An Agent Language with Destructive Assignment and Model-Theoretic Semantics

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 an action 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 assignment 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

Practical agent languages, many of which were originally inspired by the use of modal logic specifications of an agent's Beliefs, Desires and Intentions, have largely abandoned their original model-theoretic semantics in favour of operational semantics. They employ procedural representations and perform destructive assignment on "beliefs" that represent the current state of the agent's environment.

ALP (abductive logic programming) agents [12] have both an operational semantics and a model-theoretic semantics. However, they represent the agent's observations in a non-destructive database and explicitly represent and reason about time or state, using a formal action theory such as the event calculus, with the consequent inefficiencies of reasoning with explicit frame axioms.

In this paper we present a language, LPS, that combines a declarative semantics based on ALP with the features of practical agent languages, including the use of destructive assignment and a syntax that does not refer to time or state. The semantics of LPS can be viewed in terms of Kripke possible world structures, as in Transaction (*TR*) Logic [2]. However in *TR* Logic, the truth of sentences is defined along paths of possible worlds. In LPS, the possible worlds are combined into a single model with state arguments in the spirit of the situation calculus and Golog [14].

The database is structured as a deductive database, with extensional predicates that are represented explicitly and with intentional predicates that are defined by logic programs. Actions and observations are structured by means of an action theory that defines the preconditions of actions and the effects of actions and external events on the extensional predicates of the database. Intentional predicates are modified as ramifications of changes to the extensional predicates.

The frame problem is avoided by employing destructive change of state, without the use of frame axioms. The inefficiencies of planning from first principles are avoided, by using plan libraries to achieve intended consequences of actions, and by using the action theory only to transform one state of the database to the next, implementing consequences of the agent's actions and external observations.

In contrast with many agent languages, TR Logic and Golog, but as in ALP agents, LPS highlights the distinction between maintenance goals (or reactive rules), represented as integrityconstraints, and beliefs, represented as logic programs. The approach is based upon our earlier attempt to combine similar features of production systems with the model-theory of ALP [12]. We retain the name LPS, introduced in an earlier paper [13], and which stands for Logic-based Production System language, because we treat production rules and agent plans in the same way.

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 programs, SLD resolution and the minimal model semantics of Horn clauses.

1.1 Motivating Examples

The vocabulary of LPS includes both ordinary stateless predicates, as well as *fluents*, which are extensional and intensional predicates, and *actions*, which are atomic actions, macro-actions, and external events, which are observable by the agent. The semantics (or internal syntax) of a fluent P has an additional argument $P(T)$, indicating that P holds in the state T (or at the time T). Atomic and macro actions A have two additional arguments $A(T_1, T_2)$, indicating that the action A takes place from T_1 to T_2. The semantics of atomic actions and events A happening in the transition from state T to $T+1$ is given by $A(T, T+1)$.

The surface syntax of LPS does not have explicit state arguments. Instead, as well as the ordinary conjunction $\wedge$, it has two other conjunctions whose meaning is defined in terms of states. The formal syntax and semantics will be given in Section 3. But, in the meanwhile, the semantics of the two conjunctions in the following examples can be understood as follows:

$P : Q$, where both P and Q are fluents, means $P(T) \wedge Q(T)$.
$P : A$, where P is a fluent and A is an action means $P(T_1) \wedge A(T_1, T_2)$.
$A : P$, where A is an action and P is a fluent means $A(T_1, T_2) \wedge P(T_2)$.
$P ; Q$, where both P and Q are fluents, means $P(T_1) \wedge Q(T2) \wedge T_1 \leq T_2$.
$P ; A$, where P is a fluent and A is an action means $P(T_1) \wedge A(T_2, T_3) \wedge T_1 \leq T_2$.
$A ; B$, where both A and B are actions, means $A(T_1, T_2) \wedge B(T_3, T_4) \wedge T_2 \leq T_3$.

Below we illustrate our approach by giving examples formalized in the LPS language.

Example 1: We consider an online shopping scenario, similar to the running example produced by the W3C RIF Working Group 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:

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

The 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 by an offer of a promotional item. A gold customer is welcomed by 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).

