Programming with Logic without Logic Programming

Robert Kowalski and Fariba Sadri

Department of Computing
Imperial College London
{rak,fs@doc.ic.ac.uk}

Abstract. We extend a logic-based agent and production system language, LPS, from the single agent to the multi-agent case in which agents coordinate their behaviour through the medium of a shared state, which is viewed as a model-theoretic structure. In the extended language MALPS, each agent can be viewed as a collection of ***reactive rules***: *conditions* → *conclusions*, which have both a procedural and a declarative reading.

In the operational semantics of the extended language, the shared state is maintained only in its current state. Agents query the current state to determine whether the *conditions* of any reactive rules are true in the state; and, if they are, the agents perform actions to make the *conclusions* true in the future. Atomic events, including the agents' own actions and any external events, destructively update the state, adding facts initiated by events and deleting facts terminated by events.

The language also has a model-theoretic semantics, in which the agents' actions are performed with the intention of generating a model that makes all of the reactive rules true. The language is non-modal, and the model consists of the entire sequence of states and state-transforming events, treated as a single Herbrand model, with the standard semantics of classical first-order logic.

Keywords: Coordination, model-theoretic semantics, LPS, MALPS

1 Introduction

Historically, the logic-based agent and production system language LPS [8,9] was conceived as an application of abductive logic programming [7] to represent and reason about an agent's interaction with a destructively changing state. In LPS, the environment is represented by means of atomic facts expressed in terms of extensional predicates, and logic programs are used to define intensional predicates, which are like views in relational databases. Abduction is used to generate atomic actions, which destructively transform one state into another. Logic programs are also used to define macro-actions, which generate sequences of queries and actions. Macro-actions are like transactions in Transaction Logic [3].

The logic programming component of LPS is responsible for its model-theoretic semantics: Given an initial state of the extensional database and an initial goal (to be achieved in future states), the task is to generate a sequence of atomic actions and resulting states such that the initial goal is true in the minimal model of a logic program associated with the sequence of actions and states, together with the definitions of the intensional predicates and macro-actions.

In addition to logic programs and the destructively updated state, LPS includes ***reactive rules***, which are like integrity constraints in relational databases, condition-action rules in production systems, event-condition-action rules in active databases [14], and plans in BDI agents [13]. These reactive rules have the logical form *conditions* $\rightarrow$ *conclusions*, with the informal meaning that whenever the *conditions* are true, then the *conclusions* will be true in the future.

In the operational semantics, an LPS agent queries the current state to determine whether the *conditions* of any reactive rules are true in the state; and, if they are, the agent performs actions that update the extensional predicates, to make the *conclusions* true in the future. In the model theoretic semantics the intentions of the agent's actions are to generate a model in which the initial goal and the reactive rules are all true.

In this paper, we show how LPS can both be simplified and be extended to a framework for multi-agents and their coordination. LPS is simplified by removing the logic programming component, and it is extended by distributing the reactive rules among a collection of agents, with the current state serving as a coordination medium. We call the resulting framework MALPS (for Multi-Agent LPS), The extension inherits its model-theoretic semantics from LPS: The task is to generate a model in which the initial goals of all the agents and reactive rules of all the agents are true. However, in the operational semantics, different agents can update the state concurrently. LPS and MALPS have prototype implementations.

Sections 2, 3 and 4 introduce MALPS by means of examples. The first example of a mining operation shows that the shared state can simply be the current state of the world. The second example is a translation into MALPS of the solution of the dining philosophers problem presented in [1]. The third example is a MALPS version of an airline information and reservation example used to illustrate the coordination language Shared Prolog [4].

Sections 5 presents the model-theoretic semantics of MALPS, and section 6 presents its operational semantics. Finally, section 7 compares MALPS with related work, and section 8 concludes.

2 Example: Mine Operation

In LPS/MALPS, a state can be a "slice" of the real world, a mental representation of the world, a database, or some combination of all three. In all of these cases, the state can be viewed as a model-theoretic structure represented by the set of atomic sentences that are true in that state.

