Reactive Computing as Model Generation

Robert KOWALSKI and Fariba SADRI
Imperial College London
{rak, fs}@doc.ic.ac.uk

Abstract In this paper we propose a logic-based, framework 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 timestamps with facts and events. But in the operational semantics, facts are updated destructively, without timestamps. We show that the model generated by destructive updates is identical to the model generated by reasoning with facts containing timestamps. We also extend the model with intentional predicates and composite event predicates defined by logic programs containing conditions in first-order logic, which query the current state.

Keywords: State Transition Systems, Reactive Systems, Composite Event Processing, Model Generation, Frame Problem.

§1 Introduction

In this paper, we present a computational framework that uses logic for 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 suitable for more conventional computer applications. It builds on logic programming, but includes imperative language features, including reactive rules and destructive change of state.

In earlier versions[29, 30, 31, 32] of this work, 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 for agent programming, active databases, concurrent systems, and composite (or complex) event processing.

The paper is organised as follows: Section 2 presents an overview of the framework, and Section 3 illustrates the framework by means of examples. Section 4 defines the language more formally, and Section 5 presents the operational semantics and discusses our implementations. Section 6 discusses soundness and completeness. In particular, it shows that destructive updates in LPS generate the same models as the frame axiom. Sections 7 and 8 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 conditions in first-order logic (FOL), 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 Overview

2.1 The Language

An LPS framework is a state transition system, specified by a tuple **<*R*, *L*, *D*>** consisting of a set ***R*** of reactive rules, a logic program ***L***, and a domain theory ***D***, which specifies the preconditions and postconditions of state transitions**.** States S_i are represented by sets of atomic sentences, called facts or fluents. States are like relational databases, but also like program variables or tuples in a coordination language[9]. State transitions take place as the result of event occurrences.

The reactive rules ***R*** have the form $\forall X$ *[antecedent* $\rightarrow \exists Y$ *consequent]*, where *antecedent* and *consequent* are both conjunctions of state conditions expressed in first-order logic, state transforming events and temporal constraints. Reactive rules generalise condition-action rules in production systems, plans in BDI agents, and triggers in active databases.

Here is a simple example, where the variables T_1, T_2 and T_3 represent time, *tidy(Loc,* T_2, T_3*)* represents a possibly complex (or composite) action performed from time T_2 to time T_3 and *observe-trash-at(Loc,*T_1*,* T_1*+1)* represents an external event that triggers the action:

$$\forall Loc\ T_1\ [observe\text{-}trash\text{-}at(Loc,T_1, T_1+1) \rightarrow \exists\ T_2\ T_3\ [tidy(Loc, T_2, T_3) \wedge T_1 < T_2]]$$

In general, events include both external events and an agent's own internally generated actions. Events are represented by atomic formulas including time and possibly other parameters. Events that actually occur are represented by atomic sentences, in which all parameters are instantiated to ground terms. Several events can occur simultaneously, and together contribute to a single state transition. For simplicity, we use the terms *event* and *action* to refer either to an atomic formula of type event or action, or the occurrence of an event or action, whenever the meaning is clear from the context.

For simplicity, time is represented by the ticks of a logical clock, where *T+1* stands for *s(T*) and 1, 2, ... stand for *s(0), s(s(0)), …etc.,* as for example in[22, 34, 37]. Other, more elaborate representations of time are also possible. Moreover, time arguments can be hidden in an alternative syntax, as for example in[22, 32].

The logic program ***L*** represents an agent's view of states and events. It includes definitions of intensional predicates, composite events, and auxiliary predicates, which do not change over time. The combination of the extensional facts in a state S_i and the intensional predicate definitions in ***L*** includes Datalog as a special case. The combination of the simple state transforming events and the composite event definitions in $\boldsymbol{L}$ includes the functionalities of transaction logic[6)] and Golog[35)].

The clause below defines the composite action *tidy(Loc, T_2, T_3)* in terms of the simple, one step action *pickup-trash(Loc, T_2, T_2+1)* and the composite actions *goto(Loc, T_1, T_2)* and *getrid-trash(T_2+1, T_3)*:

$$\begin{aligned} &tidy(Loc, T_1, T_3) \leftarrow goto(Loc, T_1, T_2) \\ &\qquad \wedge pickup\text{-}trash(Loc, T_2, T_2+1) \wedge getrid\text{-}trash(T_2+1, T_3) \end{aligned}$$

As is usual in logic programming, variables that are not explicitly quantified are implicitly quantified with scope the clause in which they occur. The domain theory $\boldsymbol{D} = D_{pre} \cup D_{post}$, which also has the form of a logic program, defines the preconditions D_{pre} and postconditions D_{post} of state transitions, in a manner similar to that of the situation calculus[40] and event calculus[33)]. D_{post} defines only the effect of simple (or atomic) events on extensional predicates. For example, the following clauses define the postconditions of the simple event *observe-trash-at(Loc, T, T+1)* and the simple action *pickup-trash(Loc, T, T+1)*:

$$\begin{aligned} &initiated(trash\text{-}at(Loc), T+1) \leftarrow observe\text{-}trash\text{-}at(Loc, T, T+1) \\ &terminated(trash\text{-}at(Loc), T+1) \leftarrow pickup\text{-}trash(Loc, T, T+1) \end{aligned}$$

Intensional predicates are updated implicitly as ramifications of changes to the extensional predicates. For example:

$$all\text{-}clean(T) \leftarrow \neg \exists Loc\ trash\text{-}at(Loc, T)$$

FOL state conditions, such as the one in this definition, can occur both in reactive rules and in the bodies of logic programs.

2.2 The Operational Semantics

The operational semantics combines the features of a logic programming language, like Prolog and a database language, like Datalog, with the reactive rules of production systems, and “practical” BDI agent languages, such as AgentSpeak[44)], using a destructively updated database.

Informally speaking, the purpose of a framework **<*R*, *L*, *D*>**, given an initial state S_0 is to execute, for every set ext_i of external events, where $i \geq 1$, a set $acts_{i+1}$ of actions such that the reactive rules $\boldsymbol{R}$ are all true in a canonical model of the logic program determined by $\boldsymbol{L}$ and the resulting sequence of timestamped states and events.

The canonical model defined in this paper is for logic programs containing non-atomic FOL conditions**.** We extend the definition of local stratification[43)] to allow for such conditions, and call logic programs satisfying this extended definition *FOL-stratified*. We call the corresponding extension of the perfect model[43)], the *FOL-perfect model*. As in the case of perfect models, FOL-perfect models, if they exist, are unique and two-valued.

With the aim of making the reactive rules true, the operational semantics monitors the stream of states and events, to determine whether an instance *antecedent σ* of the antecedent of a reactive rule becomes true. If it does, it uses forward reasoning to generate the corresponding *consequent σ* of the rule as a goal to be achieved in the future. It uses backward reasoning to reduce such achievement goals to alternative plans, each of which is a conjunction of FOL state conditions, external events, actions and temporal constraints. It evaluates state conditions and external events, and it selects candidate actions for attempted execution. The operational semantics uses D_{pre} to ensure that only compatible sets of concurrent events actually occur.

In the operational semantics, fluents are represented without explicit time. The state transition from S_{i-1} to S_i determined by a set ev_i of concurrent event occurrences (including both external events ext_i and successfully executed actions $acts_i$) is performed destructively, using the domain theory D_{post} to delete fluents in S_{i-1} terminated by ev_i and to add fluents to S_i initiated by ev_i. Fluents that are neither initiated nor terminated simply persist from S_{i-1} to S_i without reasoning that they persist, and without copying them explicitly from S_{i-1} to S_i. Similarly, the operational semantics does not store the entire history of past events, but only the most recent set of events ev_i.

2.3 The model-theoretic semantics

In modal temporal logics[39], states are represented by sets of facts without timestamps in separate possible worlds, linked by an accessibility relation associated with state transforming events. As a consequence, events and times are not first class objects in the language. In contrast in LPS, events and times are represented explicitly in the language. In the model-theoretic semantics, fluents and events are timestamped and combined in a single model-theoretic structure.

In the case of a fluent p with timestamp t, we reify p and write *holds(p, t)*, which we also abbreviate as *p(t)*. To distinguish between a state S_i in which fluents are all without timestamps, and the same state in which all the fluents have the same timestamp t_i, we write S_i*.

In the model-theoretic semantics, a simple event e taking place between states S_{i-1} at time t_{i-1} and S_i at time t_i is written as *happens*(e, t_{i-1}, t_i), abbreviated as $e(t_{i-1}, t_i)$. For the concurrent occurrence of an unstamped set ev_i of events, occurring between t_{i-1} and t_i we write ev_i* for the same set with their timestamps.

Whereas, in the operational semantics, state transitions are performed destructively, in the model-theoretic semantics, they are specified by an event theory *ET*, which is a hybrid of the situation calculus and the event calculus:

Definition 2.1
The event theory *ET* consists of the two clauses:

$$holds(P, T) \leftarrow initiated(P, T)$$

$$holds(P, T+1) \leftarrow holds(P, T) \wedge \neg\, terminated(P, T+1)$$

The second clause in *ET* is a frame axiom in the spirit of the situation calculus. However, the ontology of *ET* in terms of events and time points is inspired by the event calculus. *ET* is both locally stratified[43] and FOL-stratified.