The semantics (i.e. the state-based translation) of the first reactive rule and the first macro-action definition are:

login(X, T-1, T) ∧ customer(X, T) → welcome(X, T1, T2) ∧ T≤T1.
welcome(X, T, T1) ← status(X, new, T) ∧ promotional-item(Y, T) ∧
offer(X, Y, T, T1).

Example 2: The following is a reformulation in LPS of an example given in [4].
Reactive Rule: If a room is dirty clean it
is-dirty(Room) → clean(Room).
Macro-actions definitions:
clean(Room) ← goto(Room); vacuum(Room).
goto(Y) ← pos(Y).
goto(Y) ← pos(X) : different(X, Y) : adjacent(X, Z): step(X, Z); goto(Y).

Here *is-dirty* and *pos* are extensional predicates, *adjacent* and *different* are state-independent predicates, *vacuum* and *step* are atomic actions, and *clean* and *goto* are macro-actions. The action *step(X,Y)* causes a change in location and the action *vacuum(Room)* causes a change in the status of the *Room* via the *action theory*:

terminates(step(X,Y), pos(X)) and *initiates(step(X,Y), pos(Y)),*
terminates(vacuum(Room), is-dirty(Room)) and
initiates(vacuum(Room), is-clean(Room)).

The semantics of the last macro-action definition is:

[1] http://www.w3.org/2005/rules/wiki/RIF_Working_Group visited in July 2009

goto(Y, T1, T3) ← pos(X, T1) ∧ different(X, Y) ∧ adjacent(X,Z) ∧ step(X, Z, T1, T1+1) ∧ goto(Y, T2, T3) ∧ T1+1≤T2.

The LPS operational semantics (the LPS cycle) works as follows: The condition *is-dirty(Room)* of the reactive rule is checked against a database that represents the current state of the environment. For all instantiations σ for which the condition is true, the goal *clean(Room)*σ is added to the agent's goals. Each goal is then planned for by planning rules or macro-actions, the resulting atomic actions are executed, and each such execution (destructively) updates the database. In general, the planning and action executions can be interleaved, provided any ordering dictated by the connectives is respected. The predicate *pos* acts as a *guard* in the last two macro-action clauses, checking the agent's current location and directing the agent towards the next action. If all the goals are successfully planned for and the atomic actions are successfully executed the agent would have traversed a sequence of states the totality of which corresponds to a (minimum) model in which the reactive rule is true.

Example 3: The following is a reformulation in LPS of another example in [4], which involves buying a gift. According to the scenario in [4], first the agent checks what gifts are available in Harrods, and forms a plan to go to Harrods and purchase the gift. Then for some reason this plan does not succeed 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, as follows:

Planning rule: *have(Gift) ← sells(harrods, Gift): buy(Gift).*
Macro-action definitions:
buy(Gift) ← goto(harrods); purchase(Gift, harrods).
buy(Gift)←online(Store):sells(Store,Gift):goOnline(Store);purchase(Gift, Store).
The database contains the fact: *online(dell).*

The LPS operational semantics is neutral with respect to the search strategy used to explore the search space, and the "conflict resolution strategy" used to select an action to execute. To obtain the behavior of the scenario described in [4], these strategies would need to try the macro-action rules 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 execute the alternative action (*goOnline(dell)*).

Here is an alternative, more flexible formalization using only planning rules:

have(Gift)←is-Store(Store): sells(Gift, Store): goto(Store); purchase(Gift,Store).
have(Gift)←online(Store):sells(Gift,Store):goOnline(Store);
purchase(Gift,Store).

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 condition-action-and-goal rules. Both

production systems and agent languages manipulate a database of facts or *beliefs*, 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, condition-action-and-goal rules, called *plans* in agent languages, provide two main functions. Arguably their primary function is as *reactive rules*, to react to changes in the database, verifying that the condition holds and adding the corresponding goals and actions either to the database or the stack of goals. However, in practice they often function as *goal-reduction rules*, to match a current goal with one of the conditions of a plan, verify the other conditions of the plan, and add the corresponding goals and actions to the database or stack of intentions.

LPS borrows from production systems and agent languages their state-free syntax and their destructively changing 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* (or *policies*) 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 performed on the internal database 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. Additions and deletions are not explicit actions, but are consequences of an action theory. It uses destructive change of state to deal with the computational aspects of the frame problem.