The following example, from [6,10], is a multi-agent version of a teleo-reactive program [10], which monitors and controls some of the operations of a mine. The state records the current levels of methane and water in the mine, as well as whether the water pump is active, represented respectively by the atomic

predicates *methane-level(M, T), water-level (W, T),* and *pump-active(T).* The state also records any actions *alarm(T)* and *pump(T)* that occur at time *T*.

Here, for consistency with other applications of MALPS, we assume that states are updated discretely. The state is updated by sensors that record the methane and water levels in the mine, and by an alarm agent, which sounds the alarm when the methane level exceeds *critical*, and by a pump agent, which pumps the water when the water level exceeds *high.* Once the water pump is activated, it continues until it reaches the level *low* or the methane level exceeds *critical*. The pumping action is not performed if the methane level exceeds *critical*, because any electrical spark could cause an explosion.

Let ε be a small number. The reactive rules of the two agents are:

alarm agent: $methane\text{-}level(M, T) \wedge critical \leq M \rightarrow alarm(T') \wedge T < T' \leq T + \varepsilon$

pump agent: $methane\text{-}level(M, T) \wedge critical > M \wedge water\text{-}level(W, T) \wedge high < W \rightarrow pump(T') \wedge T < T' \leq T + \varepsilon$

$methane\text{-}level(M, T) \wedge critical > M \wedge water\text{-}level(W, T) \wedge low < W \wedge pump\text{-}active(T) \rightarrow pump(T') \wedge T < T' \leq T + \varepsilon$

Here *methane-level* and *water-level* are external events that represent readings of the methane and water level, respectively.

In general, variables in a reactive rule that are in the conclusions but not in the conditions are implicitly existentially quantified with scope the conclusion. All other variables are implicitly universally quantified with scope the entire rule.[1]

Agents' actions and external events cause the current state to be updated at the time of their occurrence. So in the rules above, the external events *methane-level(M, T)* and *water-level(W, T)* and the actions *alarm(T')* and *pump(T')* cause state transitions at times *T* and *T'*, respectively. In this example and in teleo-reactive programs more generally, actions are *durative*, in the sense that they continue as long as their conditions hold, and they are terminated when their conditions no longer hold. The effects of the actions are sensed in future states.

The operational semantics is illustrated by the following example, in which *critical*, *high* and *low* are 100, 20 and 10, respectively.

	methane-level (Level, Time)	*water-level (Level, Time)*	*pump-active (Time)*	*action(Time)*
$state_1$	*66 0*	*18 0*		
$state_2$	*77 0.1*	*20 0.1*		
$state_3$	*88 0.2*	*20.1 0.2*		*pump(0.22)*
$state_4$	*99 0.3*	*20.1 0.3*	*0.3*	*pump(0.35)*
$state_5$	*99 0.4*	*15 0.4*	*0.4*	*pump(0.46)*
$state_6$	*100 0.5*	*12 0.5*	*0.5*	*alarm(0.53)*

[1] Conditions and conclusions can also contain first-order queries with explicit quantifiers.

$state_7$	*110*	*0.6*	*18*	*0.6*		*alarm(0.61)*
$state_8$	*104*	*0.7*	*19*	*0.7*		*alarm(0.73)*
$state_9$	*98*	*0.8*	*19*	*0.8*		
$state_{10}$	*98*	*0.9*	*15*	*0.9*		

Assuming that the mine starts operating at time 0 in $state_1$ and ceases in $state_{10}$ at time 0.9, the set of all these atomic sentences, together with atomic facts corresponding to the inequalities in the activated rules, constitute a model in which all of the reactive rules are true. This model is a *Herbrand model*, in which an atomic sentence is true if and only if the sentence belongs to this set. This definition of truth can be extended from atomic sentences to arbitrary sentences of first-order logic (FOL) in the standard way. Moreover, the number of states can be infinite.

