Abductive Logic Programming Agents with Destructive Databases

Robert Kowalski and Fariba Sadri

Department of Computing, Imperial College London, 180 Queens Gate, London SW7 2AZ

{rak, fs}@doc.ic.ac.uk

Abstract. In this paper we present an agent language that combines agent functionality with a state transition theory and model-theoretic semantics. The language is based on abductive logic programming (ALP), but employs a simplified state-free syntax, with an operational semantics that uses destructive updates to manipulate a database, which represents the current state of the environment. The language builds upon the ALP combination of logic programs, to represent an agent's beliefs, and integrity constraints, to represent the agent's goals. Logic programs are used to define macro-actions, intensional predicates, and plans to reduce goals to sub-goals including actions. Integrity constraints are used to represent reactive rules, which are triggered by the current state of the database and recent agent actions and external events. The execution of actions and the assimilation of observations generate a sequence of database states. In the case of the successful solution of all goals, this sequence, taken as a whole, determines a model that makes the agent's goals and beliefs all *true*.

Keywords – abductive logic programming, agent languages, model-theoretic semantics

1 Introduction

In this paper we present a Logic-based agent and Production System language, LPS [13, 21]), which combines a declarative semantics based on abductive logic programming (ALP) with an operational semantics inspired mainly by production systems, but also by practical agent languages. The operational semantics performs destructive updates on "beliefs" that represent the current state of the agent's environment. The declarative semantics identifies the purpose of the agent's actions as making the agent's goals all *true* in the minimal model of the world associated with the agent's beliefs and the sequence of database states.

The semantics of LPS is based on that of ALP (abductive logic programming) agents [12], which represent observations and actions in a non-destructive database and explicitly represent and reason about time or state, using a formal theory of change. The theory uses explicit *frame axioms* to reason, for example, that a fact holds in a given state of the database if it was initiated by an action or other event in an earlier state and was not terminated by any event taking place between the earlier state and the given state.

Although formal theories of time and state change, such as the situation calculus [14] and event calculus [20], solve the *declarative frame problem* of representing change of state, they incur the *computational frame problem* of reasoning efficiently about such changes. The usual way of reasoning with frame axioms in logic programming is by *reasoning backwards*, to show that a fact holds in a given state by finding the most recent earlier state in which it was initiated, and showing that the fact

has not been terminated in the meanwhile. The most obvious alternative way of reasoning is by *reasoning forwards*, whenever an event takes place, to generate a new copy of every fact that is not affected by the event. But no matter whether frame axioms are used to reason backwards or forwards, their use often gives rise to intolerable inefficiencies in practice.

Production systems and most agent languages, including those inspired by the BDI (Belief, Desire, Intention) model of agency [26], avoid the computational frame problem by ignoring the frame axiom and by performing destructive database updates instead. When an event takes place, due either to the agent's action or to some external agency, facts that are initiated by the event are added to the database and facts that are terminated are deleted. Any fact that is neither initiated nor terminated simply persists in the database without any need to reason that it persists. The persistence of such unaffected facts is an *emergent property* of the destructive changes to the database, rather than an operational procedure performed by reasoning with explicit frame axioms. The price of such destructive updates is that it is difficult to provide them with declarative semantics.

The operational semantics of LPS uses destructive change of database states, as in production systems and practical agent languages, but it obtains a declarative semantics by associating a minimal model with the agent's beliefs and the sequence of database states. The minimal model constrains the agent's actions to those that are motivated by making its goals true in the model.

The database in LPS is structured as a destructively changing, deductive database, with extensional predicates that are represented explicitly by facts and with intensional predicates that are represented implicitly by logic programs. Updates are performed by means of a transition theory *without frame axioms* that specifies both the preconditions of actions and the extensional facts initiated and terminated by events. Whereas the extensional facts are updated explicitly, intensional predicates are updated only implicitly as ramifications of the updates.

The state transition theory in LPS determines the *consequences* of the agent's actions and other events, but it does *not motivate* or determine its actions. The agent's actions are motivated by its goals, represented by reactive rules, and its beliefs, represented by the current state of the database, macro-actions definitions and planning clauses. This separation of the transition theory that determines the consequences of events from the beliefs that motivate the agent's actions helps to avoid the inefficiencies of planning from first principles.

In planning from first principles, a transition theory, with or without frame axioms, is used to construct plans of actions to achieve goals. Every consequence of an action can be used to solve a goal, including goals that are preconditions of other actions. This typically generates a large search space that is very inefficient to search.

Planning from second principles avoids these inefficiencies by using plan libraries (or plan schemata), which are precompiled abstract plans that solve typical planning problems that occur commonly in the given problem domain. In LPS, these abstract plans are represented by a combination of planning clauses and macro-action definitions (or clauses). Planning clauses specify abstract plans to achieve goals that have extensional predicates, and macro-action clauses define macro-actions in a hierarchy of atomic actions and macro-actions. Macro-actions facilitate a hierarchical style of planning, in which complex action goals are progressively decomposed into simpler action goals, in the spirit of hierarchical task networks (HTNs) [24]. Macro-action clauses are also similar to transaction definitions in Transaction Logic [2].

The operational semantics of LPS is driven by reactive rules, which are like production rules in production systems and event-condition-action rules in active database systems. Whenever an instance of the conditions of a rule becomes true in a database state, the operational semantics generates a goal to make the conclusion of the instantiated rule true in future states. The resulting sequence of database states generated both by external events and by the agent's own actions is like a Kripke possible world structure. However, in the semantics of LPS, the possible worlds are combined in a single model with state arguments in the spirit of the situation calculus and Golog [14].

In the remainder of the paper, we present motivating examples and background, and then the syntax, operational semantics and model-theoretic semantics of LPS. We assume the reader is familiar with logic programming, SLD resolution and the minimal model semantics of Horn clauses. This paper extends an earlier version [21].

1.1 Motivating Examples

In this section, we give examples in LPS. The language has two kinds of sentences:

reactive rules (or maintenance goals), which generalize both condition-action rules in production systems and event-condition-action rules in active databases, and

beliefs, which include a deductive database, planning clauses, macro-action clauses and a state transition theory, all represented by logic programming clauses.

The vocabulary of LPS includes predicates for fluents, events, and macro-actions, as well as state-independent predicates. The *state-independent predicates* include such predicates as $\leq$. *Fluents* are the state-dependent, extensional and intensional predicates that define the state of the database. *Events* are either atomic actions that can be performed by the agent or external events that are observable by the agent. Macro-action predicates represent complex actions that are defined in terms of atomic actions and other macro-actions. Atomic actions and macro-actions are also both referred to as *actions*. In addition, the vocabulary includes the *auxiliary* predicates *initiates*, *terminates* and *possible*, used in the state transition theory to specify the post-conditions of events and the preconditions of atomic actions.

The surface syntax of LPS does not have explicit state arguments. However, in the semantics (or internal syntax), fluents P have an additional argument $P(T)$ or equivalently two identical arguments $P(T, T)$, indicating that P holds in the state T (or at the time T). Events and macro-actions A have two additional arguments $A(T_1, T_2)$, indicating that the A takes place from T_1 to T_2. If A is an event (atomic action or external event) then $T_2 = T_1 + 1$.

In addition to ordinary conjunction $\wedge$, the surface syntax of LPS has two other conjunctions. The conjunction **:** stands roughly for *at the same time or immediately after,* and **;** stands for *after*. The surface syntax and semantics will be given more formally in Section 4. But, in the meanwhile, the semantics of the two conjunctions in the following examples can be understood as follows:

$P : Q$ means $P(T_1, T_2) \wedge Q(T_2, T_3)$

$P ; Q$ means $P(T_1, T_2) \wedge Q(T_3, T_4) \wedge T_2 \leq T_3$.

Here *P* and *Q* can be fluents, events, actions, or even state-independent predicates if we employ the convention that, if *P* is state-independent, then *P(T, T)*, *P(T)* and just plain *P* are all different ways of writing *P*. Throughout the paper, variables start in the upper case, and constants start in the lower case.

Example 1.1: We start with a simple example that illustrates the syntax and the basic features of the operational and model-theoretic semantics of LPS. We consider an online shopping scenario, similar to the running example in the W3C RIF Working Group document on rule interchange[1]. Reactive rules are used to welcome a customer when she logs in, and to take payment and issue confirmation when she checks out. *ID* is a unique identifier associated with the login session.

> *login(X)* **:** *customer(X) → welcome(X).*
> *checkout(X)* **:** *customer(X)* **:** *shop-cart(X, ID, Value)* **:** *Value > 0*
> *→ take-payment(X, ID, Value)* **;** *confirm(X, ID, Value).*

The symbol → denotes logical implication. The forward arrow is used for reactive rules, because they are used for forward reasoning: If an instance of the conditions of a reactive rule (representing a maintenance goal) holds in the current state, then the corresponding instance of the conclusion of the rule is added as a new achievement goal to the agent's existing achievement goals.

In this example, the new goals generated by the reactive rules are solved by macro-actions, in which a customer is *welcomed* with an appropriate offer. A *new* customer is welcomed with an offer of a promotional item. A *gold* customer is welcomed with an offer of a promotional item that is similar to an item recommended by her profile:

> *welcome(X)* ← *status(X, new)* **:** *promotional-item(Y)* **:** *offer(X, Y).*
> *welcome(X)* ← *status(X, gold)* **:** *promotional-item(Y)* **:** *profile(X, Z)* **:**
> *similar(Y, Z)* **:** *offer(X, Y).*

Here the backward arrow ← also denotes logical implication, but is used for backward reasoning. In general, the backward arrow is used for beliefs, all of which have the semantics of logic programming clauses.

The predicates *login* and *checkout* represent external events that can be observed by the agent, *welcome* is a macro-action and *offer, take-payment* and *confirm* are atomic actions that can be performed by the agent. The predicates *similar* and > are state-independent. The predicates *profile* and *status* are either extensional predicates, stored in the database explicitly, or intensional predicates, defined in terms of extensional predicates. All other predicates are extensional predicates.

Events, which include both external events and the agent's own atomic actions, are used to update the database. They do so by initiating and terminating extensional predicates, as defined by a *transition theory*. For example, an observation of the customer adding an item to the shop cart initiates a new value for the shop cart, and the action of taking payment terminates the current value for the shop cart. An

[1] http://www.w3.org/2005/rules/wiki/RIF_Working_Group visited in June 2011

observation of a customer checking out terminates the customer's shop cart if the value of the shop cart is *0*:

initiates(add-to-shop-cart(X, ID, Value), shop-cart(X, ID, NewValue))
← shop-cart(X, ID, OldValue) **:** *NewValue = Old Value + Value.*
terminates(add-to-shop-cart(X, ID, Value), shop-cart(X, ID, OldValue))
← shop-cart(X, ID, OldValue).

initiates(take-payment(X, ID, Value), paid-shop-cart(X, ID, Value)).
terminates(take-payment(X, ID, Value), shop-cart(X, ID, Value)).

terminates(checkout(X), shop-cart(X, ID, Value)) ← Value = 0.