In production systems and agent languages, when the conditions of more than one condition-conclusion rule hold, a choice needs to be made between the different conclusions. 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 it fails. In ALP and LPS, when the rules are interpreted as beliefs represented as logic programming clauses, the choice is dealt with by the selection function and search strategy. When the rules are interpeted as maintenance goals represented by integrity constraints, ***all*** maintence goals must be made true, by making their conclusions true whenever their conditions are true.

However in LPS, an analogue of conflict resolution is performed when the agent decides which action to execute. In ALP agents, we have explored the use of Decision Theory for this purpose. However, in LPS we assume that the choice is made by the

selection and search strategies, subject to the constraint that no action is selected if there are other actions that need to be executed earlier.

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*. In LPS, the abducible atoms represent actions and events, and the integrity constraints represent reactive rules (or policies).

In LPS, we use integrity constraints for reactive rules and restrict them to the form *condition* $\rightarrow$ *conclusion,* where *condition* and *conclusion* are conjunctions of atoms, and all the variables occurring in *condition* are universally quantified over the implication, and all variables occurring only in the *conclusion* are existentially quantified over the *conclusion*. For simplicitly, we restrict logic programs to Horn clauses [11]. This restriction has the advantage that Horn clauses have a unique minimal model [5]. The restriction can be relaxed in various ways, as we will discuss later.

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

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 semantics for LPS. In LPS, the analogue of the minimal model of $L \cup \Delta$ is the sequence of database states extended by the logic programming component of LPS. The analogue of *C* and ***IC*** being true in the minimal model is the truth of the initial goals and reactive rules.

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

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

database can be viewed as a Kripke-like model, transformed into a single situation-calculus-like model.

3 LPS Language – Informal Description

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

3.1 The Database

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

The databases $\boldsymbol{W_i}$ represent the agent's beliefs about the current state of the environment. These correspond to the extensional predicates of a deductive database, e.g. *customer(john-smith), spent-to-date(john-smith, 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 the facts that are not affected by the transformation persist without being copied explicitly from one state to the next.

In addition to extensional predicates, which represent database states explicitly, there are intentional predicates defined by clauses $\boldsymbol{L_{ram}}$. 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 ordinary logic programming clauses in $\boldsymbol{L_{stateless}}$. For example: *similar(X, Y)* ← *cd(X)* ∧ *dvd(Y).*

3.2 The Action Theory

State transitions are defined by a set of *action clauses* $\boldsymbol{A}$. The clauses in $\boldsymbol{A}$ are divided into clauses $\boldsymbol{A_{pre}}$ defining the preconditions and $\boldsymbol{A_{post}}$ defining the post-conditions of atomic actions. These have the form:

initiates(a, p) ← *init-conditions*
terminates(a, p) ← *term-conditions*
precondition(a, q) ← *pre-conditions*

where *a* represents an atomic action, *p* represents an extensional predicate and *q* represents an intensional, extensional or state-independent predicate. The first two types of clauses are in $\boldsymbol{A_{post}}$, and the last type of clause is in $\boldsymbol{A_{pre}}$. The conditions *init-conditions and term-conditions* are qualifying conditions, and together with *pre-conditions* are formulas that are checked in the current state. For example:

initiates(take-payment(X, ID, Value), spent-to-date(X, New)) ←
spent-to- date(X, Old) ∧ *New = Old + Value.*
terminates(take-payment(X, ID, Value), spent-to-date(X, Old)) ←
spent-to-date(X, Old).

An action *a* can be executed in state $\boldsymbol{W_i}$ provided that all of its precondions hold. This is determined by using the action theory to identify all the predicates *q* that should

hold, and then checking that all such q do indeed hold in the current state $\boldsymbol{W_i}$ extended by means of the intensional and stateless predicates, as determined by $\boldsymbol{L_{ram}}$ and $\boldsymbol{L_{stateless}}$. Not every action needs to initiate or terminate database facts. In particular, the LPS agent can execute external actions, which have no impact on the database.

For simplicity and uniformity, we treat observations as external events that initiate and terminate fluents. Their postconditions are included in $\boldsymbol{A_{post}}$. For example:

initiates(login(X), logged-on(X)) *terminates(logout(X), logged-on(X)).*

Because observations only happen if they can happen, there is no need to include their preconditions in $\boldsymbol{A_{pre}}$. It is important to note that action 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 clauses for planning.