The predicates *initiated* and *terminated* are defined by the domain theory ***D***. For simplicity, we assume that all fluents in S_i hold at time *i*, and that all events in ev_i occur from time *i-1* to *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.

Definition 2.2
Given an LPS framework **<*R, L, D*>** and an initial state S_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:

$\boldsymbol{R} \cup D_{pre}$ is true in the FOL-perfect model of $ET \cup D_{post} \cup L \cup S_0^* \cup ev_*$
where $ev_* = ev_1^* \cup ev_2^* \cup \ldots \cup ev_i^* \cup \ldots,$
$ev_i = ext_i \cup acts_i$, for $i \geq 1$, and $act_1 = \{\}$.

The notion of FOL-perfect model is defined in section 6.

In definition 2.2, the computational task is shared between an agent attempting to execute a collection of candidate actions ($candidate\text{-}acts_i$) to make ***R*** true and the environment, maintaining D_{pre}, by arbitrating between conflicting sets of candidate actions. The result of this arbitration is a set $ev_i = ext_i \cup acts_i$ of events, where $acts_i$ is the subset of $candidate\text{-}acts_i$ that have succeeded, and ext_i is the set of all other successful events.

Given an initial state S_0^* and sequence of state transforming events ev_*, the event theory *ET* together with $D_{post} \cup L$ defines the *holds* predicate for all subsequent states $S_1^*, \ldots, S_i^*, \ldots$. It is possible to use this definition as a logic program, top-down or bottom-up, to compute the *holds* predicate, but this is not computationally feasible in most practical cases. It is not feasible, when states contain many fluents, to reason bottom-up with frame axioms, duplicating facts that hold from one state to the next. Nor is it feasible, when there are many state transitions, to reason top-down, determining whether a fact holds in a given state by checking recursively whether it held in the previous state. As a consequence, frame axioms are rarely used in practical applications, and destructive assignment or destructive updates are generally used instead.

The computational inefficiencies of reasoning with frame axioms has received little attention. For example, Shanahan[49] on page 7, explicitly excludes consideration of "implementation issues". In contrast, we consider frame axioms to be one of the main reasons why purely declarative languages have not been able to compete effectively with imperative programming languages and database systems. One of the main goals of this paper, therefore, is to show that destructive change of state can be given a logical semantics, by using destructive updates to construct a model in which the reactive rules are all true.

We will define LPS and its operational semantics more precisely later in the paper. But first, we illustrate LPS with two examples.

§3 Examples

Example 3.1

In this example, an online book store uses a database to keep track of its inventory, and allocates books in response to customer requests. Several books can be requested at the same time, and several allocations can also be made at the same time, provided the constraints specified by the preconditions are satisfied. One constraint prevents the allocation of a copy of a book if there are no copies available. A second constraint ensures that two customers are not allocated a copy of the same book at the same time.

The system chooses at every time point an allocation of books that satisfies the constraints. A request for an allocation that is not satisfied at one time may be satisfied at a later time, provided the amount of waiting time (say 1000 units of time, equivalent perhaps to 1 second) is not too long.

To assist in keeping track of requests, the database contains information about requests that are currently pending. At the top-most level, the program consists of a single reactive rule and two logic programming clauses:

$\boldsymbol{R}$: $request(Customer, Item, T, T+1)$
$\quad\quad \rightarrow respond(Customer, Item, T, T_1, T_2) \wedge T < T_1$

L_{events}: $respond(Customer, Item, T, T_1, T_2) \leftarrow pending(Customer, Item, T_1)$
$\quad\quad \wedge T_1 < T+1000 \wedge allocate(Customer, Item, T_1, T_1+1)$
$\quad\quad \wedge process\text{-}order(Customer, Item, T_1+1, T_2)$

$respond(Customer, Item, T, T_1, T_1+1) \leftarrow pending(Customer, Item, T_1)$
$\quad\quad \wedge T_1 = T+1000 \wedge apologise(Customer, Item, T_1, T_1+1)$

The forward arrow $\rightarrow$ is used for logical implication in reactive rules, and the backward arrow $\leftarrow$ is used 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. Variables beginning with upper case T represent time points.

The variables *Customer, Item* and T in $\boldsymbol{R}$ are universally quantified with scope the entire rule $\boldsymbol{R}$, but T_1 and T_2 are existentially quantified with scope the consequent of the rule. In general, as we will see in section 4, the quantification of all variables that are not bound explicitly in a rule can be left implicit. Here:

- *request(Customer, Item, T, T+1)* represents a simple external event that takes place in one state transition from time *T* to *T+1*;
- *allocate(Customer, Item,* T_1*,* T_1*+1)* and *apologise(Customer, Item,* T_1*,* T_1*+1)* represent simple actions from T_1 to T_1+1;
- *respond(Customer, Item, T,* T_1*,* T_2 *)* is a composite action from T_1 to T_2, defined in L_{events}, and *T* is a reference time used to limit the delay in responding to a request;
- *process-order(Customer, Item,* T_1*+1,* T_2*)* is either a composite action defined in L_{events}, but not shown here, or a simple action (in which case $T_2 = T_1+2$) that sends a message to the department in the company responsible for processing orders. In either case we assume that *process-order* always succeeds.

There are two types of fluents, *available(Item, Number)* and *pending(Customer, Item),* which are initiated and terminated by simple events. For simplicity and to conserve space, the relationships between events and fluents are represented without their time arguments in the following table:

event/action	initiated fluents	terminated fluents
request(Customer, Item)	*pending(Customer, Item)*	-

allocate(Customer, Item)	*available(Item, New)*	*available(Item, Old)*
		pending(Customer, Item)

apologise(Customer, Item)	-	*pending(Customer, Item)*

For example, the property that the action *allocate(Customer, Item)* initiates the fluent *available(Item, New)* can be represented by the clause in D_{post}:

$$\begin{aligned}&initiated(available(Item, New), T+1) \leftarrow allocate(Customer, Item, T, T+1)\\&\qquad \wedge available(Item, Old, T) \wedge New = Old - 1\end{aligned}$$

The two precondition constraints are represented in D_{pre}:

$$\begin{aligned}&false \leftarrow allocate(Customer, Item, T, T+1) \wedge available(Item, 0, T)\\&false \leftarrow allocate(Customer_1, Item, T, T+1) \wedge allocate(Customer_2, Item, T, T+1)\\&\qquad \wedge Customer_1 \neq Customer_2\end{aligned}$$

Given S_0 = {*available(hamlet, 6), available(emma, 2)*}, here is one possible execution sequence, in which events and fluent are represented without their time arguments, and *process-order(Customer, Item)* is assumed to be a simple action:

ext_1 = {*request(bob, hamlet), request(bob, emma), request(mary, emma)*}
S_1 = {*available(hamlet, 6), available(emma, 2),*
pending(bob, hamlet), pending(bob, emma), pending(mary, emma)}

candidate-acts$_2$ =*{allocate(bob,hamlet),allocate(bob, emma),allocate(mary, emma)}*
ev$_2$ = *{allocate(bob, hamlet), allocate(mary, emma), request(john, emma)}*
S$_2$ = *{available(hamlet, 5), available(emma, 1),*
pending(bob, emma), pending(john, emma)}
candidate-acts$_3$ = *{process-order(bob, hamlet), process-order(mary, emma),*
allocate(bob, emma), allocate(john, emma)}

Because of the second clause in D_{pre}, the actions *allocate(bob, emma)* and *allocate(john, emma)* cannot both be executed at the same time. Let us suppose *allocate(john, emma)* is chosen for execution.

ev$_3$ = *{process-order(bob, hamlet), process-order(mary, emma),*
allocate(john, emma)}
S$_3$ = *{available(hamlet, 5), available(emma, 0), pending(bob, emma)}*
candidate-acts$_4$ = *{process-order(john, emma), allocate(bob, emma)}*
ev$_4$ = *{process-order(john, emma)}*

All requests have now been allocated and processed except for *bob's* request for *emma*. The action *allocate(bob, emma)* cannot be executed because of the first clause in D_{pre}.

$S_4 = \ldots = S_{1000} = S_3$
candidate-acts$_5$ = …= *candidate-acts*$_{1000}$ = *{allocate(bob, emma)}*
ev$_5$ =….= *ev*$_{1000}$ = {}

The action *allocate(bob, emma)* is now timed out. So the operational semantics tries the second clause for solving the goal *respond(bob, emma, 1,* T_1, T_1+1*)*. This generates the action *apologise(bob, emma)*:

candidate-acts$_{1001}$ =*{apologise(bob, emma)}*
ev$_{1001}$ =*{apologise(bob, emma)}*
S$_{1001}$ = *{available(hamlet, 5), available(emma, 0)}*

The operational semantics also maintains a separate goal state, which is logically a conjunction of disjunctions of conjunctions of goals to be made true in the future. For example, the goal state G_1 has three top level conjuncts:

(respond(bob, hamlet, 0, T_{11}, T_{12}*)* $\wedge\ 0 < T_{11}$*)* $\wedge$
(respond(bob, emma, 0, T_{21}, T_{22}*)* $\wedge\ 0 < T_{21}$*)* $\wedge$
(respond(mary, emma, 0, T_{31}, T_{32}*)* $\wedge\ 0 < T_{31}$*)*

The goal states $G_4 = \ldots = G_{1000}$ all have the same single conjunct:

(pending(bob, emma, T_1*)* $\wedge\ T_1 < 1000\ \wedge$ *allocate(bob, emma,* T_1, T_1+1*)*
$\wedge$ *process-order(bob, emma,* T_1+1, T_2*))* $\vee$

$(pending(bob, emma, T_1) \wedge T_1 = 1000 \wedge apologise(bob, emma, T_1, T_1{+}1))$

The goal state G_{1001} is logically equivalent to *true*.