With the LPS operational semantics, when a customer logs in, forward reasoning with the first reactive rule generates a new achievement goal to *welcome* the customer. Backward reasoning with the macro-action definition generates a plan for solving the *welcome* goal. First the *status* of the customer is checked, and if it is *gold*, for example, the database is queried for a promotional item matching the customer's profile. If one is found, it is offered to the customer.

The internal syntax (meaning or semantics) of the reactive rules, the macro-action definitions and the transition theory is:

$login(X, T_1, T_2) \wedge customer(X, T_2) \rightarrow welcome(X, T_3, T_4) \wedge T_2 \leq T_3.$
$checkout(X, T_1, T_2) \wedge customer(X, T_2) \wedge shop\text{-}cart(X, ID, Value, T_2) \wedge Value > 0$
$\rightarrow take\text{-}payment(X, ID, Value, T_3, T_4) \wedge confirm(X, ID, Value, T_5, T_6) \wedge$
$T_2 \leq T_3 \wedge T_4 \leq T_5.$

$welcome(X, T_1, T_2) \leftarrow status(X, new, T_1) \wedge promotional\text{-}item(Y, T_1) \wedge$
$offer(X, Y, T_1, T_2).$
$welcome(X\ T_1, T_2) \leftarrow status(X, gold, T_1) \wedge promotional\text{-}item(Y, T_1) \wedge$
$profile(X, Z, T_1) \wedge similar(Y, Z) \wedge offer(X, Y, T_1, T_2).$

initiates(add-to-shop-cart(X, ID, Value), shop-cart(X, ID, NewValue), T)
← shop-cart(X, ID, OldValue, T) ∧ NewValue = Old Value + Value.
terminates(add-to-shop-cart(X, ID, Value), shop-cart(X, ID, OldValue), T)
← shop-cart(X, ID, OldValue, T).

initiates(take-payment(X, ID, Value), paid-shop-cart(X, ID, Value), T).
terminates(take-payment(X, ID, Value), shop-cart(X, ID, Value), T).

terminates(checkout(X), shop-cart(X, ID, Value), T) ← Value=0.

All variables are implicitly universally quantified with scope the entire implication. However, variables in the conclusion of a reactive rule that are not in the conditions are implicitly existentially quantified with scope the conclusion. For example:

$$login(X, T1, T_2) \wedge customer(X, T_2) \rightarrow welcome(X, T_3, T_4) \wedge T_2 \leq T_3.$$

stands for: $\forall X \ \forall T_1 \ \forall T_2 \ (login(X, T_1, T_2) \wedge customer(X, T_2)$
$\rightarrow \ \exists T_3 \ \exists T_4 \ (welcome(X, T_3, T_4) \wedge T_2 \leq T_3)).$

The LPS operational semantics is an agent cycle that, given an initial state $<\boldsymbol{W_0}, \boldsymbol{G_0}>$ and a sequence of observations of external events Ob_i, generates a sequence of database states $\boldsymbol{W_i}$, goal states $\boldsymbol{G_i}$ and atomic actions a_i interleaved with the input observations:

$$<\boldsymbol{W_0}, \boldsymbol{G_0}>, a_0, Ob_0, \ldots, <\boldsymbol{W_i}, \boldsymbol{G_i}>, a_i, Ob_i \ldots$$

Suppose in this example that the initial goal state and initial action are both empty ($\boldsymbol{G_0}= \{true\}, a_0= \phi$) and the initial database is:

$\boldsymbol{W_0}$ =*{customer(john), customer(bill), status(john, new), status(bill, new), promotional-item(iceTea), shop-cart(john, j1, 0), shop-cart(bill, b1, 0)}.*

In practice when a customer logs in, a new *ID* is generated for the session and the *shop-cart* is initialised with value *0*. We ignore these details here.

Suppose also that the agent makes the following observations:

Ob_0 = *{login(john)}*
Ob_1 = *{login(bill), checkout(john), add-to-shop-cart(bill, b1, 25)}*
Ob_2 = *{checkout(bill)}.*

Then, depending upon the goal selection and search strategies, the operational semantics may produce the following sequence of database states and atomic actions:

Database State	Action
$\boldsymbol{W_0}$	$a_0 = \phi$
$\boldsymbol{W_1} = \boldsymbol{W_0}$	a_1 = *offer(john, iceTea)*
$\boldsymbol{W_2} = \boldsymbol{W_1} -$ *{shop-cart(john, j1, 0) }* $\cup$ *{shop-cart(bill, b1, 25)}*	a_2 = *offer(bill, iceTea)*
$\boldsymbol{W_3} = \boldsymbol{W_2}$	a_3 = *take-payment(bill, b1, 25)*
$\boldsymbol{W_4} = \boldsymbol{W_3}$ - *{shop-cart(bill, b1, 25)}* $\cup$ *{paid-shop-cart(bill, b1, 25)}*	a_4 = *confirm(bill, b1, 25)*

The set of beliefs consisting of this sequence together with the observations and the macro-action definitions, all with explicit state parameters, is a Horn clause logic program, with a unique minimal model, in which the reactive rules are all *true*.

Example 1.2: The following is a reformulation in LPS of an example in [4], which involves buying a gift. According to the scenario in [4], the agent first checks what gifts are available in Harrods, and forms a plan to go to Harrods and purchase the gift. Then for some reason the agent does not succeed in going to Harrods, and a special plan revision rule changes the plan to purchasing that same gift from Dell. In LPS, the

beliefs required for this scenario can be formalized without plan revision rules, by means of the planning clauses:

have(Object) ← *sells(harrods, Object)* **;** *goto(harrods)* **;**
purchase(harrods, Object).
have(Object) ← *sells(harrods, Object)* **;** *online(Store)* **:** *sells(Store , Object)* **;**
go-online(Store) **;** *purchase(Store, Object).*

The extensional predicate *have* is initiated by the atomic action *purchase*:

initiates(purchase(Store, Object), have(Object)).

Notice that *purchase* also initiates a reduction in the agent's financial resources, as a consequence of the action. But there is unlikely to be a planning clause or macro-action having such a reduction as a motivation for purchasing an object.

For simplicity, we assume *goto* and *go-online* are atomic actions, and all the other predicates are extensional predicates. We can specify preconditions for the action *purchase(Store, Object)*, but we will ignore these here.

The LPS operational semantics is neutral with respect to the search strategy used to explore the search space and to select an action to execute. To obtain the behavior of the scenario described in [4], the search strategy would need to try the planning clauses in the order in which they are written, try the first action *goto(harrods)*, and if it fails, either re-attempt the action later or try an alternative plan, involving the action *go-online(dell)*.

To illustrate this, consider the initial goal state $\boldsymbol{G_0}$ = *{have(Object)}*, and assume the database $\boldsymbol{W_0}$ contains such facts as *sells(harrods, laptop)*, *sells(dell, laptop)* and *online(dell)*. The example has no reactive rules. So the agent's task is simply to achieve the initial goal. It proceeds by reasoning backwards with the planning clauses, resulting in a sequence of goal states $\boldsymbol{G_1}$, $\boldsymbol{G_2}$, ….. Each $\boldsymbol{G_i}$ represents a search space of alternative (partial) solutions (we call each alternative a *goal clause*), and the search strategy determines how $\boldsymbol{G_{i+1}}$ is generated by SLD resolution using goal clauses in $\boldsymbol{G_i}$ and the agent's beliefs.

In Section 4 we will see that the LPS cycle uses the internal syntax for goal clauses. However, for simplicity, we can also represent goal clauses in the external syntax. For example, expressed in the external syntax, $\boldsymbol{G_1}$ might contain the three goal clauses:

*{have(Object)***,** *sells(harrods,Object)* **:** *goto(harrods)* **;** *purchase(harrods,Object)***,** *goto(harrods)* **;** *purchase(harrods,laptop)}.*

At this stage the search strategy can attempt the action *goto(harrods)* underlined in the third goal clause. If the action fails, the search strategy can explore other alternatives in the search space, possibly generating $\boldsymbol{G_2}$ as:

$\boldsymbol{G_1}$ ∪ *{sells(harrods,Object)* **;** *online(Store)* **:** *sells(Store,Object)* **:** *go-online(Store)* **;**
*purchase(Store,Object)***,** *go-online(dell)* **;** *purchase(dell,laptop)}.*

The search strategy can now attempt to execute the new action *go-online(dell)*, or re-attempt *goto(harrods)*. If it decides to try the new action and succeeds (generating G_3) and then attempts *purchase(dell, laptop)* and succeeds, then it reaches a new goal state G_4 containing the goal clause *true*.

The sequence of database states generated is $W_0 = W_1 = W_2 = W_3$, $W_4 = W_3 \cup$ *{have(laptop)}*. The agent's initial goal G_0 is *true* in the unique minimal model of the Horn clause logic program corresponding to the internal representation of this sequence of database states and successfully executed actions (with explicit state parameters).

Example 1.3: The following example is a variant of the waste-collecting robot in [19] and the vacuuming robots in [4] and [23]. In this example, whenever the robot observes waste, she collects it in a bag, and whenever the bag is full, she empties the bag into a bin.

Reactive Rules: *waste-at(X) → goto(X)* **;** *put-waste-in-bag-at(X).*
bag-full **:** *bin-at(X) → goto(X)* **;** *empty-bag-at(X).*

Macro-action definitions:

goto(X) ← robot-at(X).
goto(X) ← robot-at(Y) **:** *towards(Y, Z, X)* **:** *step(Y, Z, X)* **;** *goto(X).*

Here *waste-at* and *bag-full* are externally observed events, *robot-at* and *bin-at* are extensional predicates, *put-waste-in-bag-at*, *empty-bag-at* and *step* are atomic actions, and *goto* is a macro-action. The predicate *towards* is a state-independent predicate (whose definition is not displayed). The intention is that *Z* is one step away from *Y* in the direction of *X*.

In this example, for simplicity, only the atomic action *step* updates the database. The external events simply trigger the reactive rules, and the atomic actions *put-waste-in-bag-at* and *empty-bag-at* simply change the external state of the world, without affecting the database. The state transition theory specifies the effects of *step*:

initiates(step(Y,Z,X), robot-at(Z)).
terminates(step(Y,Z,X), robot-at(Y)).