3.3 Goals

In addition to the changing state of the database, the LPS operational semantics maintains an associated changing set of goal clauses $\boldsymbol{G_i}$, each of which can be regarded as a *partial plan* for achieving the initial goals $\boldsymbol{G_0}$ and the additional goals generated by the LPS cycle. These additional goals come from the conclusions of 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, stateless predicates and planning rules. Goals coming from different 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, as is typical of production systems. Informally speaking, the cycle succeeds in state n, if $\boldsymbol{G_n}$ contains the empty clause. However, the cycle does not terminate when it succeeds, because future observations may trigger future goals.

Initial goal clauses can contain actions, fluents, stateless predicates, and any of the logical connectives in the language, but not events. For example the goal clause

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

requires that a promotional item is determined and discounted by *20%*, and then the item and its new price are advertised.

3.4 Reactive rules

The set $\boldsymbol{P}$ of *reactive rules* has the same form *condition*$\rightarrow$ *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 the *condition* holds in the current state of the database W_i, and if it does, then the *conclusion* is added to every goal clause in $\boldsymbol{G_i}$. The *condition* can also include a single atom representing an atomic action executed in the last cycle and any number of atoms representing the last set of observations. Thus $\boldsymbol{P}$ can include the event-condition-action rules of active databases. For example:

take-payment(X, ID, Value) : *Value*$\geq$*50* $\rightarrow$*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 rules 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 and fluents. Macro-actions, defined by the set of clauses $\boldsymbol{L_{macro}}$, are like transactions in *TR* Logic and complex actions in Golog. Examples were given in section 1.1.

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 programs to be written to achieve extensional fluent goals explicitly. To ensure that the agent's beliefs are true with respect to the action theory that maintains the database, 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 action theory. $\boldsymbol{L_{plan}}$ represents such plans for achieving future states of the database. For example:

have(Gift)←is-Store(Store): sells(Gift,Store): goto(Store); purchase(Gift, Store).

Note that that the conclusions of plans represent the motivations of the agent's actions, in contrast with the action theory, which represents all the consequences of the agent's actions. For example, here the action theory may include clauses specifying other consequences of *purchase(Gift, Store)*, for example that the agent's financial resources will be reduced by the amount of the *Gift*.

Thus the planning clauses, together with the macro-action definitions implement planning from second principles, 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:

$p \leftarrow$ *init-conditions:* $pre\text{-}conditions_1: q_1: \ldots.: pre\text{-}conditions_n: q_n: a$

for every set of clauses

initiates(a, p) ← *init-conditions*

$precondition(a, q_1) \leftarrow pre\text{-}conditions_1$

….

$precondition(a, q_n) \leftarrow pre\text{-}conditions_n$

where the q_i are all the preconditions of *a.*

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.

We now define the LPS language formally.

4 LPS Language – Formal Description

The vocabulary of LPS is divided into fluent, action, and auxiliary predicates. The *fluent* predicates consist of extensional and intentional predicates. The action predicates consist of atomic, macro-actions, and observations of external events, in the sets *A, M* and *Ob* respectively. The *auxiliary* predicates consist of "ordinary" stateless predicates and the predicates *initiates, terminates, precondition* in the action theory. All these sets of predicates are mutually exclusive.

The LPS framework presented in this paper 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. Now we describe both the surface syntax and its semantics.

The surface syntax of all LPS components is defined in terms of *sequences* of predicates, where consecutive predicates are linked by : or ;. The *syntax of sequences* is defined recursively. We take the base case to be the empty sequence, which is also the empty clause, and we write it as *true*. If P is a predicate 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, stateless, atomic action, macro-action, event, extensional, intentional) predicate to mean an atom with such a predicate. The initial goal $\boldsymbol{G_0}$ is a set of goal clauses, each of which is a sequence with no events. Other goals $\boldsymbol{G_i}$, 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.

In the internal syntax, goal clauses are conjunctions of atoms, and the goals $\boldsymbol{G_i}$ represent disjunctions of goal clauses. These goals 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.

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

$\boldsymbol{L_{ram}}$ clauses have the form: $P \leftarrow P_1{:}P_2{:}\ldots :P_n$, $1 \leq n$, where P is an intensional predicate, each P_i is a fluent or stateless 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.