Each top-level conjunct in a goal state is, in effect, a separate thread, each of which represents a search space of alternative ways of solving a top-level goal. This search space is like the search space for SLD-resolution[27]. Different search strategies can be used to explore these threads. In particular there is no need to generate and store the whole search space. In several of our prototype implementations, described in greater detail in Section 5.4, we have used a Prolog-like depth-first search strategy to explore branches of the search space, and have used a Prolog-like stack to represent conjunctions of subgoals, ordered by their time parameters.

Notice that $\boldsymbol{R} \cup D_{pre}$ is true in the perfect model of the logic program:

$$S_0^* \cup S_1^* \cup \ldots \cup S_{1001}^* \cup ev_1^* \cup \ldots \cup ev_{1001}^* \cup \boldsymbol{L}$$

where $\boldsymbol{L}$ contains L_{events} and definitions of = and <.

In this example, there are many alternative actions that can be chosen in the attempt to make $\boldsymbol{R} \cup D_{pre}$ true. The framework specifies only the logic of the problem, but not the control strategy needed to obtain an efficient and fair algorithm. In several of our prototype implementations, the programmer controls which candidate actions are tried, simply by ordering clauses in $\boldsymbol{L}$ (as in Prolog). In addition, the programmer can specify whether different reactive rules should have different priorities, or whether candidate actions should succeed with a first-come-first-served strategy.

LPS, like production systems, BDI agents and database triggers in general, is incomplete with respect to the model-theoretic semantics of reactive rules, because it can generate only "supported" models that make the consequents of reactive rules true when their antecedents become true. It cannot arbitrarily perform an action that has no relation to its goals, cannot preventatively make a reactive rule true by making its antecedent false, and cannot proactively make its consequent true in anticipation of its antecedent becoming true in the future. Although this kind of incompleteness is a limitation for an AI system, it is desirable for practical programming and database systems, because it greatly contributes to their efficiency.

It is not easy to illustrate all the major features of LPS in a single, simple example. In particular, this example does not illustrate the role of FOL-state conditions, intensional fluents or composite event recognition. However, the example could easily be extended, for example with such intensional fluent definitions such as:

$$level(Item, low, T) \leftarrow available(Item, N, T) \wedge N < 2$$

The benefit of such definitions is that the intensional predicate (*level* in this case) changes its value automatically as a ramification of changes to the extensional predicates (*available*).

The next example illustrates complex event recognition, as well as the ability of the operational semantics to switch between alternative plans.

Example 3.2

This example is a variant of an example in Hausmann et al[21] in which 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 plans. 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 representation is an LPS framework <***R***, ***L***, ***D***> consisting of a main program ***R***, which contains a single reactive rule, and a logic program ***L***. However, the domain theory ***D*** and all states are all empty. The expression *n sec* is an abbreviation for some appropriate number of clock ticks, e.g. $1000 \times n$.

R: $\quad heat\text{-}sensed(Area,\ Tf,\ Tf{+}1) \wedge smoke\text{-}detected(Area,\ Ts,\ Ts{+}1)$
$\wedge\ |Tf - Ts| \leq 60\ sec\ \wedge max(Tf,\ Ts,\ T)$
$\rightarrow fire\text{-}response(Area,\ T,\ T_1,\ T_2) \wedge T < T_1$

L_{events}: $fire\text{-}response(Area,\ T,\ T_1,\ T_2{+}1) \leftarrow activate\text{-}fire\text{-}suppression(Area,\ T_1,\ T_1{+}1)$
$\wedge\ T_1 \leq T + 5\ sec \wedge send\text{-}security\text{-}guard(Guard,\ Area,\ T_2,\ T_2{+}1)$
$\wedge\ T_1 < T_2 \leq T_1 + 10\ sec$

$fire\text{-}response(Area,\ T,\ T_1,\ T_1{+}1)$
$\leftarrow call\text{-}fire\text{-}department(Area,\ T_1,\ T_1{+}1) \wedge\ T_1 \leq T + 120\ sec$

Here *heat-sensed(Area, Tf, Tf+1)* ∧ *smoke-detected(Area, Ts, Ts+1)* represent external events, each of which takes place during one state transition. For simplicity, the actions *activate-fire-suppression(Area,* T_1*,* T_1*+1), send-security-guard(Guard, Area,* T_2*,* T_2*+1)* and *call-fire-department(Area,* T_1*,* T_1*+1)* are all treated as simple actions, which also take place during one state transition. There are no fluents in this example.

The antecedent of ***R*** represents an unnamed composite event and the consequent represents the named composite action *fire-response(Area, T,* T_1*,* T_2*)*. This composite action consists of two alternative plans, each of which is represented by a clause in L_{events}. Both plans are temporally constrained.

The framework specifies only the logic of the problem, but not the control strategy. In practice, it might be a good strategy to try the first plan first. In an implementation, this can be indicated by the order in which the rules are written, as in Prolog. Also as in Prolog, 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. Notice that the temporal constraints ensure that, if the first plan takes too long, then the second plan can still be tried.

When a plan is attempted but fails, its partial execution will typically have caused changes to the current state. Even if it has not caused any changes, state changes may take place because external events have occurred. Thus, if the plan is re-tried later or if an alternative plan is attempted instead, it is in the context of the new changed state. Indeed, this is why previously failed actions remain in the goal state, because, if the temporal constraints allow it, they can be retried later and may succeed in the new state.

The operational semantics maintains only the current state and only the events that gave rise to the current state. So when an event occurs, for example *smoke-detected(kitchen, 15, 16),* a partially executed rule is derived:

$$\begin{aligned}&\textit{heat-sensed(kitchen, } Tf, Tf{+}1) \wedge |Tf - 15| \leq 60\ sec \wedge max(Tf, 15, T)\\ &\quad \rightarrow \textit{fire-response(kitchen, } T, T_1, T_2) \wedge T < T_1\end{aligned}$$

If, for example a later event *heat-sensed(kitchen, 26, 27)*, matches a condition of the derived rule, and its time of occurrence satisfies the temporal constraints, then a new goal is added to the goal state:

$$\textit{fire-response(kitchen, } 26, T_1, T_2) \wedge 26 < T_1$$

In this way, it is possible to recognize a complex event consisting of several simple events occurring over a period of time without the need to store the entire history of simple events. The simple events can be processed as streams, and need not be stored for longer than a single state transition.

§4 The Language

4.1 Vocabulary

We assume a sorted language in which constants and variables are assigned sorts. 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.

Predicate symbols 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.***
- *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.***
- *Auxiliary predicates* consist of predicates that do not vary with time, such as *max* and *min* and others used for arithmetic*,* all of which are defined in ***L***.
- The *meta-predicates* consist of the predicates *initiated* and *terminated* defined in ***D***, 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 *i.* Similarly, the sets ev_i of events are not represented explicitly, but are represented implicitly by the set of simple events that occur from time *i-1* to *i.*

4.2 Reactive Rules

Reactive rules (or simply *rules*) in ***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*, rules can contain other bound variables in FOL state conditions. Because of these restrictions on the quantification of variables, we can omit the quantifiers $\forall X$ and $\exists Y$. More formally:

Definition 4.1
A reactive rule is a sentence of the form *antecedent(X)→ consequent(X, Y),* where both *antecedent(X)* and *consequent(X, Y)* are a conjunction of conditions each of which is either a state condition, event atom or temporal constraint.

- A *state condition* is a formula of first-order logic (FOL) in the vocabulary of the fluent and auxiliary 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, where the time parameter refers to the current time.
- 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.
- A temporal constraint is an atomic formula of the form $t_1 < t_2$ or $t_1 \leq t_2$ where t_1 and t_2 are terms represent time points.

The only variables that occur in temporal constraints must also 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 in FOL state conditions, events can occur only as atomic conjuncts. This is because events are processed as streams, and are stored for only a single state transition. This makes it hard, but not impossible for the operational semantics to check, for example, the condition that no event of a certain kind occurs within a certain interval of time. This restriction on the syntax of reactive rules is not necessary for the model-theoretic semantics, which requires only that the reactive rules are true, as determined by the standard definition of truth for sentences of FOL. The restriction can be removed at the expense of complicating the operational semantics. However, discussion of this topic is beyond the scope of this paper.

4.3 Goal clauses

In the model-theoretic 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. For this purpose, the operational semantics maintains a goal state containing goal clauses:

Definition 4.2
A goal clause is an existentially quantified conjunction of state conditions, event atoms and temporal constraints. All variables in the temporal constraints occur in the state conditions and event atoms of the goal clause.

