Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Programming* 8 231-274.
- Harel, D. (2009). Statecharts in the making: a personal account. *Communications of the ACM*, 52(3), 67-75.
- Hayashi, H., Cho, K., & Ohsuga, A. 2005. A new HTN planning framework for agents in dynamic environments. *Computational Logic in Multi-Agent Systems*, 55-56.
- Hayashi, H., Tokura, S., Ozaki, F., Doi, M. 2009. Background Sensing Control for Planning Agents Working in the Real World. *International Journal of Intelligent Information and Database Systems*, Inderscience Publishers, 3(4): 483-501.
- Hausmann, S., Scherr, M., Bry, F. 2012. Complex Actions for Event Processing, Research Report, Institute for Informatics, University of Munich.
- Hayes-Roth, B. 1985. A blackboard architecture for control, *Artificial Intelligence*, Volume 26, Issue 3, 251-321.
- Hellerstein, J.M. 2010. The Declarative Imperative: Experiences and Conjectures in Distributed Logic, *SIGMOD Record* 39(1).
- Kakas, A. C., Kowalski, R., Toni, F. 1998. The Role of Logic Programming in Abduction, *Handbook of Logic in Artificial Intelligence and Programming 5*, Oxford University Press, 235-324.

- Kakas, A. C., Mancarella, P., Sadri, F., Stathis, K., and Toni, F. 2004. The KGP model of agency, In *Proc. ECAI-2004*.
- Khandelwal, A., Fox, P. 2012. General Descriptions of Additive Effects via Aggregates in the Circumscriptive Event Calculus.
- Kowalski, R. 1979. *Logic for Problem Solving*, North Holland.
- Kowalski, R. 2011. *Computational Logic and Human Thinking: How to be Artificially Intelligent*, Cambridge University Press.
- Kowalski, R. and Sadri, F. 1999. From Logic Programming Towards Multi-agent Systems, *Annals of Mathematics and Artificial Intelligence*, Volume 25, 391-419.
- 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.
- 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.
- Kowalski, R. and Sadri, F. 2011. Abductive Logic Programming Agents with Destructive Databases, *Annals of Mathematics and Artificial Intelligence*, Volume 62, Issue 1, 129-158.
- Kowalski, R., & Sadri, F. (2012a). Teleo-Reactive abductive logic programs. *Logic Programs, Norms and Action*, 12-32.
- Kowalski, R. and Sadri, F. 2012b. RuleML 2012, A Logic-Based Framework for Reactive Systems, *Rules on the Web: Research and Applications, 2012* – Springer-Verlag. A. Bikakis and A. Giurca (Eds.), LNCS 7438, pp. 1–15.
- Kowalski, R., Sergot, M. 1986. A Logic-based Calculus of Events. In: *New Generation Computing*, Vol. 4, No.1, 67—95.
- Knuth, D. E. 1973, *The Art of Computer Programming*. Vol. 1: Fundamental Algorithms. Addison-Wesley.
- Kunen, K. 1987. Negation in Logic Programming. *Journal of Logic Programming*, 4:4 289—308.
- Lausen, G., Ludäscher, B., May, W. 1998. On Active Deductive Databases: The Statalog Approach. In: *Transactions and Change in Logic Databases*, Decker, H., Freitag B., Kifer, M., Voronkov, A. (eds.), LNCS 1472, Springer.
- Lloyd, J.W., Topor, R.W. 1984. Making PROLOG More Expressive, *Journal of Logic Programming* 1, 3 225-240.
- Loo, B., Gill, H., Liu, C., Mao, Y., Marczak, W., Sherr, M., Wang, A., & Zhuo, W. 2012. Recent Advances in Declarative Networking, Practical Aspects of Declarative Languages-14th International Symposium, PADL 2012.
- Loo, B. T., Condie, B. T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. 2009. Declarative Networking. In *Communications of the ACM (CACM)*.
- Mancarella, P., Terreni, G., Sadri, F., Toni, F., Endriss, U. 2009. The CIFF Proof Procedure for Abductive Logic Programming with Constraints: Theory, Implementation and Experiments. *Theory and Practice of Logic Programming*.
- McCarthy, J. and Hayes, P. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence, *Machine Intelligence 4*, Edinburgh University Press. 463-502.
- Miller R., Shanahan, M. 2002. Some Alternative Formulations of the Event Calculus. *Computational logic: logic programming and beyond*. Springer-Verlag, 452-490.
- Newell, A. 1973. Production Systems: Models of Control Structure. In: Chase W. (ed.), *Visual Information Processing*, 463-526 New York, Academic Press, pp. 463—526.
- Nicolas, J.M., Gallaire, H. 1978. Database: Theory vs. Interpretation. In: Gallaire, H., Minker, J. (eds.), *Logic and Databases*, Plenum, New York.
- Nilsson, N.J. 2001. Teleo-reactive Programs and the Triple-tower Architecture, *Electronic Transactions on Artificial Intelligence*, Vol. 5, Section B, 99-110.
- Pereira, F. C.N., Warren, D. H.D. 1980. Definite clause grammars for language analysis—A survey of the formalism and a comparison with augmented transition networks, *Artificial Intelligence*, Volume 13, Issue 3, 231–278.

- Przymusiński, T. 1987. On the declarative semantics of stratified deductive databases and logic programs. *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, J. Minker (Ed.) 193 – 216.
- Przymusińska, H., Przymusiński, T. 1988. Weakly perfect model semantics for logic programs, Fifth Int'l Conf. Symp. on Logic Programming.
- Raschid, L. 1994. A Semantics for a Class of Stratified Production System Programs. *J. Log. Program.* 21(1): 31–57.
- Rao, A. (1996). AgentSpeak (L): BDI agents speak out in a logical computable language. *Agents Breaking Away*, 42-55.
- Rao, A. S., Georgeff, M. P. 1995. BDI Agents: From Theory to Practice, International Conference on Multiagent Systems - ICMAS , 312-319.
- Reiter, R. 1988. On Integrity Constraints. In: 2nd Conference on Theoretical Aspects of Reasoning about Knowledge, pp. 97—111.
- Reiter, R. 1996. Natural actions, concurrency and continuous time in the situation calculus, In Proceedings of Principles of Knowledge Representation.
- Reisig, W. (2012). The expressive power of abstract-state machines. *Computing and Informatics*, 22(3-4), 209-219.
- Rezk, M., & Kifer, M. (2012). Formalizing production systems with rule-based ontologies. *Foundations of Information and Knowledge Systems*, 332-351.
- Shanahan, M. (1997). *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. MIT press.
- Thielscher, M. (2005). FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5), 533-565.
- Thielscher, M. 2010. Integrating Action Calculi and AgentSpeak. In Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), Lin, F and Sattler, U. (eds.), Toronto.
- Van Gelder, A., Ross, K. A., & Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM (JACM)*, 38(3), 619-649.
- Zaniolo, C. 1993. On the Unification of Active Databases and Deductive databases. In: 11th British National Conference on Databases, 23-39.