P 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 stateless predicate, A is an atomic action, and each Evt_i is an event. All Evt_i and A may be absent, in which case $1 \leq m$, otherwise $0 \leq m$.

A clauses have the forms :
$initiates(a, p) \leftarrow P_1{:}P_2{:}\ldots :P_n$
$terminates(a, p) \leftarrow P_1{:}P_2{:}\ldots :P_n$
$precondition(a, q) \leftarrow P_1{:}P_2{:}\ldots :P_n$
where each P_i is a fluent or stateless predicate, and $0 \leq n$.

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

true is a stateless predicate, $true^*$ is *true*.
If P is a stateless predicate, then P^* also written $P^*(T)$ is P.
If P is a fluent, then P^* also written $P^*(T, T)$ is $P(T)$.
If P is an atomic action or 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* $P^*(T_1, T2)$ is $P(T_1, T2)$.

Sequences have a similar semantics to predicates, either as stateless sequences or with two state arguments, which can be identical. The semantics of sequences is defined recursively, with the empty sequence having the semantics *true*.

Let P be a predicate and S a sequence, with semantics P^* and S^* respectively.
Let F be $P{:}S$, where neither P nor S is stateless.
Then $F^*(T_1, S_2)$ is $P^*(T_1, T_2) \wedge S^*(S_1, S_2) \wedge T_2 = S_1$.
Let F be $P;S$ where neither P nor S is stateless.
Then $F^*(T_1, S_2)$ is $P^*(T_1, T_2) \wedge S^*(S_1, S_2) \wedge T_2 \leq S_1$.
Let F be $P{:}S$ or $P;S$. Then:
If both P and S are stateless, then F^* is $P^* \wedge S^*$ and stateless.
If P is stateless and S is not, then $F^*(T_1, T_2)$ is $P^* \wedge S^*(T_1, T_2)$.
If S is stateless and P is not, then $F^*(T_1, T_2)$ is $P^*(T_1, T_2) \wedge S^*$.

The semantics of the initial goal $\boldsymbol{G_0}$ is the semantics of its sequences. All the variables in $\boldsymbol{G_0}$ (and subsequent $\boldsymbol{G_i}$) are existentially quantified.

The semantics of $\boldsymbol{L_{ram}}$ clauses $P \leftarrow P_1{:}P_2{:}\ldots{:}P_n$ is $P(T) \leftarrow P_1(T) \wedge P_2(T) \wedge \ldots \wedge P_n(T)$.
The semantics of $\boldsymbol{L_{macro}}$ clauses $M \leftarrow S$ is $M(T_1, T_2) \leftarrow S^*(T_1, T_2)$.
The semantics of $\boldsymbol{L_{plan}}$ clauses $P \leftarrow S$ is $P(T_2) \leftarrow S^*(T_1, T_2)$.

Note that these clauses do not contain any analogue of the frame axiom(s) in the situation calculus. Persistence (or inertia), which is formalised by frame axioms, is obtained in LPS implicitly through the maintenance of the current state of the database, without the computational overheads of reasoning with frame axioms.

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

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 generally not accessible in the current state T of the database.

The semantics of $\boldsymbol{A}$ clauses:

initiates$(a, p) \leftarrow P_1{:}\ P_2{:}\ \ldots{:}\ P_n$ is *initiates*$(a, p, T) \leftarrow P_1(T) \wedge P_2(T) \wedge \ldots \wedge P_n(T)$
terminates$(a, p) \leftarrow P_1{:}\ P_2{:}\ \ldots{:}\ P_n$ is *terminates*$(a, p, T) \leftarrow P_1(T) \wedge P_2(T) \wedge \ldots \wedge P_n(T)$
precondition$(a, q) \leftarrow P_1{:}\ P_2{:}\ \ldots{:}\ P_n$ is *precondition*$(a, q, T) \leftarrow P_1(T) \wedge P_2(T) \wedge \ldots \wedge P_n(T)$

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

Note that these syntax and semantics impose the restriction that no two actions (whether from A or M) have the same pair of state arguments. This is because, for simplicity, the LPS operational semantics executes at most a single 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 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 $\boldsymbol{W^*_i}$ when we need to refer to facts containing explicit state arguments: $W^*_i = \{p(i) : p \in W_i\}$.