The operational semantics of LPS is sound with respect to the model theoretic semantics, but is incomplete, because as we have shown in [9] it cannot generate models in which a reactive rule can be made true by performing actions that make its conditions false. Nor can it proactively perform actions to make its conclusions true before its conditions become true.

3 Example: The Dining Philosophers Problem

The mining example is a set of distributed reactive rules interacting through a shared state, which can be viewed as a relational database or a Herbrand model. It contains no logic program, other than those used to define the relations $\leq$ and $<$. In general in LPS/MALPS, as in the coordination languages Gamma [2] and Shared Prolog [4], the reactive rules are the driving force; and, as in Shared Prolog, logic programs play only a secondary, supporting role.

The dining philosophers problem is a classic problem of concurrent computing, which illustrates the simplicity and power of the coordination model [5]. The representation presented below follows the solution presented in [1].

The initial state of the problem consists of five philosophers sitting around a circular table with a bowl of spaghetti in the middle of the table and five forks, one to the left and one to the right of each philosopher. There are also four meal tickets. Each philosopher alternates between thinking and eating. In order to eat, a philosopher needs a meal ticket and two forks. A philosopher can pick up a fork from the table if the adjacent philosopher is not using it.

In this example, the state can be viewed as a relational database, logically represented by a set of *ground* (variable-free) atomic sentences (also called *facts*). In [1], the initial state is a set of forks and a multiset of tickets, generated by a main program, which also generates the five philosophers as active tuples. In contrast, in the MALPS solution, the five philosophers are agents represented by reactive rules, and the initial state is represented by a set of facts without an explicit representation of time:

ticket(0), ticket(1), ticket(2), ticket(3),
available($fork_0$), available($fork_1$), available($fork_2$), available($fork_3$),

available(fork$_4$), adjacent(fork$_0$, philosopher$_0$, fork$_1$),
adjacent(fork$_1$, philosopher$_1$, fork$_2$), adjacent(fork$_2$, philosopher$_2$, fork$_3$),
adjacent(fork$_3$, philosopher$_3$, fork$_4$), adjacent(fork$_4$, philosopher$_4$, fork$_0$).

The C-Linda program in [1] for the i-th philosopher is extremely simple:

```
philosopher(int i)
{
   while(TRUE) {
     think();
     in("meal ticket"); in("fork", i); in("fork", (i+1)%5);
     eat();
     out("fork", i); out("fork", (i+1)%5); out("meal ticket");
   } }
```

Here (i+1)%5 represents addition modulo five, and converts (4+1) into 0.

LPS distinguishes between facts that persist in the database (also called *fluents*) and *events*, including both external events in the environment and agents' actions, which add and delete persistent facts. Events transform one state into another, and can be regarded as taking place instantaneously. In the case of states represented as databases, an *event* can be represented simply as a set (or logical conjunction) of atomic sentences of the form *add(fact, T)* and *delete(fact, T)* for every *fact* added or deleted by the *event* at time *T*.

LPS/MALPS does not have a direct analogue of *in*, which simultaneously reads and deletes a tuple in Linda. However, it is possible to define *in* as a macro-action by means of a logic program, for example:

$$in(Fact, T, T') \leftarrow holds(Fact, T) \wedge delete(Fact, T') \wedge T < T' \leq T + \varepsilon$$

This definition of *in* is not strictly necessary, because all atoms of the form *in(Fact, T)* in the conclusion of a reactive rule can simply be replaced by the three subgoals $holds(Fact, T) \wedge delete(Fact, T) \wedge T < T' \leq T + \varepsilon$.

Here *holds(Fact, T)* means that *Fact* is in the state/database at time *T*. As we will see later, in the operational semantics, to facilitate destructive updates, facts are represented in the database/state without time or state parameters. However, the time or state parameter is necessary for the model-theoretic semantics. The representation *holds(Fact, T)* is interchangeable with the representation in which predicates have an extra time parameter *Fact(T)*.