The atomic actions *step*, *put-waste-in-bag-at* and *empty-bag-at* can be performed only if their preconditions hold. The transition theory also specifies these preconditions:

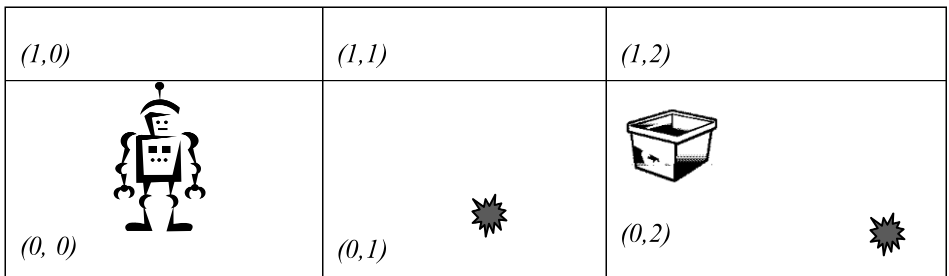
possible(step(Y,Z,X)) ← robot-at(Y).
possible(put-waste-in-bag-at(X)) ← robot-at(X).
possible(empty-bag-at(X)) ← robot-at(X).

We will see later that preconditions are automatically checked by the operational semantics, to ensure that no action is chosen for execution unless its execution is possible. This is necessary in this example with the actions *put-waste-in-bag-at(X)* and *empty-bag-at(X)* because the sequential connective ***;*** before these actions in the reactive rules means that the robot might have moved away from the location *X* before

performing these actions. However, the automatic check is redundant in the case of the action *step(Y, Z, X)* because the sequential connective **:** before this action in the macro-action definition means that the precondition is guaranteed to hold when the action is chosen for execution. The definition of preconditions is necessary only for the agent's own atomic actions, and not for external events, because the external world ensures that external events occur only if they *can* occur.

Notice that the sequential connective **;** between the atomic action *step(Y, Z, X)* and the macro-action *goto(X)* means that the macro-action of going to *X* can be interrupted between steps. We will show this possibility in the following illustration of the operational semantics.

Suppose that the initial goal state and initial action are both empty ($\boldsymbol{G_0}$ **=** *{true}*, $a_0 = \phi$) and the initial database is W_0 = *{ robot-at((0,0)), bin-at((0,2))}*.



Suppose also that the robot makes the following observations:

> Ob_0 = *{waste-at((0, 2))}*
> Ob_1 = *{waste-at((0, 1))}*
> Ob_2 = *{bag-full}*

Then, depending upon the goal selection and search strategies, the operational semantics may produce the following sequence of database states and atomic actions:

Database State	*Action*
$W_1 = W_0$	a_1 = *step((0,0),(0,1),(0,2))*
W_2 = *{robot-at((0,1)), bin-at((0,2))}*	a_2 = *put-waste-in-bag-at((0,1))*
$W_3 = W_2$	a_3 = *step((0,1),(0,2),(0,2))*
W_4 = *{robot-at((0,2)), bin-at((0,2))}*	a_4 = *empty-bag-at((0,2))*
$W_5 = W_4$	a_5 = *put-waste-in-bag-at((0,2))*

We will see later in Section 5 how the selection and search strategies can generate this particular sequence of database states and actions. In the meanwhile, we note that the robot has interrupted its original plan to pick up the waste at location *(0, 2)* when it observes waste at *(0, 1)*, taking advantage of the fact that it is already at *(0, 1)*. It further delays its original goal when it observes that the bag is full and needs emptying. Finally, it returns to the first goal and picks up the waste at *(0, 2)*.

As in Example 1.1, in the internal syntax, the reactive rules are all *true* in the unique minimal model of the Horn clause logic program consisting of the internal

representation of the agent's beliefs including the sequence of database states, atomic actions and observations.

2 Background

2.1 Informal comparison of Agent Languages and LPS

Practical agent languages can be regarded as an extension of production systems, in which condition-action rules are generalised to trigger-guard-body rules. Both production systems and agent languages manipulate a database of facts, which represents the current state of the environment. The database is updated destructively both by the agent's observations and by the agent's actions. The agent's goals are represented either as goal facts in the database, or in a separate stack of goals and actions, which represents the agent's intentions.

Like condition-action rules in production systems, trigger-guard-body rules, often called *plans* in agent languages, provide two main functions. Arguably their primary function is as *reactive rules*, triggered by changes in the database, verifying that the guard holds and adding the corresponding body either to the database or the stack of goals. However, in practice they often function as *goal-reduction rules*, triggered by a current goal, verifying the guard, and adding the body as a collection of goals to the database or stack of intentions.

LPS borrows from production systems and agent languages their state-free syntax and their destructively updated database. It uses the database to represent the current state of the environment, and represents goals (or alternative candidate intentions) as a set of goal clauses, executing them as in SLD resolution. The search strategy and selection function can treat the set as a stack in the same way that Prolog implements a restricted version of SLD resolution. Alternatively, it can use the selection function more freely to interleave planning with plan execution.

The main difference between LPS and more conventional agent languages is that LPS interprets and represents reactive plans and goal-reduction plans differently, and this difference is exploited to provide LPS with a model-theoretic semantics. It interprets goal-reduction plans as *beliefs* and represents them as logic programs. It provides them with a backward reasoning operational semantics and a minimal model declarative semantics. It interprets reactive plans as *maintenance goals* and represents them as integrity constraints (as in abductive logic programming). It provides them with a forward reasoning operational semantics and the model-theoretic semantics of integrity constraints.

Production systems and agent languages typically represent actions as additions or deletions of facts in the database. LPS employs a more structured representation of actions in the tradition of the situation calculus and event calculus [14, 20]. Additions and deletions are consequences of a transition theory. However, instead of using frame axioms, LPS uses destructive change of state to avoid the computational inefficiencies of the frame problem.

In production systems and agent languages, when the conditions of more than one rule hold, a choice needs to be made between "firing" the different conclusions of the rules. In production systems, this is made by means of a conflict resolution strategy. In agent languages, it is made by selecting one of the conclusions as an intention, and possibly repairing the resulting plan if the plan fails. It is partly because

there is no semantic constraint on conflict resolution that these languages do not have a model-theoretic semantics.

In contrast, the model theory of ALP and LPS imposes the constraint that for ***every*** goal there needs to be ***some*** way of making the goal *true*. Therefore, when a reactive rule is interpreted as a maintenance goal or integrity constraint, then the conclusion of the rule becomes an (achievement) goal that must be made *true*, whenever its conditions become *true*.

To make an achievement goal *true*, the search strategy needs to find a collection of beliefs and a set of actions, so that the goal is *true* if the beliefs are *true* and the actions are executed successfully. Thus the analogue of conflict resolution in ALP and LPS is the use of the search strategy to find a way to solve every goal.

But different solutions can achieve their goals to different degrees. For example, different ways of purchasing a gift cost more or less money, and involve more or less effort. Different ways of collecting waste can involve more or fewer steps. Search strategies for finding best solutions in the time available are a well-established area of AI and Operations Research. By separating search spaces from search strategies, LPS benefits not only from the model-theoretic semantics facilitated by this separation, but also from the possibility of exploiting efficient search strategies, like A* and branch-and-bound. In ALP agents, we have explored the use of Decision Theory for this purpose [22].

However, there is another kind of conflict resolution, called *refraction*, which is not a matter of search strategy, and which also applies to LPS. In production systems, refraction is used to trigger a *condition-action* rule only when its *condition* first becomes *true,* preventing the rule from firing again if its *condition* remains *true*. In LPS we deal with refraction as a matter of knowledge representation, by using events, rather than fluents, to trigger reactive rules, as in Examples 1.1 and 1.3.

2.2 Abductive Logic Programming

LPS is based on abductive logic programming (ALP) [9] and abductive logic programming agents (ALP agents) [12]. ALP extends logic programming (LP) by allowing some predicates, ***Ab****,* the *abducibles*, to be undefined, in the sense that they do not occur in the conclusions of clauses. Instead, they can be assumed, but are constrained directly or indirectly by a set ***IC*** of *integrity constraints*.

Thus an *ALP framework* <***L****,* ***Ab****,* ***IC***> consists of a logic program ***L***, a set of abducibles ***Ab***, and a set of integrity constraints ***IC***. The predicates in the conclusions of clauses in ***L*** are disjoint from the predicates in ***Ab***. An atom whose predicate is in ***Ab*** is called *abducible* (or open). In LPS, the abducible atoms represent atomic actions, and the integrity constraints represent reactive rules (or maintenance goals).

In LPS, we represent reactive rules in the form *condition* $\rightarrow$ *conclusion,* where *condition* and *conclusion* are conjunctions of atoms. For simplicitly, we restrict logic programs to Horn clauses [11]. This has the advantage that Horn clauses have a unique minimal model [5]. The restriction can be relaxed, as we will discuss later.

Definition 1. Given an ALP framework <***L****,* ***Ab****,* ***IC***> and a goal *C* that is a conjunction of atoms (which can be empty, equivalent to *true*)*,* a *solution* of the goal is a set of atomic sentences $\boldsymbol{\Delta}$ in the predicates ***Ab***, such that both *C* and ***IC*** are *true* in the minimal model of $\boldsymbol{L} \cup \boldsymbol{\Delta}$. □

This semantics is one of several that have been proposed for ALP and for integrity constraints more generally. It has the advantage that it provides a natural basis for the semantics of LPS. Roughly speaking, in LPS, the analogue of $\boldsymbol{L}$ is the agent's beliefs, and the analogue of $\boldsymbol{\Delta}$ is the sequence of the agent's actions. The analogue of C and $\boldsymbol{IC}$ being *true* in the minimal model of $\boldsymbol{L} \cup \boldsymbol{\Delta}$ is $\boldsymbol{G_0}$ together with the reactive rules being *true* in the minimal model determined by the agent's beliefs, augmented by its actions and observations.

The ALP agent model [12] embeds the IFF [8] proof procedure for ALP in an observation-thought-decision-action cycle, in which abducible atoms $\boldsymbol{Ab}$ represent events, logic programs $\boldsymbol{L}$ represent the agent's beliefs, and integrity constraints $\boldsymbol{IC}$ represent the agent's maintenance goals. Logic programs give the pro-active behaviour of goal-reduction procedures, and integrity constraints give the reactive behaviour of reactive rules. However, goals and beliefs also have a declarative semantics, inherited from ALP. The ALP agent cycle generates a sequence of actions in the attempt to make the initial goal and the maintenance goals *true.*

In ALP agents, the agent's environment is an external, destructively changing semantic structure. The set $\boldsymbol{\Delta}$, on the other hand, is the agent's internal representation of its interactions with the environment. This internal representation is monotonic, in the sense that observations and actions are time-stamped and state representations are derived by a transition theory, such as the situation or event calculus. In contrast, in production systems, in most agent systems and in LPS, the environment is simulated by an internal, destructively changing database.