Actions and events 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 action *a* is *executable* in state $\boldsymbol{W_i}$ if and only if for every *precondition(a, q, i)* that holds in $\boldsymbol{W^*_i \cup A^* \cup L_{ram}^* \cup L_{stateless}}$,
q holds in $\boldsymbol{W^*_i \cup L_{ram}^* \cup L_{stateless}}$.

In the LPS cycle, when an 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*, as defined below.

Definition 3. A selection function is *safe* if and only if, when it selects an action, the action is ground (except possibly for state variables). A selection function is *timely* if and only if, when it selects an action *a(t, t+1)* in a goal clause *C*, then *C* contains no other atom which is earlier in the same sequence in *C*. 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 which is earlier in the same sequence in *C*.

Note that the selection function is not restricted to selecting predicates in the sequence in which they are written. Predicates 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*. We define range-restriction after the LPS cycle.

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 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.

The internal syntax of LPS clauses and rules includes inequalities (≤) between states. For the model-theoretic semantics we need a theory $\boldsymbol{L_{temp}}$ that defines the inequality relation. However, 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 in the goal clause are also true. So for implementation purposes we can assume the inequalities are deleted from the clauses and rules. This is equivalent to resolving inequalities with clauses in $\boldsymbol{L_{temp}}$, which always succeeds.

Definition 4. LPS cycle: Let **Max** be a bound on the number of resolution steps to be performed in each iteration of cycle. Given a range-restricted LPS framework $\langle \boldsymbol{W_0}, \boldsymbol{G_0}, \boldsymbol{A}, \boldsymbol{P}, \boldsymbol{L_{ram}}, \boldsymbol{L_{stateless}}, \boldsymbol{L_{macro}}, \boldsymbol{L_{plan}} \rangle$, a safe and timely selection function $\boldsymbol{s}$, a timely search strategy $\sum$, and a sequence of sets of observations $Ob_0, Ob_1, \ldots$, the LPS cycle determines a sequence of state transitions $\langle \boldsymbol{W_0}, \boldsymbol{G_0} \rangle, (Ob_0, a_0), \ldots, \langle \boldsymbol{W_i}, \boldsymbol{G_i} \rangle, (Ob_i, a_i) \ldots$where for all i, $0 \leq i$, Ob_i, a_i and $\langle \boldsymbol{W_{i+1}}, \boldsymbol{G_{i+1}} \rangle$ are obtained from Ob_{i-1}, a_{i-1} and $\langle \boldsymbol{W_i}, \boldsymbol{G_i} \rangle$ by the following steps:

LPS0. Let Ob_i be the set of observations made in this round of cycle. $\boldsymbol{W_i}$ is updated to $\boldsymbol{WO_i}$ as follows: $\boldsymbol{WO_i} = (\boldsymbol{W_i} - \{p: a \in Ob_i$ and *terminates(a, p, i)* holds in $\boldsymbol{W}^*_i \cup \boldsymbol{A}^* \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{stateless}}\}) \cup \{p: a \in Ob_i$ and *initiates(a, p, i)* holds in $\boldsymbol{W}^*_i \cup \boldsymbol{A}^* \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{stateless}}\}$.

LPS1. For every instance *condition* $\sigma \rightarrow$ *conclusion* σ of a rule in $\boldsymbol{P}^*$ such that *condition* σ holds in $\boldsymbol{WO_i}^* \cup \{a_{i-1}^*\} \cup Ob_{i-1}^* \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{stateless}}$, add *conclusion* σ to every clause in $\boldsymbol{G_i}$. Let $\boldsymbol{GP_i}$ be the resulting set of goal clauses.

LPS2. Using the selection function $\boldsymbol{s}$ and search strategy $\sum$, let $\boldsymbol{GPL_i}$ be a set of goal clauses, starting from $\boldsymbol{GP_i}$, derivable by SLD-resolution using the clauses in $\boldsymbol{WO_i}^* \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{plan}}^* \cup \boldsymbol{L_{macro}}^* \cup \boldsymbol{L_{stateless}}$ 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 $\boldsymbol{G_{i+1}} = \boldsymbol{GPL_i}$, $\boldsymbol{W}_{i+1} = \boldsymbol{WO_i}$, and a_i 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 literal is an executable action is generated within the maximum number, **Max**, of resolution steps.