Note that the effect of having an initial goal clause C_0 can be obtained by having instead a rule $start(0, 1) \rightarrow C_0$, where *start* is a simple event occurring from time *0* to *1*.

4.4 Logic programs

The logic program ***L*** of an LPS framework **<*R*, *L*, *D*>** can contain non-atomic FOL conditions, as in the extended logic programs of Lloyd and Topor[29]. Here we refer to "extended logic programs" simply as "logic programs":

Definition 4.3
A logic program is a set of *clauses* of the form $head(X) \leftarrow body(X, Y)$, where X is the set of all variables that occur in $head(X)$, and Y is set of all unbound variables that occur only in $body(X, Y)$. $head(X)$ is an atomic formula, and $body(X, Y)$ is a (possibly empty) conjunction of *conditions,* which are atomic and non-atomic FOL formulas.[1] An extended logic program whose bodies are (possibly empty) conjunctions of atomic formulas is a Horn clause program.

Clauses are implicitly quantified in one of the two equivalent forms:

[1] Although a conjunction of FOL formulas is itself an FOL formula, it is useful in LPS to distinguish between atomic and non-atomic FOL formulas,

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

Definition 4.4
The logic programming component ***L*** of an LPS framework is partitioned into three components: $\boldsymbol{L} = L_{int} \cup L_{events} \cup L_{aux}$.

- L_{int} consists of clauses of the form $head(X, T) \leftarrow body(X, Y, T)$ in which the predicate of $head(X, T)$ is an intensional predicate, and the predicates in the *body* are intensional, extensional or time-independent. Each clause in L_{int} contains exactly one time parameter T that is a variable.
- L_{events} consists of clauses of the form $head(X, T_1, T_2) \leftarrow body(X, Y, T_1, T_2)$ in which the predicate of $head(X, T_1, T_2)$ is a composite event predicate, and $body(X, Y, T_1, T_2)$ is a conjunction of FOL state conditions, event atoms and temporal constraints. 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 $body(X, Y, T_1, T_2)$. (X might include other time parameters, as in the examples in section 3). The time variables in temporal constraints must all occur in the *head* or unbound in fluent or event atoms in the *body*.
- L_{aux} defines auxiliary predicates, such as $\leq$, $<$, *max* and *min*, which do not change with time.

Notice that the body of a clause in L_{events} is similar in form both to a goal clause, and to the antecedent or consequent of a reactive rule.

4.5 The Domain Theory *D*

The domain theory $\boldsymbol{D} = D_{post} \cup D_{pre}$ has two components. D_{post} is a logic program that specifies the extensional fluents that are initiated and terminated by simple events. D_{pre} is a set of integrity constraints restricting the occurrence and co-occurrence of sets of simple events.

Definition 4.5
D_{post} is a set of clauses of the form $head(T+1) \leftarrow body(T, T+1)$.
D_{pre} is a set of integrity constraints of the form $false \leftarrow body(T, T+1)$.
$head(T+1)$ is an atom of the form $initiated(P, T+1)$ or $terminated(P, T+1)$, where P is an extensional fluent.
$body(T, T+1)$ is a conjunction of simple event predicates of the form $happens(e, T, T+1)$ and FOL state conditions with time parameter T.

In the operational semantics, $body(T, T+1)$ is a query to the augmented current state $S_i^* \cup L_{int} \cup L_{aux} \cup ev_i^*$ at time i. An answer to the query is a ground instantiation of the free variables in $body(T, T+1)$, with T instantiated to i, and with bound variables treated according to the classical semantics of universal and existential quantifiers.

4.6 The Environment

An LPS framework ***<R, L, D>*** represents the goals ***R*** and beliefs ***L*** of an individual agent embedded in an environment, which includes both a shared global state, as well as the agent's own, encapsulated local state. In a multi-agent system, the global component of the state is shared among a collection of agents, as in the Linda coordination language paradigm[9].

Given a current state S_{i-1} and candidate actions submitted by different agents, the environment merges them into a combined set ev_i of concurrent events satisfying D_{pre}, arbitrating between conflicting candidates. Similarly, the environment updates the state S_{i-1} to S_i, using D_{post}.

Computation is defined as performing actions to make the goals R of an individual agent *true*, while ensuring that the constraints D_{pre} are not violated by becoming *false*. 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.

§5 The Operational Semantics

The operational semantics (OS) can be thought of as a potentially non-terminating cycle, in which external events and the agent's own successfully executed 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 attempt to execute actions.

The OS is compatible with many different implementations. In particular, although it is defined for programs written with an explicit representation of time, it can also be implemented, as in Kowalski and Sadri[31], 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 only semi-constructive. Extended states can contain a countably infinite number of ground atoms, and an FOL state condition, which queries the extended state, can have a countably infinite number of answers. In practice, these infinities can be avoided, for example by avoiding function symbols, as in Datalog.

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 goal clause at the root of the tree. This top-level goal clause is 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

- 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, with root 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, 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*.

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 in advance, using L_{events} to reduce composite events to conjunctions of simple events, state conditions and temporal constraints. 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. Alternatively, backward reasoning could be used at "run time" to reduce composite event predicates to simpler event predicates. 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. It is the generation of such new rules that makes it possible to forget the history of past events. Conceptually, the rules in R_i and the goal clauses in G_i are just different kinds of goals. However, in the OS, it is useful to treat them separately.

5.2 Restricting the Amount of Computation within a Cycle

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, perhaps by restricting the language, to ensure that these steps can be performed in a timely manner, before the next time in the succession of time points.

On the other hand, it is necessary to restrict the amount of time that is spent on goal reduction in step 2. This can be done in 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.

5.3 The OS Cycle

Initially $R_0 = \boldsymbol{R}$, $act_1 = \{\}$; and $G_0 = \{\}$. The *i-th* iteration of the cycle, for $i > 0$, consists of the following steps:

Step 0. Update the current state. Select a set of concurrent events $ev_i = ext_i \cup acts_i$ such that D_{pre} is true in the FOL-perfect model of $S_{i-1}^* \cup L_{int} \cup L_{aux} \cup ev_i^*$, where ext_i is a set of external events and, for $i > 1$, $acts_i$ is a subset of the submitted candidate actions *candidate-acts*$_i$.

Transform state S_{i-1} into S_i, by deleting any fluents p such that *terminated(p, i)* is true in the FOL-perfect model of $D_{post} \cup S_{i-1}^* \cup L_{int} \cup L_{aux} \cup ev_i^*$ and adding any fluents p such that *initiated(p, i)* is true in the FOL-perfect model of $D_{post} \cup S_{i-1}^* \cup L_{int} \cup L_{aux} \cup ev_i^*$.

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-antecedents $\wedge$ *other-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 *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 *i*.

For each such parsing and each ground instance *early-antecedents* σ that is true in the FOL-perfect model of $S_i^* \cup L_{int} \cup L_{aux} \cup ev_i^*$, 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 the FOL-perfect model of L_{aux}. 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 time of the next cycle has been reached or there are no new steps that can be performed in this iteration of the cycle, then this iteration of the cycle terminates. Otherwise, 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 *E* in *C,* unify *E* 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 modest 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 *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 *i.*

If there is a ground instance *early-consequents σ* that is *true* in the FOL-perfect model of $S_i^* \cup L_{int} \cup L_{aux} \cup ev_i^*$, 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, T+1)* such that all the time parameters *T* and *T+1* can be unified with the times *i* and *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 *i.*

Add all of the simple actions *happens(a, i, i+1)* to $candidate\text{-}acts_{i+1}$. Any candidate action *happens(a, i, i+1)* that is successfully executed in step 0 of the next iteration of the cycle is then resolved upon in step 2.2 of that iteration.

Notes:

1. Steps 1 and 2 of the OS are the operational semantics of a single agent, possibly interacting with other agents. 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.

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*$(X, i, i+1)$ 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.

3. In steps 1, 2.2 and 2.3, different parsings amount to different ways of sequencing state conditions and simple events. 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 $p(T_1) \wedge r(T_3)$

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 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 can 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 E in C with the head of a clause in L_{events} that has not been tried before. In step 2.2 it can retry the same parsing *early-consequents* of conditions and simple event atoms, because the augmented current state $S_i^* \cup L_{int} \cup L_{aux} \cup ev_i^*$ may have changed. For similar reasons, in step 2.3 it can retry the same conjunction of actions, because actions that were not possible before may become possible in the new current state.

5.4 Implementation

The OS is compatible with many different implementations. We have developed several implementations in Prolog and Java, and have tested them on a variety of examples, including the blocks world, the dining philosophers, a traffic norms example[2)] and a tool hire company example. In these implementations, we have explored different strategies for goal reduction in steps 2.1 and 2.2, and for action selection in step 2.3 of the OS.

In all of the Prolog implementations, the goal state is represented as a list of goal trees (or threads). In most of these implementations, each goal tree (or thread) is also represented as a list, representing a current, single branch of the goal tree, which is searched in a depth-first manner, as in Prolog.

In each cycle, goal reduction in step 2 re-commences from the thread at the beginning of the list representing the goal state. The number of goal reductions in a cycle is restricted by setting a maximum number *N* that can be performed in each cycle. During each cycle, as many threads of the goal state are explored as possible, extending each branch by a proportion of *N* or until the branch ends with a goal clause starting with a simple action. The set of all such actions at the ends of branches makes up the set of *candidate-acts* at the end of the cycle.