In addition to the different internal representations of the environment in ALP agents and in LPS, the other main difference is in the nature of their observations. In LPS, observations are external events, which initiate and terminate extensional predicates. But in ALP agents, observations also include fluents. The ALP agent operational semantics uses backward reasoning with the IFF proof procedure to generate alternative hypothetical events as abductive explanations of these fluent observations. We have excluded this kind of abductive reasoning from LPS, because it is much more complex than straight-forward assimilation of external events, and because abduction is not a feature of practical production systems and BDI agents. Its possible incorporation into LPS is a possible topic of research for the future.

3 LPS Language – Informal Description

In this section we give an informal description of the LPS language. In the next section we define the language and its internal, state-based representation formally.

3.1 The Database

The LPS semantics is defined in terms of a minimal model associated with a sequence of databases state transitions $\boldsymbol{W_0}$, a_0, Ob_0, …, $\boldsymbol{W_i}$, a_i, Ob_i … , where the $\boldsymbol{W_i}$ represent the successive states of the database, the a_i represent the agent's actions, and the Ob_i represent a set of observations.

The databases $\boldsymbol{W_i}$ contain only the extensional predicates of a deductive database, e.g. *customer(john), spent-to-date(john, 500).* Because the transition from $\boldsymbol{W_i}$ to $\boldsymbol{W_{i+1}}$ is implemented by destructive assignment, the facts in $\boldsymbol{W_i}$ are written without state arguments. This means that facts that are not affected by the transition persist without the inefficiencies of being copied explicitly from one state to the next.

In addition to extensional predicates, which represent database states explicitly, there are intensional predicates defined by clauses L_{int}. For example:

*status(X, gold) ← spent-to-date (X, V) **:** 500 ≤ V.*
*status(X, new) ← spent-to-date (X, V) **:** V < 500.*

Here *spent-to-date* is an extensional predicate, which changes directly as the result of actions, such as *take-payment*, and *status* is an intensional predicate, which changes as a ramification of changes to the predicate *spent-to-date*.

The state-independent predicates are defined by logic programming clauses in $L_{stateless}$. For example:

*similar(X, Y) ← cd(X) **:** dvd(Y).*

3.2 The State Transition Theory

State transitions are defined by a set ***Tr*** of clauses of the form:

initiates(e, p) ← init-conditions.
terminates(e, p) ← term-condition.
possible(a) ← pre-conditions.

where *e* is event, which is an atomic action or external event, *a* is an atomic action, and *p* is an extensional predicate. The conditions *init-conditions* and *term-conditions* are *qualifying conditions*, and together with *pre-conditions* are formulas that are checked in the current state. In the case of events that are actions, the qualifying conditions do not check whether an action is executable -*pre-conditions* does that - but determine what the action initiates or terminates once it is executed.

Not every event needs to initiate or terminate database facts. In particular, some actions can be external actions, which have no direct impact on the database.

In the operational semantics, the search strategy can choose an atomic action *a* for execution only if it is executable, in the sense that it checks whether *possible(a) holds* in the current state of the agent's beliefs. We will define the notion that a sentence *holds* with respect to a set of Horn clauses in Section 5.

Because observations of external events happen only if they can happen, there is no need to specify their preconditions in ***Tr***. However, because we allow several external events to be observed concurrently, we assume that concurrently observed events are *independent*, in the sense that no event initiates a fluent that is concurrently terminated by another.

It is important to note that state transition theories are **not** used for planning, but only to perform the state transitions associated with the agent's actions and external events. We use planning and macro-actions clauses for planning.

3.3 Goals

In addition to the changing state of the database, the LPS operational semantics maintains an associated changing goal state G_i, which is a set of goal clauses. Each such goal clause can be regarded as a *partial plan* for achieving the initial achievement goals G_0 and the additional achievement goals generated by the reactive

rules. Both the initial goals and the additional goals are reduced to sub-goals by the logic programs used to define intensional predicates, macro-actions, state-independent predicates and planning clauses. Goals coming from different instances of reactive rules can be solved independently and concurrently.

The intended semantics of goals is that, for every $\boldsymbol{G_i}$, one of the goal clauses in $\boldsymbol{G_i}$ should be *true* in the model that is generated by the LPS cycle. $\boldsymbol{G_0}$ may contain only the empty clause, which is equivalent to the sentence *true*, as is typical of production systems. Informally speaking, the cycle succeeds in state *n*, if $\boldsymbol{G_n}$ contains the empty clause (or *true*). However, the cycle does not terminate when it succeeds, because future observations may trigger future goals.

Initial goal clauses can contain actions, fluents, state-independent predicates, and the logical connectives **:** and **;** but not external events. For example, the following goal clause requires that a promotional item is determined and discounted by 20%, and then the item and its new price are advertised:

promotional-offer(Item) **:** *discount(Item, 20%, NewPrice)* **;** *advertise(Item, NewPrice).*

3.4 Reactive rules

The rules in the set $\boldsymbol{R}$ of *reactive rules* have the same form *condition→conclusion* and the same implicit quantification as ALP integrity constraints, where *condition* is a conjunction of atoms and *conclusion* has the same form as a goal clause. Reactive rules are executed by checking whether an instance of the *condition* holds in the current state of the agent's beliefs, and if it does, then the corresponding instance of the *conclusion* is added to *every* goal clause in $\boldsymbol{G_i}$.

The *condition* can also include a single atom representing an atomic action successfully executed in the last cycle and any number of atoms from the set of current observations. Thus $\boldsymbol{R}$ can include the event-condition-action rules of active databases. For example:

take-payment(X, ID, Value) **:** *Value ≥ 50 → issue-sport-voucher(X, ID).*

3.5 Macro-actions

It would be possible to write agent programs using reactive rules alone, restricting the conclusions of reactive rules to atomic actions, and to extensional and intensional predicates that are checked in the current state as implicit consequences of the agent's actions or as serendipitous consequences of external events. Such reactive rules would be sufficient for implementing purely reactive agents. However, macro-actions and planning clauses in LPS make it possible to implement agents with more deliberative/proactive capabilities.

Macro-actions are complex actions defined in terms of simpler (atomic and macro) actions, fluents and state-independent predicates. Macro-actions, defined by the set of clauses $\boldsymbol{L_{macro}}$, are like hierarchical tasks in HTNs, transactions in *TR* Logic and complex actions in Golog. Examples of macro-actions are the action *welcome*, in Example 1.1, and the action *goto* in Example 1.3.

3.6 Planning clauses

Agent programs written using only reactive rules and definitions of macro-actions achieve fluent goals only emergently and implicitly. Planning clauses allow

extensional fluent goals to be achieved explicitly. To ensure that the agent's beliefs are *true* with respect to the state transition theory, we impose the restriction that the last condition in a planning clause is an atomic action that initiates the conclusion fluent, as determined by the transition theory. L_{plan} represents such plans for achieving future states of the database. For example:

have(Object) ← *is-store(Store)* **:** *sells(Store, Object)* **:**
goto(Store) **;** *purchase(Store, Object).*

Note that that the conclusions of plans represent the *motivations* of the agent's actions, in contrast with the transition theory, which represents all the *consequences* of its actions.

Thus the planning clauses, together with the macro-action clauses implement planning from second principles, namely planning by using pre-compiled plan schemata. However, planning clauses can also be used to implement planning from first principles, by including a planning clause of the form:

p ← *pre-conditions* **:** *init-conditions* **:** *a*

for every pair of clauses:

initiates(a, p) ← *init-conditions.*
possible(a) ← *pre-conditions.*

in the transition theory, where *a* is an atomic action.

Whether the planning clauses are used for planning from first principles or planning from second principles, they share with classical planning the repeated reduction of fluent goals to fluent and action sub-goals. Because LPS is neutral with respect to search and action selection strategies, different strategies for interleaving planning and execution can be implemented. At one extreme, as in classical planning, plans can be fully generated before they are executed. At the other extreme, actions can be executed as soon as they are generated in a partial plan.

3.7 Goal Reduction

In the operational semantics, atomic goals in goal clauses are solved by performing SLD resolution using the internal syntax for both goals and beliefs. These beliefs include the clauses in W_i, L_{int}, L_{plan}, L_{macro}, $L_{stateless}$ and a_i, but not the transition theory Tr and not the observations in Ob_i.

3.8 Sequential Conjunction

The two sequential conjunctions **:** and **;** are both used for sequencing, but **:** means "at the same time as or immediately afterwards", and **;** means "anytime afterwards". From a practical point of view, the conjunction **;** allows a sequence of goals and actions to be interrupted and interleaved with the solution of other goals, whereas the conjunction **:** forces the sequence to be solved and executed without interruption.

For example, the goal clause *open-fridge* **:** *get-drink* ensures that the agent will attempt to get a drink immediately after opening the fridge, minimizing the possibility that the fridge will be closed (by the agent or an external event) between the two actions. In contrast, the goal clause *open-fridge* **;** *get-drink* allows the agent to try

to get a drink any time after opening the fridge. Thus it is more likely that the fridge door may be shut between the two actions, and the *open-fridge* action will have to be repeated. Thus the use of sequential conjunction ***:*** is related to *transaction atomicity* in database systems. We will not pursue this relationship further in this paper.

The connective ***:*** is required whenever we need to express that a conjunction of fluents should hold simultaneously in the same state. For example, in the definitions of intensional predicates, such as:

above(X, Y) $\leftarrow$ *on(X, Z)* **:** *above(Z, Y).*

or in the conditions of reactive rules, such as:

take-payment(X, ID, Value) **:** *spent-to-date(X, Value)* **:** *Value* $\geq$ *1000*
$\rightarrow$ *issue-voucher(X, ID, 50).*

The connective ***:*** is also needed in L_{plan} or L_{macro} clauses when we want to ensure that the qualifying conditions of an atomic action hold when the action is attempted, for example:

clear(X) $\leftarrow$ *on(Y, X)* **:** *move-to(Y, table).*

In all other cases the programmer has a choice between using ***:*** and ***;*** depending on how much interleaving and how much flexibility with respect to temporal ordering is desired. For example both clauses below are acceptable in L_{plan}, but can generate different behaviour, as explained earlier:

on(X, Y) $\leftarrow$ *clear(X)* **:** *clear(Y)* **:** *move-to(X, Y).*
on(X, Y) $\leftarrow$ *clear(X)* **;** *clear(Y)* **;** *move-to(X, Y).*

Fluents can also be sequenced by ***;***. For example the goal clause:

professor **;** *head-of-department* **;** *dean*

represents the ambitions of an agent first to become a professor, then head of department, and then dean, but not necessarily without interruption. Similarly, actions and fluents can also be sequenced by ***;***. For example:

dean **;** *change-retirement-rules* **:** *retire.*

4 LPS Language – Formal Description

The vocabulary of LPS is divided into fluent, event, macro-action, auxiliary and state-independent predicates. The *fluent* predicates consist of extensional and intensional predicates. The *event* predicates consist of atomic actions, and observations of external events. The *auxiliary* predicates consist of predicates *initiates, terminates,* and *possible* in the transition theory. The *state-independent* predicates include such predicates as $\leq$. All these sets of predicates are mutually exclusive.