LPS2.2.1 Then one such action *a(T,T+1)* in a goal clause C in $\boldsymbol{GPL_i}$ is chosen for execution by the search strategy $\sum$. Note that *a(T, T+1)* 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. Moreover, 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 a constant = i or a variable.

LPS2.2.2 The action *a(T,T+1)* is executed. If the action fails, then $\boldsymbol{G_{i+1}} = \boldsymbol{GPL_i}$, $\boldsymbol{W}_{i+1} = \boldsymbol{WO_i}$, and a_i is the *empty action* ϕ. If the action succeeds, then a_i is *a(i, i+1)*.
$\boldsymbol{G_{i+1}} = \boldsymbol{GPL_i} \cup \boldsymbol{C'}$, where $\boldsymbol{C'}$ is the resolvent of *C* with *a(i, i+1)*.
$\boldsymbol{W_{i+1}} = (\boldsymbol{WO_i} - delete(a)) \cup add(a)$ where
delete(a) = {*p: terminates(a, p, i)* holds in $\boldsymbol{WO}^*_i \cup \boldsymbol{A}^* \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{stateless}}$ }
add(a) = {*p: initiates(a, p, i)* holds in $\boldsymbol{WO}^*_i \cup \boldsymbol{A}^* \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{stateless}}$ }.

The LPS cycle 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 an empty clause and $\boldsymbol{GP_n = G_n}$.

Definition 6. An LPS framework $\boldsymbol{<W_0, G_0, A, P, L_{ram}, L_{stateless}, L_{macro}, L_{plan}>}$ is *range-restricted* if and only if all rules in $\boldsymbol{P}$ and all clauses in $\boldsymbol{A}$, $\boldsymbol{L_{ram}}$, $\boldsymbol{L_{stateless}}$, $\boldsymbol{L_{macro}}$, $\boldsymbol{L_{plan}}$ and $\boldsymbol{G_0}$ 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* ← *conditions* in $\boldsymbol{L_{ram}}$, $\boldsymbol{L_{stateless}}$, $\boldsymbol{L_{macro}}$, $\boldsymbol{L_{plan}}$ is range-restricted if and only if *conditions* is range-restricted and every variable in *conclusion* occurs in *conditions.*

A clause *conclusion* ← *conditions* in $\boldsymbol{A}$, where *conclusion* is *initiates(a, p*), *terminates(a, p),* or *precondition(a, p),* is range-restricted if and only if every variable in *p* occurs either in *conditions* or in *a.*

A rule *condition* → *conclusion* in $\boldsymbol{P}$ 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 $\boldsymbol{L_{temp}}$ 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 $\boldsymbol{S_n}$ of sentences $\boldsymbol{W_0^*} \cup \ldots \cup \boldsymbol{W_n^*} \cup \{a_0^*, \ldots, a_{n-1}^*\} \cup Ob_0^* \cup \ldots \cup Ob_{n-1}^* \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{temp}} \cup \boldsymbol{L_{macro}}^*$ is a Horn clause logic program. Therefore, $\boldsymbol{S_n}$ has a unique minimal model $\boldsymbol{M_n}$. This model is like a Kripke structure of possible worlds $\boldsymbol{M^i} = \boldsymbol{W_i} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}}$ embedded in a single model $\boldsymbol{M_n}$, where the actions and observations $\{(Ob_0, a_0), \ldots, (Ob_{n-1}, a_{n-1})\}$ determine the transition relation from one possible world to another.

6.1 Soundness

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

Definition 7. $\boldsymbol{L_{plan}}$ is *compatible* with $\boldsymbol{A}$ if for every clause in $\boldsymbol{L_{plan}}$ of the form $p \leftarrow S$ there exists an instance of a clause in $\boldsymbol{A}$ of the form $initiates(a, p) \leftarrow P_1 : P_2 : \ldots : P_n$ such that $S^* \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{temp}}$ entails $(P_1 : P_2 : \ldots : P_n)^*$.

It is easy to satisfy this condition, and all the examples in this paper, if done in full will have this property. Note that we can plan to achieve intentional atoms by combining such clauses in $\boldsymbol{L_{plan}}$ with clauses in $\boldsymbol{L_{ram}}$ and $\boldsymbol{L_{macro}}$.