New top-level goals, starting new threads, are added to the goal state in step 1 on the cycle. We have explored several strategies for prioritizing these goals. Adding the new threads to the front of the list representing the goal state gives priority to the new threads, processing them *last-in-first-out*. Adding them to the end of the list processes them *first-in-first-out*.

We have also explored the alternative strategy of giving the user the ability to assign priorities to the reactive rules. These priorities are then inherited by the new threads that are generated when the antecedents of the rules become true. For example, the second of the following two rules is assigned higher priority than the first:

$get\text{-}hungry(T, T+1) \rightarrow eat(T_1, T_2) \wedge T < T_1$	Priority 1
$attacked(T, T+1) \rightarrow run\text{-}away(T_1, T_2) \wedge T < T_1$	Priority 100

When a new thread is added to the list representing the goal state in step 1 of the cycle, the list is re-ordered, with higher priority threads positioned earlier in the list and lower priority threads positioned later

We have also investigated prioritizing goals in order of their deadlines, which is the latest time at which they must start. For example, given the rules:

$get\text{-}hungry(T, T+1) \rightarrow eat(T_1, T_2) \wedge T < T_1 \wedge T_1 < T+10$
$attacked(T, T+1) \rightarrow run\text{-}away(T_1, T_2) \wedge T < T_1 \wedge T_1 < T+3$

and observations:

$get\text{-}hungry(0, 1)$
$attacked(0, 1)$

we get two top-level goal clauses, starting new threads:

$eat(T_1, T_2) \wedge 0 < T_1 \wedge T_1 < 10$
$run\text{-}away(T_1, T_2) \wedge 0 < T_1 \wedge T_1 < 3$

Using a Prolog solver for the temporal constraints, the goal clauses and the conjuncts in goal clauses are sorted according to their deadlines, from the earliest to the latest. Since the action *run-away* has an earlier deadline, the goal clauses will be ordered with $run\text{-}away(T_1, T_2) \wedge 0 < T_1 \wedge T_1 < 3$ given higher priority than $eat(T_1, T_2) \wedge 0 < T_1 \wedge T_1 < 10$. Assuming that the two actions cannot be executed concurrently (an assumption that can be represented in D_{pre}), the action $run\text{-}away(T_1, T_2)$ will be attempted before the action $eat(T_1, T_2)$.

In addition, we have also implemented a prioritization strategy in which the programmer specifies the maximum number of cycles that a goal clause can wait before it is processed. When a new thread is added to a goal state, it is stamped with the current cycle index. This cycle index is then used to recognise when a thread has reached the maximum time it is allowed to wait, at which time if it has not already been processed, it is promoted to a position of highest priority in the list representing the goal state.

Perhaps the hardest decision that the OS needs to make is whether and for how long to retry a simple action or to evaluate a state condition, when they fail. In some cases, there may be other alternatives that can be tried. In any case, a decision needs to be made whether or not to retry the failed action or failed state condition in the future, when it may succeed in a new state.

In one of the strategies we have implemented, the programmer can specify a maximum number of cycles that a simple action or state condition can be retried. If this maximum number of retries has been reached and the action or state condition has not succeeded, then the OS needs to try an alternative.

We have also implemented a breadth-first strategy for non-recursive LPS programs. In this implementation in steps 2.1 and 2.2 of the OS, goal-reduction, using the clauses in L_{events}, L_{int} and L_{aux}, is performed in all possible ways. This simplifies the resulting goal state, which then consists of a list of threads, each of which consists of a top-level goal whose immediate successors are a disjunction (represented by a list), each of whose disjuncts is a conjunction (represented by a list) of only simple actions, state conditions involving only extensional predicates and temporal constraints. In addition, we have combined different strategies, including this breadth-first strategy, with the different strategies described earlier for prioritising goals.

Our prototype implementations confirm that the OS and the model-theoretic semantics of LPS can be realised in a variety of different ways. Determining the best strategies, both in terms of expressive power and in terms of efficiency is ongoing work.

§6 Soundness and the Frame Theorem

In this section, we define FOL-stratification and FOL-perfect model. Lloyd and Topor[36] reduce logic programs with FOL conditions in their bodies to normal logic programs whose bodies are conjunctions of *literals*, namely atomic formulas and negations of atomic formulas. This reduction involves the introduction of new, auxiliary predicates. In contrast, in the operational semantics of LPS we evaluate FOL conditions directly using the standard definition of truth for sentences of first-order logic. For this purpose, we generalise the treatment of negative literals in the perfect model semantics of locally stratified logic programs to non-atomic FOL conditions. Because the resulting FOL-perfect models are two-valued, the standard two-valued Tarskian semantics applies.

6.1 FOL-stratification and FOL-perfect Model

As usual in logic programming, we treat a logic program P as standing for the set of all its instances over the Herbrand universe, which is the set of all well-sorted, variable-free (i.e. ground) atoms constructible from the vocabulary of P. 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 explicitly by quantifiers in non-atomic FOL conditions in $body(X, Y)$.

Definition 6.1

Let P be a ground logic program. Let $H = \cup_{0 \leq i \leq a} H_i$, where a is a countable, possibly transfinite ordinal, be a partitioning and ordering of the Herbrand base (i.e. the set of all well-sorted ground atoms over the Herbrand universe) 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 for every ground instance A of an atomic subformula of C $stratum(A) < stratum(head)$.

Any such FOL-stratification of H induces a corresponding FOL-stratification of $P = \cup_{0 \leq i \leq a} P_i$, where $P_i = \{head \leftarrow body \in P \mid stratum(head) = i\}$ defines the predicates in H_i.

Informally speaking, the FOL-perfect model of an FOL-stratified program P is constructed bottom-up, by repeatedly using the FOL-perfect model of all predicates defined at strata P_j, $j < i$ to evaluate the truth values of FOL-conditions in P_i. This construction can be carried out with the aid of a variant of the Gelfond and Lifschitz reduct[14].

Definition 6.2

Let P_1 be a set of ground atoms and $P = P_1 \cup P_2$ be an FOL-stratified program with respect to the stratification $H = H_1 \cup H_2$ of the Herbrand base H of P. Then $reduct(P, P_1)$ is the set of Horn clauses generated from P by deleting all FOL conditions in the bodies of clauses in P that are true in P_1 and deleting all clauses in P that have an FOL condition that is false in P_1. Note that P_1 is contained in $reduct(P, P_1)$.

Notice that this definition exploits the dual character of the set of ground atoms P_1 - both as clauses in P and as a Herbrand model of the predicates defined in P_1. Notice also that the definition of *reduct* could be extended to include the evaluation of conditions involving aggregate operators.

Definition 6.3
Let P be a ground Horn clause program, and let H be the Herbrand base of P. 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, if all the conditions in *body* are in M, then *head* is also in M.

Definition 6.4
Let P be a ground FOL-stratified logic program with respect to H_i, $0 \leq i \leq a$. The *FOL-perfect model* $perfect(P)$ of P is defined by:

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

6.2 The Soundness Theorem

Theorem 6.1
Given an LPS framework <***R***, ***L***, ***D***> and an initial state S_0, where ***L*** and ***D*** are FOL-stratified, suppose for every set ext_i of external events, where $i > 0$, the OS selects a set of concurrent events $ev_i = ext_i \cup acts_i$ at step 0 of the i^{th} iteration of the OS cycle.

Then $\boldsymbol{R} \cup D_{pre}$ is true in the FOL-perfect model of $ET \cup D_{post} \cup \boldsymbol{L} \cup S_0^* \cup ev_*$
where $ev_* = ev_1^* \cup ev_2^* \cup \ldots \cup ev_i^* \cup \ldots$,
$ev_i = ext_i \cup acts_i$, for $i \geq 1$, and $act_1 = \{\}$,

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.

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 possible in step 2.2, to ensure that any sub-goals that are true in the model generated so far are recognized and reduced to true.

6.3 The Frame Theorem

The proof of the soundness theorem makes use of the more general frame theorem, which states that destructive updates generate the same model as the event theory *ET* given in Definition 2.1:

$$holds(P, T) \leftarrow initiated(P, T)$$
$$holds(P, T+1) \leftarrow holds(P, T) \wedge \neg\, terminated(P, T+1)$$

Theorem 6.2
Let $\boldsymbol{L} = L_{int} \cup L_{aux}$ and D_{post} be FOL-stratified programs, $ev_i{}^*$ a set of ground atoms of the form *happens(e, i-1, i),* and S_0 a set of ground atoms of the form *holds(p, 0)* where *p* is an extensional fluent. Let

$$\begin{aligned} S_i =& (S_{i-1} - \{p \mid terminated(p, i) \in perfect(D_{post} \cup \boldsymbol{L} \cup S_{i-1}{}^* \cup ev_i{}^*)\}) \\ & \cup \{p \mid initiated(p, i) \in perfect(D_{post} \cup \boldsymbol{L} \cup S_{i-1}{}^* \cup ev_i{}^*)\} \end{aligned}$$

Then $perfect(D_{post} \cup \boldsymbol{L} \cup S^* \cup ev^*) = perfect(ET \cup D_{post} \cup \boldsymbol{L} \cup S_0{}^* \cup ev^*)$
where $S^* = S_0{}^* \cup S_1{}^* \cup \ldots \cup S_i{}^* \cup \ldots$
and $ev^* = ev_1{}^* \cup ev_2{}^* \cup \ldots \cup ev_i{}^* \cup \ldots.$

The appendix contains sketches of proofs of both theorems.