The LPS framework employs a stateless *surface* syntax, which is syntactic sugar for an underlying *internal* syntax with explicit state arguments (which specify the semantics of the surface syntax). We use the internal syntax when describing the operational and the model-theoretic semantics later in the paper. Here we describe both the surface syntax and its semantics.

The surface syntax of all LPS components is defined in terms of *sequences* of atomic formulas (or atoms), where consecutive atoms are linked by **:** or **;**. The *syntax of sequences* is defined recursively. The base case is the empty sequence, which is also the empty clause, written as *true*. If P is an atom and S is a sequence, then $P : S$ and $P ; S$ are sequences.

Below, where it is clear from the context, we use the terminology (fluent, state-independent, atomic action, macro-action, event, extensional, intensional) predicate to mean an atom with such a predicate symbol.

The initial goal state $\boldsymbol{G_0}$ is a set of goal clauses, each of which is a sequence with no observation atoms. Other goal states $\boldsymbol{G_i}$, derived in the LPS cycle are sets of clauses expressed in the internal syntax with state arguments. They do not appear in the surface syntax.

$\boldsymbol{L_{stateless}}$ clauses have the form: $P \leftarrow P_1 : P_2 : \ldots : P_n,\ 0 \le n$,
where P and each P_i are state-independent predicates.

$\boldsymbol{L_{int}}$ clauses have the form: $P \leftarrow P_1 : P_2 : \ldots : P_n,\ 1 \le n$,
where P is an intensional predicate, each P_i is a fluent or state-independent predicate, and at least one P_i is a fluent.

$\boldsymbol{L_{macro}}$ clauses have the form: $M \leftarrow S$, where M is a macro-action predicate,
and S is a sequence containing at least one fluent or action predicate and no event.

$\boldsymbol{L_{plan}}$ clauses have the form: $P \leftarrow S$, where P is an extensional predicate,
and S is a sequence containing no event, and ending in an atomic action.

$\boldsymbol{R}$ reactive rules have the form: $[Evt_1 \wedge Evt_2 \wedge \ldots \wedge Evt_n \wedge A] : Q_1 : Q_2 : \ldots : Q_m \rightarrow S$
where S is a non-empty sequence, containing no event, and each Q_i is a fluent or state-independent predicate, A is an atomic action, and each Evt_i is an event. All Evt_i and A may be absent, in which case $1 \le m$, otherwise $0 \le m$.

The conditions of reactive rules do not contain macro-actions, because the sequence of states from T_1 to T_2 associated with the semantics $M(T_1, T_2)$ of a macro-action M is not accessible in the current state T of the database.

The only place we use the connective $\wedge$ in the external syntax is in the conditions of reactive rules, where it is needed to refer to observations and the action executed all at the *same* time in the current state. The alternative of using the connective **:** to conjoin these events would mean that they occur in sequence rather than simultaneously, which is not what is intended.

In principle, it would be possible to keep a database of the history of past events and to allow the conditions of reactive rules to refer to this history. This would require an extension of the external syntax, but can be dealt with without extension in the internal syntax because of its explicit representation of state.

Tr clauses have the forms: $initiates(e, p) \leftarrow P_1 : P_2 : ... : P_n$
$terminates(e, p) \leftarrow P_1 : P_2 : ... : P_n$
$possible(a) \leftarrow P_1 : P_2 : ... : P_n$

where e is an event, a is an atomic action, p is an extensional fluent, each P_i is a fluent or state-independent predicate, and $0 \leq n$.

The semantics of each formula F of LPS, including atomic formulas (atoms), predicates, goals, rules, clauses and sequences, is denoted by F^*. Every such formula F^* can be written in the form $F^*(T_1, T2)$, where T_1 and $T2$ are as explained below. The semantics P^* of an atomic formula P is given by:

true is a state-independent formula, and *true** is *true*.
If P is a state-independent formula, then P^* also written $P^*(T)$ and $P^*(T, T)$ is P.
If P is a fluent, then P^* also written $P^*(T, T)$ is $P(T)$.
If P is an event, then P^* also written $P^*(T, T+1)$ is $P(T, T+1)$.
If P is a macro-action, then P^* *also written* $P^*(T_1, T_2)$ is $P(T_1, T_2)$.

The semantics of sequences is defined recursively, with the empty sequence having the semantics *true*.

Let P be an atom and S a sequence, with semantics P^* and S^* respectively.

Let F be $P : S$, where neither P nor S is state-independent.
Then $F^*(T_1, T_2)$ is $P^*(T_1, T) \wedge S^*(T, T_2)$.

Let F be $P ; S$, where neither P nor S is state-independent.
Then $F^*(T_1, T_4)$ is $P^*(T_1, T_2) \wedge S^*(T_3, T_4) \wedge T_2 \leq T_3$.

Let F be $P : S$ or $P ; S$.
If both P and S are state-independent, then F^* is $P^* \wedge S^*$ and state-independent.
If P is state-independent and S is not, then $F^*(T_1, T_2)$ is $P^* \wedge S^*(T_1, T_2)$.
If S is state-independent and P is not, then $F^*(T_1, T_2)$ is $P^*(T_1, T_2) \wedge S^*$.

The semantics $\boldsymbol{G_0}^*$ of the initial goal state $\boldsymbol{G_0}$ is the semantics of its sequences. Because all the $\boldsymbol{G_i}$, for i > 0, are expressed only in the internal semantics, $\boldsymbol{G_i}^* = \boldsymbol{G_i}$ for i > 0. All the variables in $\boldsymbol{G_i}$ and $\boldsymbol{G_i}^*$, for i ≥ 0, are existentially quantified.

Hence, in the internal syntax, goal clauses are conjunctions of atoms, and the goal states $\boldsymbol{G_i}^*$ all represent disjunctions of goal clauses. These goal states have a search tree structure, which is not apparent in the set representation. As in normal logic programming, other representations, including search tree and and-or tree representations are possible. For simplicity, we do not explore these other representations in this paper.

The semantics of an $\boldsymbol{L_{stateless}}$ clause $P \leftarrow P_1 : P_2 : ... : P_n, 0 \leq n$ is
$P \leftarrow P_1 \wedge P_2 \wedge ... \wedge P_n$.
The semantics of an $\boldsymbol{L_{int}}$ clause $P \leftarrow P_1 : P_2 : ... : P_n$ is
$P(T) \leftarrow P_1(T) \wedge P_2(T) \wedge ... \wedge P_n(T)$.
The semantics of an $\boldsymbol{L_{macro}}$ clause $M \leftarrow S$ is $M(T_1, T_2) \leftarrow S^*(T_1, T_2)$.

The semantics of an $\boldsymbol{L_{plan}}$ clause $\quad P \leftarrow S \quad$ is $\quad P(T_2) \leftarrow S^*(T_1, T_2)$.

The semantics of a reactive rule $[Evt_1 \wedge Ev_2 \wedge ... \wedge Evt_n \wedge A] : Q_1 : Q_2 : ... : Q_m \rightarrow S$ is
$[Evt_1(T\text{-}1, T) \wedge ... \wedge Evt_n(T\text{-}1, T) \wedge A(T\text{-}1, T)] \wedge Q_1(T) \wedge Q_2(T) \wedge ... \wedge Q_m(T) \rightarrow S^*$
if S is state-independent, and $[Evt_1(T\text{-}1, T) \wedge ... \wedge Evt_n(T\text{-}1, T) \wedge A(T\text{-}1, T)] \wedge Q_1(T) \wedge Q_2(T) \wedge ... \wedge Q_m(T) \rightarrow S^*(T_1, T_2) \wedge T \leq T_1$ otherwise.

As already mentioned in Example 1.1, in all implications, all variables are implicitly universally quantified with scope the entire implication, except for variables in the conclusions of reactive rules that are not in their conditions, which are existentially quantified with scope the conclusions. These existentially quantified variables contribute to the existentially quantified variables of goal clauses.

The semantics of ***Tr*** clauses:

$initiates(e, p) \leftarrow P_1 : P_2 : ... : P_n$ is $initiates(e, p, T) \leftarrow P_1(T) \wedge P_2(T) \wedge ... \wedge P_n(T)$
$terminates(e, p) \leftarrow P_1 : P_2 : ... : P_n$ is $terminates(e, p, T) \leftarrow P_1(T) \wedge P_2(T) \wedge ... \wedge P_n(T)$
$possible(a) \leftarrow P_1 : P_2 : ... : P_n$ is $possible(a, T) \leftarrow P_1(T) \wedge P_2(T) \wedge ... \wedge P_n(T)$

Finally if S is a set of formulas then S^* is the set of all F^* for F in S.

Note that all fluents in clauses of $\boldsymbol{L_{int}}^*$ contain the same state variable. Consequently the connective ***:*** in the surface syntax of $\boldsymbol{L_{int}}$ could be replaced by logical conjunction $\wedge$. Similarly, the connective ***:*** in $\boldsymbol{L_{stateless}}$ clauses could also be replaced by $\wedge$.

Notice also that the syntax and semantics impose the restriction that no two actions (whether an atomic action or a macro-action) have the same pair of state arguments in the conditions of the same clause or the conclusion of the same reactive rule. This is because, for simplicity, the LPS operational semantics executes at most a single atomic action in each cycle/state. Because the operational and model-theoretic semantics of LPS are both defined for the internal semantics, it is possible to define other surface syntaxes and to mix state-based and stateless syntaxes. The syntax chosen for this paper can be extended in several ways, but has the advantage of simplicity.

5 The Operational Semantics

The operational semantics manipulates the database by adding and deleting facts with extensional predicates. However, the model-theoretic semantics interprets the facts in state $\boldsymbol{W_i}$ as containing the implicit state argument i. We use the notation $\boldsymbol{W_i}^*$ to refer to facts containing explicit state arguments: $\quad \boldsymbol{W_i}^* = \{p(i) : p \in \boldsymbol{W_i}\}$.

Events, together with the state transition theory, update the database from one state to the next, as specified in the LPS cycle below. However, for the execution of an action a to be attempted all of its preconditions must hold in the current state of the database $\boldsymbol{W_i}$.