Theorem. Given a range-restricted LPS framework $\boldsymbol{<W_0, G_0, A, P, L_{ram}, L_{stateless}, L_{macro}, L_{plan}>}$, a safe and timely selection function $\boldsymbol{s}$, a timely search strategy $\sum$, and a sequence of sets of observations $Ob_0, Ob_1, \ldots, Ob_{n-1}$, if $\boldsymbol{L_{plan}}$ is compatible with $\boldsymbol{A}$ 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 $\boldsymbol{P}^*$ 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 clauses, starting with some C_0 in $\boldsymbol{G_{0*}}$: C_0 ,…,C_i, ….,C_m = *true*, where 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 $\boldsymbol{P}^*$ such that *condition* σ holds in $\boldsymbol{WO_i}^* \cup \{a_{i-1}^*\} \cup Ob_{i-1}^* \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{stateless}}$.

2. C_{i+1} is obtained by SLD-resolution between C_i and some clause C in $\boldsymbol{WO_i}^* \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{plan}}^* \cup \boldsymbol{L_{macro}}^* \cup \boldsymbol{L_{stateless}}$ in LPS2, by resolution with a_j^* in LPS2.2.2, 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 $\boldsymbol{C_0}$ is true in $\boldsymbol{M_n}$. Together with the condition $\boldsymbol{GP_n} = \boldsymbol{G_n}$, the lemma also implies that all the rules in $\boldsymbol{P}^*$ 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 $\boldsymbol{WO_i}^* \cup \{a_{i-1}^*\} \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{macro}}^* \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{temp}}$ 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 $\boldsymbol{A}$.

This theorem is restrictive 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 $\boldsymbol{G_0}$ 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 $\boldsymbol{M_\omega}$ determined by the potentially infinite Horn clause program $\boldsymbol{W_0}^* \cup \ldots \cup \boldsymbol{W_n}^* \cup \ldots \{a_0^*, \ldots, a_n^*, \ldots\} \cup Ob_0^* \cup \ldots \cup Ob_n^* \cup \ldots \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}}^* \cup \boldsymbol{L_{temp}} \cup \boldsymbol{L_{macro}}^*$.

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 would hold for LPS: Given a minimal model $\boldsymbol{M}$ of some clause C_0 in $\boldsymbol{G_0}$ and of all the rules in $\boldsymbol{P}$, it might be hoped that there

would exist 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***. Unfortunately this is not always possible. The LPS cycle will not generate models that make rules true by making their conditions false. For example:

P: $q \rightarrow a$ ***A***: *terminates(b, q)* $\mathbf{W_0}$: {q}

Here *a* and *b* are actions. There is a minimal model corresponding to the sequence of actions *b, a*, but the LPS cycle can only generate the non-terminating sequence *a, a, …*

This problem can be dealt with in the manner of the IFF proof procedure, by replacing every reactive rule of the form $p : q \rightarrow a$ with rules of the form $p: q \rightarrow a \vee b \vee c$, where *b* and *c* are atomic actions such that *terminates(b, q)* and *terminates(c, p)*. We do not consider completeness further here for lack of space.

6.3 Relationship with the situation calculus and event calculus

The minimal model ***M*** generated by LPS is both like a modal possible worlds semantic structure and like a minimal model of the situation calculus represented as a logic program. Ignoring observations and simplifying the situation calculus representation, the frame axioms have the form:

$$P(T+1) \leftarrow P(T) \wedge A(T, T+1) \wedge \neg terminates(A, P, T)$$

for every extensional predicate *P*. In LPS, these axioms are true in ***M,*** but are not used to generate ***M.*** Instead of reasoning explicitly that most fluents *P* that hold in state *T* continue to hold in *T+1*, destructive assignment is used to update only those fluents explicitly affected by *A*.

The use of destructive assignment, as in LPS, to implement the frame axiom, 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 Related and Future 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 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 action 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 rules, 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 rules, 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 gives a Kripke-like semantics for transactions (which are similar to macro-actions), represented in a state-free syntax. T^R 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 T^R Logic and Golog are defined specifically for those languages.

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 valuable to explore the relationship with model checking and model generation in other branches of computing.

In this paper we have focused on the theoretical framework of LPS. However, the ultimate test of the framework is its value as a practical agent language. For this purpose, we are developing further enhancements and are experimenting with an implementation.

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

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)