As pointed out in an earlier paper[32], the operational semantics is incomplete. In particular, it cannot preventatively make a reactive rule true by making its *antecedent* false. For example, it cannot make the rule:

$$attacks(X, you, T, T+1) \wedge \neg\, prepared\text{-}for\text{-}attack(you, T+1)$$
$$\rightarrow surrender(you, T+1, T+2)$$

true by performing actions to make *prepared-for-attack(you, T+1)* true and so $\neg$ *prepared-for-attack(you, T+1)* false.

Also, it cannot proactively make a rule true by making its *consequent* true before its *antecedent* becomes true. For example, it cannot make:

$$enter\text{-}bus(T, T+1) \rightarrow have\text{-}ticket(T+1)$$

true by proactively making *have-ticket(T+1)* true, before *enter-bus(T, T+1)* is true.

We have investigated the completeness of the operational semantics with respect to the generation of more restricted *supported models*[54]. Informally speaking and ignoring composite events, a Herbrand model *M* 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-consequents* is true in ***M***. It is possible to show that, under certain conditions, the OS can generate all such supported models. These conditions include safety restrictions (*allowedness* or *range-restriction*) on ***R*** and ***L***, to ensure that (except for time variables) candidate actions are ground when they are selected as candidates for execution in step 2.3 of the operational semantics.

§7 Comparison with Other Work

LPS evolved from our attempts to reconcile and combine the reactive rules and destructive updates of production systems, active databases and BDI programming languages with the logical representations and semantics of logic programming, deductive databases and action/event theories in AI

An LPS framework **<*R, L, D*>** achieves this combination by using the logic programming semantics of ***L*** and ***D*** to define a model ***M*** whose purpose is to make the reactive rules ***R*** true. Because ***M*** is a model-theroretic structure, and not a theory, it can be constructed by using destructive updates, without losing its logical character. Because ***M*** contains the whole history of states and events, and because truth is defined for arbitrary sentences of FOL, the antecedents and consequents of rules in ***R*** can refer to complex events, generalizing the more restricted syntax of production systems, active database rules and BDI programming languages.

7.1 Deductive Databases

The importance of acknowledging and exploiting the distinctive characters of reactive rules and logic programs was first drawn to our attention by the distinction made by Nicolas and Gallaire[42] between deduction rules and integrity constraints in deductive databases. In LPS, logic programs can be viewed as deduction rules, and reactive rules can be viewed as integrity constraints. In deductive databases, the semantics of integrity constraints was the subject of considerable debate in the 1980s.

The two main views, to begin with, were the *consistency view* and the *theorem-hood view*. 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. For relational databases, the two views are 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 extends this idea, to the notion that a set of reactive rules should be true in a Herbrand model determined by a sequence of states and events.

In recent years, the field of deductive databases has evolved into the field of Datalog, in which databases are represented by logic programs without function symbols. Datalog±[8] extends Datalog with existential rules, having existentially quantified consequents. Existential rules are similar to reactive rules in LPS, and can be viewed similarly as integrity constraints. However in Datalog±, existential rules contain only state conditions, and are used to answer queries in a single database state. In LPS, reactive rules contain both state conditions and events, and are used to monitor updates and generate sequences of database states. As a result, existential rules and reactive rules have different semantics, reflecting their different use.

In LPS, logic programs are used to generate a perfect model, with the aim of making the reactive rules true. Because, truth is defined for arbitrary sentences of FOL, reactive rules could in theory have the form of arbitrary sentences of FOL. In practice, the syntax of reactive rules is restricted, both for efficiency and to suit their intended use of generalizing the reactive rules of production systems, active databases, and BDI agent programming languages.

Similarly, because FOL-perfect models are constructed bottom-up in separate strata, state conditions in LPS, which are evaluated in lower strata, can also have arbitrary FOL syntax. The desirability of having such FOL state conditions was inspired in large part by transaction logic[6], which uses FOL conditions in database 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.

7.2 Abductive Logic Programming

The distinction between logic programs and integrity constraints, which is the foundation for the distinction between logic programs and reactive rules in LPS, also underpins abductive logic programming[23] (ALP). In ALP, a program consists of a triple $<\boldsymbol{L}, \boldsymbol{IC}, \boldsymbol{A}>$, where $\boldsymbol{L}$ is a logic program, $\boldsymbol{IC}$ is a set of integrity constraints, and $\boldsymbol{A}$ is a set of "abducible" predicates, not defined by $\boldsymbol{L}$. Given a goal $\boldsymbol{G}$**,** which is an observation to explain or a state to achieve, the task is to generate a set Δ of ground atoms in the vocabulary of the abducible predicates such that $\boldsymbol{L} \cup \Delta$ solves $\boldsymbol{G}$, and $\boldsymbol{L} \cup \Delta$ satisfies $\boldsymbol{IC}$. Similarly to the case of deductive databases, different notions of solving a goal and satisfying an integrity constraint have been proposed.

In LPS, solving a goal $\boldsymbol{G}$ and satisfying reactive rules $\boldsymbol{R}$ are understood in the same way, as meaning that $\boldsymbol{G} \cup \boldsymbol{R}$ is true in the FOL-perfect model of $\boldsymbol{L} \cup \Delta$, where Δ represents a sequence of states and events. Compared with other semantics for solving a goal and satisfying integrity constraints, this model-theoretic semantics makes it possible for integrity constraints (and therefore both reactive rules and state conditions) to be arbitrary formulas of FOL, evaluated by using the standard definition of truth for sentences of FOL.

7.3 Logic Programming Semantics

In LPS, logic programs $\boldsymbol{L}$ play a supporting role, used to define canonical models that make the reactive rules $\boldsymbol{R}$ true. In earlier papers, these models were the perfect models of locally stratified logic programs[43]. In this paper, we extend local stratification and perfect models to programs including FOL conditions, and we prove soundness and the frame theorem for FOL-stratified logic programs. It would be interesting to extend the proofs to the more general case of well-founded models[52], and to explore whether alternative sets of possible concurrent events could be represented by alternative stable models[18].

7.4 Agent Languages

LPS is a direct descendant of our work on ALP agents[28], which embed ALP in the thinking component of a BDI-like agent[45] cycle. In ALP agents, the logic program $\boldsymbol{L}$ represents the agent's beliefs, and the goals and integrity constraints $\boldsymbol{G} \cup \boldsymbol{IC}$ represent the agent's goals (or desires). 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[24, 38]. In contrast with both ALP agents and KGP agents, the operational semantics of LPS employs a destructively updated database that represents the current state.

The destructive updates of LPS were inspired in part by their use in BDI-agent languages such as AgentSpeak[44]. In AgentSpeak, programs are collections of plans that have the form:

event E: conditions C $\Leftarrow$ *goals G and actions A.*

The event E can be the addition or deletion of a belief literal or a goal atom, stored in a database. The conditions C query the current state, and the goals G and actions A update the database by adding or deleting goals and beliefs. As a result, plans combine some of the functionality of both reactive rules and logic programs in LPS. However, they do not allow complex events in the event part of plans, and they do not include temporal constraints. Moreover, they do not a have a logical semantics.

A number of other authors have also developed agent languages and systems in a logic programming context. For example in both DALI[10] and EVOLP[7], events transform an initial agent logic program into a sequence of logic programs. ERA[3] extends EVOLP by adding complex events, complex actions, and event-condition-action rules. The semantics of the evolutionary sequence of logic programs in DALI, EVOLP and ERA is given by an associated sequence of models. In LPS, this sequence is represented instead by a single model of a single logic program using timestamps.

FLUX[50] is a constraint logic programming language for the design of intelligent agents that reason about their actions using the fluent calculus, in which states are represented as lists. Although it is claimed that updates are performed destructively, the list representation of states requires the explicit use of recursion both to query whether a fluent is a member of a state, and to delete the fluent if it is terminated by an action. In LPS, states are represented by sets of fluents, and membership and deletion are performed by associative look-up. In FLUX, states can be updated by sensing actions, but there seems to be no analogue of the reactive rules of LPS.

Eiter et al.[13] 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[17] maps agent programs into temporal modal logic sentences of the form "past or present conditions imply present or future conclusions". As in LPS, the computational task is to generate a model in which such agent programs are *true*. Unlike MetaTEM, which uses frame axioms for updating states and employs a possible world semantics, LPS uses an operational semantics with destructive updates and a semantics in which all states and events are combined in a single model.

7.5 Active Databases

As pointed out by Bailey et al.[4], although they differ in their intended applications and research communities, agent systems and active databases employ similar approaches to programming reactive systems. For example, event-condition-action (ECA) rules in active databases are similar to agent plans in BDI agents. Both active databases and BDI agents 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. Zaniolo[53], for example, uses a situation calculus-like representation with frame axioms, and reduces ECA rules to logic programs. Like Zaniolo[53], Statelog[34] uses a situation calculus-like representation with frame axioms, and gives ECA rules a logic programming semantics. Fernandes et al.[15] also views ECA rules in terms of change of state, but uses the event calculus as the basis for an ECA language coupled with a deductive database.