Definition 2. An atomic action a is *executable* in state $\boldsymbol{W_i}$ if and only if $possible(a)$ holds in $\boldsymbol{W_i}^* \cup \boldsymbol{L_{int}}^* \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{Tr}^*$. □

In the model theory, a sentence S *holds* in a set H of Horn clauses if and only if S is *true* in the minimal model of H. In the operational semantics, S *holds* if S can be

proved from *H*. However, truth in an arbitrary minimal model is not in general recursively enumerable. Nonetheless in many practical cases it can be determined by such proof procedures as SLD resolution. The operational semantics defined below is neutral with respect to how *holds* is determined.

The LPS cycle checks if an action is executable before choosing it for execution. This is to avoid wasting time trying to execute an action if the agent does not believe that its preconditions actually hold. The programmer can ensure that the preconditions of actions hold by writing the clauses in L_{plan} and L_{macro} in such a way that atomic actions immediately follow (conjoined by the connective **:**) their explicitly represented preconditions or other actions that initiate their preconditions. If the programmer does not do this, the cycle performs the check automatically. (However, it does not automatically *make* the preconditions hold.)

In the LPS cycle, when an atomic action is chosen for execution, all of its arguments (other than state arguments) need to be variable-free (a safety requirement). In addition, the selection function and search strategy need to be *timely*.

Definition 3. A selection function is *safe* if and only if, when it selects an atomic action, the action is ground (except possibly for state variables). A selection function is *timely* if and only if, when it selects an atomic action $a(t, t+1)$ in a goal clause *C*, then *C* contains no other atom which needs to be evaluated earlier in *C*. The term *t* can be a variable *T*, or a constant *i* instantiated in the previous cycle. A search strategy is *timely* if and only if, when it resolves an extensional atom in a goal clause *C* with the database, then *C* contains no other atom that needs to be evaluated earlier in *C*. □

Informally speaking, an atom in a goal clause *C* needs to be evaluated earlier than an atomic action $a(t, t+1)$ if the execution of $a(t, t+1)$ in *C* would make it impossible to solve the atom afterwards, because its state arguments would be in the past. Timeliness is not needed for soundness, but it makes LPS more complete.

Note that the selection function is not restricted to selecting atomic goals in the sequence in which they are written. Atoms can be selected and resolved, so that planning and execution are interleaved. However, to ensure the existence of safe selection functions, LPS frameworks need to be range-restricted (defined after the LPS cycle).

The internal syntax of LPS includes inequalities between states. For the model-theoretic semantics we need a theory L_{temp} that defines this inequality relation. This theory is not needed for the operational semantics, because timeliness and range-restriction ensure that if all other goals in a goal clause succeed, then all the inequalities between states in the goal clause are also *true*. So for implementation purposes we can assume that these inequalities are deleted from the clauses and rules. This is equivalent to resolving these inequalities with clauses in L_{temp}, which always succeeds.

The operational semantics is a potentially non-terminating cycle in which the agent repeatedly observes events in the environment, updates the database to reflect the changes brought about by those events, performs a bounded number of inferences, and selects an atomic action to execute. If there is no such action that can be executed within the bound or if the action is attempted and fails, then an empty action is generated. Similarly, if there is no observation available at the beginning of a cycle then the set of observations is empty.

Definition 4. LPS cycle: Let **Max** be a bound on the number of resolution steps to be performed in each iteration of the cycle. Given a range-restricted LPS framework <$\boldsymbol{W_0}$, $\boldsymbol{G_0}$, $\boldsymbol{Tr}$, $\boldsymbol{R}$, $\boldsymbol{L_{int}}$, $\boldsymbol{L_{stateless}}$, $\boldsymbol{L_{macro}}$, $\boldsymbol{L_{plan}}$>, a safe and timely selection function $\boldsymbol{s}$, a timely search strategy $\sum$, and a sequence of sets of observations Ob_0, Ob_1,…., the LPS cycle determines a sequence of state transitions <$\boldsymbol{W_0}$, $\boldsymbol{G_0}$>, a_0, Ob_0, …, <$\boldsymbol{W_i}$, $\boldsymbol{G_i}$>, a_i, Ob_i …. The transition from <$\boldsymbol{W_i}$, $\boldsymbol{G_i}$>, a_i, Ob_i to <$\boldsymbol{W_{i+1}}$, $\boldsymbol{G_{i+1}}$>, a_{i+1} is given by the following steps:

LPS0. If a_i is a non-empty action that has been successfully executed, then let E_i = $Ob_i \cup \{a_i\}$. Otherwise let $E_i = Ob_i$. Then:
$\boldsymbol{W_{i+1}} = (\boldsymbol{W_i} - \{p$: $e \in E_i$ and $terminates(e, p, i)$ holds in $\boldsymbol{W_i}^* \cup \boldsymbol{Tr}^* \cup \boldsymbol{L_{int}}^* \cup \boldsymbol{L_{stateless}}\}) \cup \{p$: $e \in E_i$ and $initiates(e, p, i)$ holds in $\boldsymbol{W_i}^* \cup \boldsymbol{Tr}^* \cup \boldsymbol{L_{int}}^* \cup \boldsymbol{L_{stateless}}\}$. We assume that if action a_i has been successfully executed then it is independent of the observations Ob_i, in the sense that no observation in Ob_i terminates a fluent that a_i initiates and vice versa. Note also that $\boldsymbol{a_i^*}$ **is** $a(i, i+1)$.

LPS1. For every instance $condition\ \sigma \rightarrow conclusion\ \sigma$ of a rule in $\boldsymbol{R^*}$ such that $condition\ \sigma$ holds in $\boldsymbol{W_{i+1}}^* \cup E_i^* \cup \boldsymbol{L_{int}}^* \cup \boldsymbol{L_{stateless}}$ add $conclusion\ \sigma$ to every clause in $\boldsymbol{G_i}$. Let $\boldsymbol{GR_i}$ be the resulting set of goal clauses.

LPS2. Using the selection function $\boldsymbol{s}$ and search strategy $\sum$, let $\boldsymbol{G_{i+1}}$ be a set of goal clauses, starting from $\boldsymbol{GR_i}$, derivable by SLD-resolution using the clauses in $\boldsymbol{W_{i+1}}^* \cup \boldsymbol{L_{int}}^* \cup \boldsymbol{L_{plan}}^* \cup \boldsymbol{L_{macro}}^* \cup \boldsymbol{L_{stateless}} \cup \{a_i^*\}$ such that one of the following holds:

LPS2.1 No goal clause containing an executable action is generated within the maximum number, **Max**, of resolution steps. This includes the case of an empty clause being generated. Then a_{i+1} is the *empty action* ϕ (an action that will always succeed, but has no effect on the database). Cycle will proceed into further rounds because further observations are possible. (An agent cycle must be perpetual; it never stops, because there can always be observations.)

LPS2.2 At least one goal clause whose selected atom is an executable action is generated within the maximum number, **Max**, of resolution steps. Then one such action $a(t, t')$ in a goal clause in $\boldsymbol{G_{i+1}}$ is chosen for execution by the search strategy $\sum$. Note that $a(t, t')$ might have been generated and selected in an earlier cycle, but not have been executable before. Moreover, even if it was selected and executable before, the search strategy might have chosen some other action. Furthermore, it might have been executed and failed. It might even have been executed before and succeeded, but might need to be executed again, because later goals, dependent upon it, have failed. Note t can be $i+1$ or a variable. If the action succeeds, then in effect a_{i+1} is observed in the next cycle. □

The specification of the LPS cycle given above is an operational semantics, not an efficient proof procedure. However, there are many refinements that would make it more efficient. These include the deletion of subsumed clauses (including all other goal clauses, once the empty goal clause has been generated), as well as the deletion of clauses containing fluents or actions whose state argument is instantiated to a state earlier that the current state.

Definition 5. The cycle *succeeds in state n* if and only if $\boldsymbol{G_n}$ contains the empty clause. □

Example 5.1: We return to Example 1.3 of Section 1.1. Here the purpose is to illustrate the sequence of goals generated by the operational semantics. To simplify the example, we employ obvious abbreviations for the predicate symbols and their arguments (omitting parentheses where it is unambiguous to do so). Also, we extend the external syntax defined earlier for clauses in G_0 to all goal clauses in G_i. Also we do not present the entire set of goal clauses in each goal state, but only the goal clauses selected by the search strategy to generate the sequence of actions in Example 1.3. This is the same set of goal clauses that would be generated by employing depth-first search implemented by means of a stack, as in Prolog and most BDI languages.

For simplicity, we ignore the definition of the state-independent predicate *toward* (*t* below). This is equivalent to assuming it is defined by facts, not displayed in the database. Assume, as before, that the initial state and sequence of observations is:

$W_0 = \{r(0,0),\ b(0,2)\}$ $\quad \boldsymbol{G_0} = \{true\}$ $\quad a_0 = \phi$
$Ob_0 = \{waste\text{-}at(0,\ 2)\}$
$Ob_1 = \{waste\text{-}at(0,\ 1)\}$
$Ob_2 = \{bag\text{-}full\}$

Then, depending upon the goal selection and search strategies, the operational semantics may produce the following sequence of state transitions. Atoms selected and resolved upon are underlined. New sequences of subgoals that are added by reactive rules are conjoined to all existing goal clauses using the connective $\wedge$, to indicate that the different sequences in the same goal clause are temporally independent of one another.

$W_1 = W_0$ $\qquad \boldsymbol{GR_0} = \{\underline{g(0,2)}\ ;\ p(0,2)\}$
$\boldsymbol{G_1}$ *includes* in addition to $\boldsymbol{GR_0}$ also the two goal clauses:
$\underline{r(Y)} : \underline{t(Y,\ Z,\ (0,2))} : s(Y,\ Z,\ (0,2))\ ;\ g(0,2)\ ;\ p(0,2)$
$\underline{s((0,0),\ (0,1),\ (0,2))}\ ;\ g(0,2)\ ;\ p(0,2)$
$a_1 = s((0,0),(0,1),(0,2))$

$W_2 = \{r(0,1),\ b(0,2)\}$ $\boldsymbol{GR_1}$ is obtained by adding the new goal sequence $g(0,1)\ ;\ p(0,1)$ to every goal clause in $\boldsymbol{G_1}$.
Therefore $\boldsymbol{G_2}$ *includes* the goal clause: $(\underline{g(0,1)\ ;}\ \underline{p(0,1)}) \wedge (g(0,2)\ ;\ p(0,2))$
$a_2 = p(0,1)$

$W_3 = W_2$ $\boldsymbol{GR_2}$ is obtained by adding the new goal sequence $g(0,2)\ ;\ e(0,2)$ to every goal clause in $\boldsymbol{G_2}$.
Therefore $\boldsymbol{G_3}$ *includes* the three goal clauses $(\underline{g(0,2)}\ ;\ e(0,2)) \wedge (g(0,2)\ ;\ p(0,2))$
$(\underline{r(Y)} : \underline{t(Y,\ Z,\ (0,2))} : s(Y,\ Z,\ (0,2))\ ;\ g(0,2)\ ;\ e(0,2)) \wedge\ (g(0,2)\ ;\ p(0,2))$
$(\underline{s((0,1),\ (0,2),\ (0,2))}\ ;\ g(0,2)\ ;\ e(0,2)) \wedge\ (g(0,2)\ ;\ p(0,2))$
$a_3 = s((0,1),(0,2),(0,2))$

$W_4 = \{r(0,2),\ b(0,2)\}$ $\boldsymbol{GR_3} = \boldsymbol{G_3}$.

G_4 *includes* the goal clause: $(\underline{g(0,2)}\ ;\ \underline{e(0,2))} \wedge (g(0,2)\ ;\ p(0,2))$
$a_4 = e(0,2)$

$W_5 = W_4 \quad GR_4 = G_4$.
G_5 *includes* the goal clause: $\underline{g(0,2)}\ ;\ \underline{p(0,2)}$
$a_5 = p(0,2)$

$W_6 = W_5 \quad GR_5 = G_5$. G_6 *includes true.*

As mentioned earlier, range-restriction ensures the existence of safe selection functions. It also ensures that the implicit existential quantification of variables in the conclusions of reactive rules is dealt with correctly.

Definition 6. An LPS framework $<W_0, G_0, Tr, R, L_{int}, L_{stateless}, L_{macro}, L_{plan}>$ is *range-restricted* if and only if all rules in R and all clauses in Tr, L_{int}, $L_{stateless}$, L_{macro}, L_{plan} and G_0 are range-restricted, where:

A sequence S is range-restricted if and only if every variable in an atomic action in S occurs earlier in the sequence.
A clause *conclusion* $\leftarrow$ *conditions* in L_{int} , $L_{stateless}$, L_{macro} , L_{plan} is range-restricted if and only if *conditions* is range-restricted and every variable in *conclusion* occurs in *conditions.*
A clause *conclusion* $\leftarrow$ *conditions* in Tr, where *conclusion* is *initiates(e, p)* or *terminates(e, p),* is range-restricted if and only if every variable in *p* occurs either in *conditions* or in *e.*
A clause *possible(a)* $\leftarrow$ *preconditions* in Tr, is range-restricted, because *possible(a)* is checked only when *a* is ground, to verify the executability of *a*.
A rule *condition* $\rightarrow$ *conclusion* in R is range-restricted if and only if every variable occurring in an atomic action *a* in *conclusion*, occurs either in the *condition* or in an atom earlier than *a* in the *conclusion.* □

6 Model-theoretic Semantics

The model-theoretic semantics requires a Horn clause definition L_{temp} of the inequality relations. Any correct definition will serve the purpose including, for example:

$$0 \leq T \qquad\qquad S+1 \leq T+1 \leftarrow S \leq T.$$

Every set S_n of sentences $W_0^* \cup ... \cup W_n^* \cup \{a_0^*, ..., a_{n-1}^*\} \cup Ob_0^* \cup ... \cup Ob_{n-1}^* \cup L_{stateless} \cup L_{int}^* \cup L_{temp} \cup L_{macro}^*$ is a Horn clause logic program. Therefore, S_n has a unique minimal model M_n. This model is like a Kripke structure of possible worlds M^i (minimal model of $W_i \cup L_{stateless} \cup L_{int}$) embedded in a single model M_n, where the actions and observations $\{(Ob_0, a_0), ..., (Ob_{n-1}, a_{n-1})\}$ determine the accessibility relation from one possible world to another. The macro-action definitions can be regarded as determining paths between possible worlds, like transactions in *TR* Logic.

6.1 Soundness

To prove the soundness of the LPS cycle, $\boldsymbol{L_{plan}}$ needs to be compatible with the transition theory $\boldsymbol{Tr}$. Compatibility ensures that the clauses in $\boldsymbol{L_{plan}}$* are *true* in all $\boldsymbol{M_n}$.

Definition 7. $\boldsymbol{L_{plan}}$ is *compatible* with $\boldsymbol{Tr}$ if every clause in $\boldsymbol{L_{plan}}$ is of the form
$p \leftarrow S : P_1 : P_2 : ... : P_n : a$ or $p \leftarrow S ; P_1 : P_2 : ... : P_n : a$,
where S is any sequence, and there exists an instance of a clause in $\boldsymbol{Tr}$ of the form
$initiates(a, p) \leftarrow P_1 : P_2 : ... : P_n$. □

It is easy to satisfy this condition, and all the examples in this paper, if done in full will have this property. However, the notion of compatibility that is required for the soundness of LPS can be extended to allow a more liberal syntax of $\boldsymbol{L_{plan}}$ clauses, for example one that would allow macro-actions as well as atomic actions as the last action in their conditions.

Theorem. Given a range-restricted LPS framework <$\boldsymbol{W_0}$, $\boldsymbol{G_0}$, $\boldsymbol{Tr}$, $\boldsymbol{R}$, $\boldsymbol{L_{int}}$, $\boldsymbol{L_{stateless}}$, $\boldsymbol{L_{macro}}$, $\boldsymbol{L_{plan}}$>, a safe and timely selection function $\boldsymbol{s}$, a timely search strategy $\sum$, and a sequence of sets of observations Ob_0, Ob_1,…., Ob_{n-1}, if $\boldsymbol{L_{plan}}$ is compatible with $\boldsymbol{Tr}$ and the cycle succeeds in state n, then some clause C_0 in $\boldsymbol{G_0}$* is *true* in $\boldsymbol{M_n}$ and all the rules in $\boldsymbol{R}$* are *true* in $\boldsymbol{M_n}$.

Sketch of proof: If the cycle succeeds in state n, then $\boldsymbol{G_n}$ contains the empty clause. The proof of this empty clause can be traced backwards to a sequence of goal clauses C_0 ,…,C_i, ….,C_m = *true*, where C_0 is in $\boldsymbol{G_0}$* and C_{i+1} is obtained from C_i in one of two ways:

1. In LPS1, C_{i+1} is C_i conjoined with *conclusion* σ for every instance *condition* $\sigma \rightarrow$ *conclusion* σ of a rule in $\boldsymbol{R}$* such that *condition* σ holds in $\boldsymbol{W}_{i+1}^* \cup E_i^* \cup \boldsymbol{L_{int}}^* \cup \boldsymbol{L_{stateless}}$.
2. C_{i+1} is obtained by SLD-resolution between C_i and some clause C in $\boldsymbol{W}_{i+1}^* \cup \boldsymbol{L_{int}}^* \cup \boldsymbol{L_{plan}}^* \cup \boldsymbol{L_{macro}}^* \cup \boldsymbol{L_{stateless}} \cup \{a_i^*\}$ in LPS2 or by implicit resolution of inequalities with clauses in $\boldsymbol{L_{temp}}$.

It suffices to prove the **lemma:** All the C_i are *true* in $\boldsymbol{M_n}$. The lemma implies that $\boldsymbol{C_0}$ and all the rules in $\boldsymbol{R}$* are *true* in $\boldsymbol{M_n}$.

Proof of lemma: The lemma follows by induction, by showing the base case C_m = *true* is *true* in $\boldsymbol{M_n}$ and the induction step if C_{i+1} is *true* in $\boldsymbol{M_n}$, then C_i is *true* in $\boldsymbol{M_n}$. The base case is trivial. For the induction step, there are two cases: In case 1 above, if C_{i+1} is *true* in $\boldsymbol{M_n}$, then C_i is *true* in $\boldsymbol{M_n}$, because if a conjunction is *true* then so are all of its conjuncts.

In case 2 above, the clauses C_{i+1} and C_i are actually the negations of clauses in ordinary resolution. So, according to the soundness of ordinary resolution, $\neg C_{i+1}$ is a logical consequence of $\neg C_i$ and C. Therefore, if both C and C_{i+1} are *true* in $\boldsymbol{M_n}$, then C_i is *true* in M. But any clause C in $\boldsymbol{W}_{i+1}^* \cup \boldsymbol{L_{int}}^* \cup \boldsymbol{L_{macro}}^* \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{temp}} \cup \{a_i^*\}$ is *true* in $\boldsymbol{M_n}$ by the definition of $\boldsymbol{M_n}$. It suffices to show that all clauses in $\boldsymbol{L_{plan}}$* are also *true* in $\boldsymbol{M_n}$. But this follows from the compatibility of $\boldsymbol{L_{plan}}$ with $\boldsymbol{Tr}$. □

This theorem is restricted in two ways. First, it considers only the first *n* sets of observations. Second, it considers only the case in which the actions needed to solve all the goals in G_0 and introduced by the reaction rules are successfully executed by state *n*. Both of these restrictions can be liberalised, mainly at the expense of complicating the statement of the theorem, but the proofs are similar. We omit the theorems and their proofs for lack of space. However, it is worth noting that to deal with potentially non-terminating sets of observations, we need minimal models M_ω determined by the potentially infinite Horn clause program $W_0^* \cup ... \cup W_n^* \cup ...\{a_0^*, ..., a_n^*, ...\} \cup Ob_0^* \cup ... \cup Ob_n^* \cup ... L_{stateless} \cup L_{int}^* \cup L_{temp} \cup L_{macro}^*$.

Note also that LPS can be extended to include negation in both the conditions and conclusions of reaction rules and in the conditions of clauses. The most obvious such extension is to the case of locally stratified programs with their perfect models.

6.2 Completeness

Because of the completeness result for the IFF proof procedure [8] for ALP, it might be expected that a similar completeness result holds for LPS: Given a minimal model M of some clause C_0 in G_0 and of all the rules in R, it might be expected that there exists some search strategy $\sum$ that together with the LPS cycle could generate some related model M', possibly determined by a subsequence of the actions of M. But this is not always possible. Like production systems and BDI agents, LPS can only generate models that make reactive rules *true* by performing actions after their conditions have been made *true*. In particular, it cannot generate models that make rules *true* by making their conditions *false.* For example:

Example 6.1:

***R**:* *debit-transaction(Amount)* **:** *account-balance(B)* **:** *B < Amount*
→ pay-penalty(Amount-B)

***Tr**:* *initiates(pay-into-account(Amount), account-balance(New))*
← account-balance(Old) **:** *New = Old + Amount*
terminates(pay-into-account(Amount), account-balance(Old))
← account-balance(Old)

$\mathbf{W_0}$ = *{account-balance(0)}* G_0 = *{true}* $a_0 = \phi$
Ob_0 = { } Ob_1 = { *debit-transaction(100)*}

Here *debit-transaction* is an observation and *pay-penalty* and *pay-into-account* are atomic actions.