LPS employs a similar model-theoretic semantics for state transitions, specified by the event theory *ET*. However, it differs from them in distinguishing between the semantics of logic programs and the semantics of reactive rules, and in employing destructive updates in the operational semantics.

Active databases are one way of implementing database updates. Other methods for updating deductive databases have also been explored and are summarized in the textbook by Abiteboul et al[1]. These methods perform destructive updates, but without giving them a logical semantics. In contrast, LPS, gives destructive updates the semantics of constructing a single canonical model in an attempt to make the reactive rules true.

7.6 Production systems

Arguably, production systems are the simplest kind of reactive system. It was the attempt to understand the difference and relationship between production rules and logic programming rules that eventually led to our development of ALP agents and LPS. Several other authors have made related attempts to provide production rules with a declarative semantics.

Raschid[46], for example, maps production rules that add facts into logic programs, and production 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 perform forward chaining. Baral and Lobo[5] translate production rules into the situation calculus represented as a logic program with the stable model semantics.

Recently, there has been a revival of work on implementing production systems in logic programming terms. For example, Damasio et al.[11] use incremental Answer Set Programming (ASP) to realize different conflict resolution strategies for the RIF-PRD production system dialect. Eiter et al.[14] simulate production systems in ASP with an interface to an external environment, performing state changes by updating and accessing the environment via action atoms and external atoms. Rezk and Kifer[48] combine production rules and ontologies, using transaction logic.

In the majority of these approaches production rules are mapped into logic programs. In contrast, in the semantics of LPS, both logic programs and reactive rules have their own distinctive characters.

7.7 Action and Event Theories in AI

The situation calculus[40] and event calculus[33] are among the most established formalisms for representing and reasoning about fluents, actions and other events in artificial intelligence. Golog[35] extends the situation calculus with imperative programming language constructs, which are like composite actions in LPS. Although the semantics of Golog is expressed in second order logic, as Levesque et al.[35] points out, in most practical cases composite actions in Golog can be translated into pure Prolog programs and can be given a minimal model semantics. This minimal model semantics is similar to the semantics of LPS.

The situation calculus, event calculus and Golog all employ frame axioms to reason about change. Destructive changes are not possible in these systems, because it is not possible to delete axioms in the middle of the proof of a theorem. In contrast in LPS, states and events are model-theoretic structures. As a consequence, they can be constructed piecemeal by means of destructive updates, without destroying any axioms used in a proof.

To the best of our knowledge, LPS is the only framework that combines destructive updates with a logic-based semantics. For example, although STRIPS[16] uses destructive updates for planning, it does not have a logical semantics. On the other hand, all the action languages surveyed by Turner[51] have a logical semantics, but use frame axioms in their operational semantics.

Moreover, none of these languages include any component similar to the reactive rules of LPS.

7.8 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 contribution comes from the use of the domain theory ***D***, to reason about the combined effects of concurrent actions, in the spirit of Miller and Shanahan's[41) treatment in the event calculus. Recently, Khandelwal and Fox[25) 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.

Our use of a Linda-like shared state to handle concurrency is similar to the approach of Dovier et al.[12). In particular, 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.[12), to arbitrate between the conflicting actions of different agents in the pursuit of different goals.

The explicit representation of time in LPS is similar to its use by Loo et al.[37) and Hellerstein[22) for programming distributed and parallel systems. Although frame axioms are represented explicitly in Hellerstein[22), they are not used in the implementation, which uses instead "traditional storage technology rather than re-deriving tuples each timestep". Our frame theorem can be regarded as justifying the use of such technology.

7.9 State Transition Semantics of Algorithms

The semantics of LPS as a programming language builds upon the idea that logic programs can be understood both declaratively in logical terms, and imperatively as goal-reduction procedures. Arguably, this idea has had limited impact in Computing, largely because conventional imperative programming languages are mainly concerned with algorithms whose semantics are defined in terms of state transitions.

As Reisig[47) 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 $F : Q \to Q$ (the next-state function)". He advocates formalizing such systems by means of abstract state machines (ASM), but points out that, already in the first volume of *The Art of Computer Programming*[26), Donald Knuth suggested state transition systems as a general semantics for algorithms.

ASMs formalize states as algebraic structures, which are like model-theoretic structures consisting of objects and functions. States in ASMs are abstract, in the sense that they need not be defined symbolically. However, state transitions are defined by next-state functions represented in symbolic form, typically by means of guarded assignment statements, which are like production systems in which all the rules that are triggered in a given state fire in parallel.

Compared with LPS, in which all states are combined into a single model, in ASM only the individual states are model-theoretic in character. Moreover, because programs consist of sets of guarded assignment statements, they have the same limitations, as productions systems, compared with LPS.

An LPS framework ***<R, L, D>*** is a reactive state transition system in which logic programs ***L*** and domain theories ***D*** play a supporting role to reactive rules ***R***. Harel[20)] 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*".

LPS can be viewed as an attempt to reconcile Harel's two kinds of computational formalism, with reactive rules providing the reactive component, and logic programs providing structure for the "transformational" part.

§8 Future Work

LPS has its origins in AI knowledge representation and reasoning systems, but for the sake of efficiency and to focus on the features required for more mainstream 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 and proactive actions has also been deliberately left out.

There are two complementary directions for future work. One direction is to reintroduce into LPS some of the more powerful, but more expensive features of ALP agents. 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.

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.

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 Maurice Bruynooghe, Agostino Dovier, Andrea Formisano, Luis Pereira, Enrico Pontelli and Ken Satoh for their comments on an earlier draft of this paper. We also thank the anonymous reviewers for their helpful suggestions.

References

1. Abiteboul, S., Hull, R. and Vianu, V., *Foundations of Databases.* Addison-Wesley 1995.
2. Agotnes, T., Van Der Hoek, W., Rodriguez-Aguilar, J. A., Sierra, C., and Wooldridge, M. "On the logic of normative systems," *Proc. of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pp. 1181-1186), 2007.
3. Alferes, J.J., Banti, F. and Brogi A., "An Event-Condition-Action Logic Programming Language," *10th European Conference on Logics in Artificial Intelligence. JELIA 06:* Lecture Notes in Artificial Intelligence 4160, Springer-Verlag, pp. 29- 42, 2006.
4. Bailey, J., Georgeff, M., Kemp, D., Kinny, D. and Ramamohanarao, K., "Active databases and agent systems—A comparison," *Rules in Database Systems*, pp. 342-356, 1995.
5. Baral, C. and Lobo, J., "Characterizing production systems using logic programming and situation calculus" http://www.cs.utep.edu/baral/papers/char-prod-systems.ps, 1995.
6. Bonner, A. and Kifer, M., "Transaction logic programming," In Warren D. S., (ed.), *Logic Programming: Proc. of the 10th International Conf.,* pp. 257-279. 1993.
7. Brogi, A., Leite, J. A. and Pereira, L. M., "Evolving Logic Programs," In: *8th European Conference on Logics in Artificial Intelligence (JELIA'02),* S. Flesca, S. Greco, N. Leone, G. Ianni (eds.), Spriger-Verlag, LNCS 2424, Springer-Verlag, pp. 50-61, 2002.
8. Calì, A., Gottlob, G., and Lukasiewicz, T., "Datalog±: a unified approach to ontologies and integrity constraints," *Proc. of the 12th International Conference on Database Theory*, pp. 14-30, ACM, 2009.
9. Carriero, N. and Gelernter, D., "Linda in Context," *Communications of the ACM.* Volume 32 Issue 4, 1989.
10. Costantini, S. and Tocchio, A., "The DALI Logic Programming Agent-Oriented Language," In: .Alferes, J.J., Leite, J. (eds.) *JELIA 2004. LNCS (LNAI), Vol. 3229*, Springer, Heidelberg, pp. 685–688, 2004.
11. Damásio, C., Alferes, J. and Leite, J., "Declarative semantics for the rule interchange format production rule dialect," *The Semantic Web–ISWC 2010*, pp. 798-813, 2010.
12. Dovier, A., Formisano, A. and Pontelli, E., "Autonomous agents coordination: Action languages meet CLP() and Linda," *Theory and Practice of Logic Programming*, 2012.
13. Eiter, T., Subrahmanian, V.S. and G Pick, G., "Heterogeneous active agents, I," *AI Journal, Vl. 108, No. 1-2,* pp. 179-255, 1999.
14. Eiter, T., Feier, C., and Fink, M., "Simulating production rules using ACTHEX," *Correct Reasoning*, pp. 211-228, 2012.