In LPS the only models that can be generated to make R true are ones that include the action *pay-penalty(100)*. But there are many other models that make R true, and that do not include the action *pay-penalty(100)*, but include instead an action a_1^* where a_1 = *pay-into-account(amount)* where *amount* ≥ 100.

Similarly, LPS cannot make rules *true* by deliberately making their conclusions *true* before their conditions are *true*. It is interesting to note that both the IFF proof procedure and the ALP proof procedure of [13] and [22] will generate the minimal models needed in both kinds of situations.

7 Related Work

7.1 Relationship with the situation calculus and event calculus

The minimal model ***M*** generated by LPS is both like a Kripke possible worlds semantic structure and like a minimal model of a logic program including the transition theory ***Tr*** and the situation calculus. In LPS, the situation calculus axioms are *true* in ***M,*** but are not used to generate ***M.*** They are, instead, an *emergent property* of the way the transition theory generates the sequence of database states. It is easy to see that the following variant of the situation calculus axioms are *true* in ***M***:

$$P(T+1) \leftarrow E(T, T+1) \wedge initiates(E, P, T)$$
$$P(T+1) \leftarrow P(T) \wedge \neg \exists E\ (E(T, T+1) \wedge terminates(E, P, T))$$

where P is an extensional predicate and E is an event. The second of these axioms is a variant of the frame axiom in second-order logic. It can be reformulated as a first-order, meta-logical axiom in the standard way. Reasoning explicitly, whether forwards or backwards, with these axioms is computationally explosive, and is an aspect of the frame problem that is often overlooked.

The situation calculus ontology of global states (or situations) has a natural correspondence with the database states of LPS. However, the event calculus axiom:

$$P(T2) \leftarrow E(T, T+1) \wedge initiates(E, P, T) \wedge \neg \exists E, T'\ (E(T', T'+1) \wedge terminates(E, P, T') \wedge T \leq T' \wedge T' \leq T2)$$

is also an emergent property of the LPS operational semantics. This follows from the fact that, in the special case in which time is identified with global states, the situation calculus and event calculus represent the same fluent relationships [25].

The use of destructive assignment, as in LPS, to implement the frame axiom as an emergent property, can be exploited for other applications, such as planning, provided only one state is explored at a time. In particular, for classical planning applications, the LPS approach can be generalised to store the complete history of actions and events leading up to a current database state. The database can be rolled back to reproduce previous states, and rolled forward to generate alternative databases states. However, these possibilities are topics of research for the future.

7.2 Other Related Work

LPS provides an agent framework that combines a model-theoretic semantics with a state-free syntax and a database maintained by destructive assignment. To the best of our knowledge, this combination is novel. Most agent frameworks have an operational semantics, but no declarative semantics. Some logic-based frameworks like Golog, ALP agents and KGP [10] have a model-theoretic semantics, but represent the environment using time or state and manipulate the representation using the situation or event calculus. Metatem [6], on the other hand, is a logic-based agent language with a Kripke semantics for modal logic sentences resembling production rules. Because of the Kripke-like semantics of LPS, it would be interesting to explore a similar modal syntax and semantics for LPS.

Costantini and Tocchio [3] also employ a logic programming approach with a similar model-theoretic semantics, in which external and internal events transform an initial agent program into a sequence of agent programs. The semantics of this

evolutionary sequence is given by the associated sequence of models of the sequence of programs. In LPS, this sequence is represented by a single model.

FLUX [15] is a logic programming agent language with several features similar to LPS, including the use of destructive assignment to update states. In FLUX, these states are not stored in a database as in LPS, but in a reified, list-like structure. FLUX employs a sensing and acting cycle, which it uses to plan and execute plans for achievement goals.

Thielscher [17] provides a declarative semantics for AgentSpeak by defining its cycle and procedures by means of a meta-interpreter represented as a logic program. Like LPS, the resulting agent language incorporates a formal transition theory. However, unlike LPS, the language does not distinguish between different kinds of procedures, according to their different functionalities. LPS, in contrast, distinguishes between reactive rules, planning clauses, macro-actions and ramifications, representing different kinds of AgentSpeak-like procedures in different ways. On the other hand, the agent architecture of Hayashi et al. [18] separates the representation of reactive rules and planning clauses, as in LPS.

There is also related work, combining destructive assignment and model-theoretic semantics in other fields, not directly associated with agent programming languages. EVOLP [1], in particular, gives a model-theoretic semantics to evolving logic programs that change state destructively over the course of their execution. Several other authors, including [7, 16] obtain a model-theoretic semantics for event-condition-action rules in active database systems, by translating such rules into logic programs with their associated model theory.

Perhaps the system closest to LPS is Transaction Logic [2], which employs a destructive database and gives a Kripke-like semantics for transactions (which are similar to macro-actions), represented in a state-free syntax. *TR* Logic also gives a semantics to reactive rules, which involves translating them into transactions. In LPS, the Kripke-like semantics is transformed into a single situation-calculus-like model, in the spirit of Golog. This transformation makes it possible to apply the general-purpose semantics of ALP to the resulting minimal model. In contrast, the semantics of *TR* Logic and Golog are defined specifically for those languages.

8 Future Work

Because LPS is based on the ALP agent model and the ALP model is more powerful than LPS, it would be interesting to extend LPS with some additional ALP agent features. These features include: partially ordered plans, more complex constraints on when actions should be performed and when fluent goals should be achieved, concurrent actions, conditionals in the conditions of clauses, active observations, a historical database of past actions and observations, abduction to explain observations that are fluents rather than events, and integrity constraints that prohibit actions rather than generate actions.

It would also be useful to study more closely the relationship between LPS and other agent models with a view to using the LPS approach to provide those languages with model-theoretic semantics. In addition, because the LPS cycle can be viewed as a model generator, which makes the reactive rules *true*, it would be interesting to explore the relationship with model checking and model generation in other branches of Computing.

Acknowledgments. We are grateful to Ken Satoh, Luis Moniz Pereira, Harold Boley, Thomas Eiter and Keith Stenning for helpful discussions, and to the anonymous reviewers for their helpful comments.

References

1. Alferes, J., Leite, J., Pereira, L.M., Przymusinska, H. & Przymusinski, T.:Dynamic Updates of Non-Monotonic Knowledge Bases, J. of Logic Programming 45(1-3):43-70 (2000)
2. Bonner and M. Kifer.: Transaction logic programming. In Warren D. S., (ed.), Logic Programming: Proc. of the 10th International Conf., 257-279 (1993)
3. Costantini, S. and Tocchio, A.: About Declarative Semantics of Logic-Based Agent Languages, Dalt 2005, LNAI 3904, Baldoni, M. et al (eds.), 106-123 (2006)
4. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A Common Semantics Basis for BDI Languages, ProMAS, LNAI 4908, Dastani, M. et al (eds.) Springer-Verlag Berlin Heidelberg, 124-139 (2008)
5. van Emden, M. and Kowalski, R.: The Semantics of Predicate Logic as a Programming Language, in JACM, Vol. 23, No. 4, 733-742 (1976)
6. Fisher, M.: A Survey of Concurrent METATEM - The Language and its Applications. Lecture notes in computer science, 827, Springer Verlag (1994)
7. Flesca, S. and Greco, S. Declarative Semantics for Active Rules. Theory and Practice of Logic Programming **1** (1): 43-69, (2001)
8. Fung, T.H. and Kowalski, R. : The IFF Proof Procedure for Abductive Logic Programming. J. of Logic Programming (1997)
9. Kakas,T., Kowalski, R., Toni, F.:The Role of Logic Programming in Abduction, Handbook of Logic in Artificial Intelligence and Programming 5, Oxford University Press, 235-324 (1998)
10. Kakas, A., Mancarella, P., Sadri, F., Stathis, K., Toni, F.: Computational Logic Foundations of KGP Agents. Journal of Artificial Intelligence Research. 33, 285-348 (2008)
11. Kowalski, R.: Predicate Logic as Programming Language, in Proceedings IFIP Congress, Stockholm, North Holland Publishing Co., 569-574 (1974)
12. Kowalski, R. and Sadri, F.: From Logic Programming Towards Multi-agent Systems, Annals of Mathematics and Artificial Intelligence, Volume 25, 391-419 (1999)
13. 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)
14. Reiter, R.: Knowledge in Action. MIT Press (2001)
15. Thielscher, M.: FLUX: A Logic Programming Method for Reasoning Agents, Theory and Practice of Logic Programming, 5(4-5), 533-565 (2005)
16. Zaniolo, C. A Unified Semantics for Active and Deductive Databases, Procs. 1993 Workshop on Rules In Database Systems, RIDS'93, Springer-Verlag, 271-287 (1993)
17. Thielscher, M., Integrating Action Calculi and AgentSpeak. In Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), Lin, F and Sattler, U. (eds.), Toronto (2010)
18. Hayashi, H., Tokura, S., Ozaki, F., Doi. M.: Background Sensing Control for Planning Agents Working in the Real World. International Journal of Intelligent Information and Database Systems, Inderscience Publishers, 3(4): 483-501 (2009)
19. Rao, Anand S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. Proceedings of Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96) (1996)

20. Kowalski, R. and Sergot, M.: A Logic-based Calculus of Events. In New Generation Computing, Vol. 4, No.1, 67-95 (1986). Also in The Language of Time: A Reader (eds. Inderjeet Mani, J. Pustejovsky, and R. Gaizauskas) Oxford University Press (2005)
21. 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), 200-218 (2010)
22. Kowalski, R.: Computational Logic and Human Thinking: How to be Artificially Intelligent, To be published by Cambridge University Press
23. Wooldridge, M.: An Introduction to MultiAgent Systems, John Wiley & Sons Ltd (2009)
24. Nau, D., Cao, Y., Lotem, A., Munoz-Avila, H.: SHOP: Simple Hierarchical Ordered Planner, Proc. of the 16th International Joint Conference on Artificial Intelligence (IJCAI), 968-973 (1999)
25. Kowalski, R. and Sadri, F.: The Situation Calculus and Event Calculus Compared, Proc. of International Logic Programming Symposium (ILPS) 94, MIT Press, 539-553 (1994)
26. Rao, A. S., Georgeff, M. P.: Modeling Rational Agents within a BDI-Architecture, Proc. of the 2nd International Conference on Principles of Knowledge Representation and Reasoning, 473-484 (1991)