15. Fernandes, A.A.A., Williams, M.H., and Paton, N., "A Logic-Based Integration of Active and Deductive Databases," *New Generation Computing, Vol 15, No 2,* pp. 205—244, 1997.
16. Fikes, R. E. and Nilsson, N. J., "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, *2(3),* pp.189-208, 1972.
17. Fisher, M., "A Survey of Concurrent METATEM - The Language and its Applications," *Lecture notes in computer science, 827*, Springer Verlag pp. 480-505, 1994.
18. Gelfond, M. and Lifschitz, V., "The stable model semantics for logic programming," In *Proceedings of the 5th International Conference on Logic programming*, pp. 1070-1080, 1988.
19. Gurevich, Y., "Evolving algebras 1993: Lipari guide," *Specification and validation methods*, pp. 9-36, 1995.
20. Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Programming* 8 pp. 231-274, 1987.
21. Hausmann, S., Scherr, M. and Bry, F., "Complex Actions for Event Processing," Research Report, Institute for Informatics, University of Munich, 2012.
22. Hellerstein, J.M., "The Declarative Imperative: Experiences and Conjectures in Distributed Logic," *SIGMOD Record 39(1),* 2010.
23. Kakas, A. C., Kowalski, R. and Toni, F., "The Role of Logic Programming in Abduction," *Handbook of Logic in Artificial Intelligence and Programming 5*, Oxford University Press, pp. 235-324, 1998.
24. Kakas, A. C., Mancarella, P., Sadri, F., Stathis, K, and Toni, F., "The KGP model of agency," In *Proc. ECAI-2004*, 2004.
25. Khandelwal, A. and Fox, P., "General Descriptions of Additive Effects via Aggregates in the Circumscriptive Event Calculus." Technical Report, Computer Science, Rensselaer Polytechnic Institute, New York, 2012.
26. Knuth, D. E., *The Art of Computer Programming. Vol. 1: Fundamental Algorithms*, Addison-Wesley, 1973.
27. Kowalski, R., "Predicate logic as programming language," In *IFIP Congress*, *Vol. 74*, pp. 569-544, 1974.
28. Kowalski, R. and Sadri, F., "From Logic Programming Towards Multi-agent Systems," *Annals of Mathematics and Artificial Intelligence, Vol 25,* pp. 391-419, 1999.
29. Kowalski, R. and Sadri, F., "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, 2009.
30. Kowalski, R. and Sadri, F., "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*), pp. 200-218, 2010.
31. Kowalski, R. and Sadri, F., "Abductive Logic Programming Agents with Destructive Databases," *Annals of Mathematics and Artificial Intelligence*, *Vol 62, No 1*, pp. 129-158, 2011.
32. Kowalski, R. and Sadri, F., "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. 2012.

33. Kowalski, R. and Sergot, M., “A Logic-based Calculus of Events,” In: *New Generation Computing*, *Vol. 4, No.1*, pp. 67—95, 1986.
34. Lausen, G., Ludäscher, B. and May, W., “On Active Deductive Databases: The Statelog Approach,” In: *Transactions and Change in Logic Databases,* Decker, H., Freitag B., Kifer, M., Voronkov, A. (eds.), LNCS 1472, Springer, 1998.
35. Levesque, H. J., Reiter, R., Lesperance, Y., Lin, F. and Scherl, R. B., “GOLOG: A logic programming language for dynamic domains,” *The Journal of Logic Programming*, *31*(1), pp. 59-83, 1997.
36. Lloyd, J.W. and Topor, R.W., “Making PROLOG More Expressive,” *Journal of Logic Programming 1*, *3*, pp. 225-240, 1984.
37. Loo, B. T., Condie, B. T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I., “Declarative Networking,” In *Communications of the ACM (CACM),* 2009.
38. Mancarella, P., Terreni, G., Sadri, F., Toni, F. and Endriss, U., “The CIFF Proof Procedure for Abductive Logic Programming with Constraints: Theory, Implementation and Experiments,” *Theory and Practice of Logic Programming*, 2009.
39. Manna, Z. and Pnueli, A., *The temporal logic of reactive and concurrent systems: specifications*. Vol. 1. Springer, 1992.
40. McCarthy, J. and Hayes, P., “Some Philosophical Problems from the Standpoint of Artificial Intelligence,” *Machine Intelligence 4,* Edinburgh University Press. pp. 463-502, 1969.
41. Miller R. and Shanahan, M., “Some Alternative Formulations of the Event Calculus.” *Computational logic: logic programming and beyond*. Springer-Verlag, 452-490, 2002.
42. Nicolas, J.M. and Gallaire, H., “Database: Theory vs. Interpretation,” In: Gallaire, H., Minker, J. (eds.), *Logic and Databases,* Plenum, New York, 1978.
43. Przymusinski, T., “On the declarative semantics of stratified deductive databases and logic programs,” *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, J. Minker (Ed.) pp. 193 – 216, 1987.
44. Rao, A., “AgentSpeak (L): BDI agents speak out in a logical computable language,” *Agents Breaking Away*, pp. 42-55, 1996.
45. Rao, A. S. and Georgeff, M. P., “BDI Agents: From Theory to Practice,” *International Conference on Multiagent Systems - ICMAS* , pp. 312-319, 1995.
46. Raschid, L., “A Semantics for a Class of Stratified Production System Programs,” *J. Log. Program. 21(1),* pp. 31—57, 1994.
47. Reisig, W., “The Expressive Power of Abstract-State Machines,” *Computing and Informatics, Vol. 22*, pp. 209–219, 2003.
48. Rezk, M. and Kifer, M., “Formalizing production systems with rule-based ontologies,” *Foundations of Information and Knowledge Systems*, pp. 332-351, 2012.
49. Shanahan, M., *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. MIT Press, 1997.
50. Thielscher, M., “FLUX: A logic programming method for reasoning agents,” *Theory and Practice of Logic Programming*, *5*(4-5), pp. 533-565, 2005.
51. Turner, H., “Non-monotonic Causal Logic,” In: Frank van Harmelen, Vladimir Lifschitz and Bruce Porter, Editor(s), *Foundations of Artificial Intelligence*, Elsevier, Vol. 3, pp. 759-776, 2008.
52. Van Gelder, A., Ross, K. A., and Schlipf, J. S., “The well-founded semantics for general logic programs,” *Journal of the ACM (JACM)*, *38*(3), pp. 619-649, 1991.

53. Zaniolo, C., "On the Unification of Active Databases and Deductive databases," In: *11th British National Conference on Databases,* pp. 23-39, 1993.
54. Kowalski, R. and Sadri, F., "Completeness of a Reactive System Language," Department of Computing, Imperial College London, 2014.

Appendix A Sketch of the Proof of the Frame Theorem 6.2

We need to show that

$perfect(D_{post} \cup \boldsymbol{L} \cup S^* \cup ev^*) = perfect(ET \cup D_{post} \cup \boldsymbol{L} \cup S_0^* \cup ev^*).$

The model $perfect(ET \cup D_{post} \cup \boldsymbol{L} \cup S_0^* \cup ev^*)$ can be constructed by partitioning the Herbrand base H of $ET \cup D_{post} \cup \boldsymbol{L} \cup S_0^* \cup ev^*$ into strata associated with the succession of time points $0, s(0), s(s(0))$:

$H_0 =$ $\{holds(p, 0) \mid p$ is an extensional fluent$\}$
$\{a \mid a$ is an atom defined in $L_{aux}\} \cup$
$\{happens(e, t, u) \mid e$ is a simple event and t and u are time points$\}$
For $i \geq 0$, $H_{3i+1} = \{holds(p, i) \mid p$ is an intensional fluent$\}$
$H_{3i+2} = \{initiated(p, i+1) \mid p$ is an extensional fluent$\} \cup$
$\{terminated(p, i+1) \mid p$ is an extensional fluent$\}$
$H_{3i+3} = \{holds(p, i+1) \mid p$ is an extensional fluent$\}$

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

Now, to proceed with the proof, note that

$ground(ET) = ET_1 \cup ET_2 \cup \ldots \cup ET_i \cup \ldots$
where $ET_i = \{holds(p, i) \leftarrow body \in ground(ET)\}$.

The theorem follows from the fact, which can be proved by induction on i, that for all $i \geq 0$:

$perfect(D_{post} \cup \boldsymbol{L} \cup S_0^* \cup S_1^* \cup \ldots \cup S_i^* \cup ev^*) =$
$perfect(ET_1 \cup ET_2 \cup \ldots \cup ET_i \cup D_{post} \cup \boldsymbol{L} \cup S_0^* \cup ev^*).$

Appendix B Sketch of the Proof of the** Soundness Theorem **6.1

As a result of the frame theorem, soundness can be restated in the form: Given an LPS framework **<*R*, *L*, *D*>** and an initial state S_0, where ***L*** and ***D*** are FOL-stratified, suppose for every set ext_i of external events, where $i > 0$, the OS selects a set of concurrent events $ev_i = ext_i \cup acts_i$ at step 0 of the ith iteration of the OS cycle.

Then $\boldsymbol{R} \cup D_{pre}$ is true in *perfect(*$D_{post} \cup \boldsymbol{L} \cup S^* \cup ev^*$*)* 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.

To show that D_{pre} is true in *perfect(*$D_{post} \cup \boldsymbol{L} \cup S^* \cup ev^*$*)* it suffices to show that D_{pre} is true in each *perfect(*$D_{post} \cup \boldsymbol{L} \cup S_{i-1}^* \cup ev_i^*$*)*. But this is ensured by step 0 of the OS.

The proof that $\boldsymbol{R}$ is true in *perfect(*$D_{post} \cup \boldsymbol{L} \cup S^* \cup ev^*$*)* follows from the fact that the condition for soundness in the statement of the theorem mimics the definition of truth for reactive rules. 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.