Using this definition of *in*, we can write a program that is similar to that of [1]. However, to avoid problems with the semantics, we add a condition *add(hungry(philosopher$_i$), T)*, which specifies more precisely when and for how long the philosophers engage in dining:

philosopher$_i$:
$add(hungry(philosopher_i), T) \rightarrow think(T_1, T_2) \wedge in(ticket(N), T_3, T_4) \wedge$

$adjacent(Fork, philosopher_i, Fork') \wedge in(available(Fork), T_5, T_6) \wedge$
$in(available(Fork'), T_7, T_8) \wedge eat(T_9, T_{10}) \wedge add(ticket(N),T_{11}) \wedge$
$add(available(Fork), T_{12}) \wedge add(available\ (Fork'), T_{13}) \wedge$
$T < T_1 < T_2 < T_3 < T_4 < T_5 < T_6 \wedge T_4 < T_7 < T_8 \wedge T' = max(T_6, T_8) \wedge$
$T' < T_9 < T_{10} < T_{11} < T_{12} < T_{13}$

adjacent is a stateless predicate, i.e. one whose extension does not change from one state to another. Note that the temporal constraints $T_4 < T_5 \wedge T_4 < T_7 \wedge T' = max(T_6, T_8)$ mean that the philosopher can pick up the adjacent forks as they become available, in any order. Note also that thinking and eating take time, determined perhaps by procedures encapsulated within the philosopher. Notice too that is possible to impose temporal constraints on the duration of actions, for example philosophers might be constrained to think or eat for no more than 5 time units. In the operational semantics, the simplification and solution of such temporal constraints can be dealt with by means of a specialised constraint solver.

Philosophically speaking, in this example, the event of becoming hungry can be understood as an update of the shared state. However, the fact that a state is shared does not mean that all agents have equal access to all data in the state. In particular, only the philosopher who becomes hungry would have access to the fact that the philosopher is hungry.

The explicit representation of time in LPS/MALPS clarifies the logical meaning of reactive rules, but its syntax is clumsy, and it distracts attention from the operational behaviour. We have defined a state and time free external syntax for LPS, which is closer to conventional imperative syntax, and which is more suggestive of the operational semantics. In this syntax, the reactive rules would be written using ";" for <. Here we introduce an additional notation, with $P \parallel Q$ denoting the conjunction of P and Q in any temporal order. Also $P : Q$ means $holds(P, T) \wedge Q(T') \wedge T < T' \leq T + \varepsilon$ when P is a query and Q is an action, and means $P(T) \wedge Q(T)$ when both P and Q are actions. The operators $:$ and $;$ can also be given suitable definitions when one of their operands is an atom with a stateless predicate, but we will ignore these details for lack of space.

Using this external syntax, the reactive rule can be written in the form:

$add(hungry(philosopher_i)) \rightarrow think;\ in(ticket(N))\ ;\ adjacent(Fork,\ philosopher_i,\ Fork') : (in(available(Fork)) \parallel in(available(Fork'))) \ ;\ eat \ ;\ add(ticket(N)) \ ;\ add(available(Fork)) \ ;\ add(available\ (Fork'))$

4 Airline Flight Reservation

The following example is an MALPS version of a Shared Prolog program [4] for coordinating an airline company's flight reservations. In this example, the database records the number of seats available on flights by means of the atomic predicate *available(Flight, Seats).*

In the flight reservation example, clients can access the flight reservation system only through travel agencies, which simply submit requests to the airline company on their clients' behalf. In the spirit of Linda, we represent such client

requests and their corresponding responses by events that update the database, rather than as "read" and "write" in Shared Prolog.

Because agencies and the airline company have limited resources, there can be delays in responding to requests. Here, in MALPS external syntax, is one of many possible ways of representing the reactive rules that govern the agencies' behaviour:

travel agent $agency_i$:

add(customer-request(Customer, Request, $agency_i$))→
add(agency-request($agency_i$, Request))

add(company-reply($agency_i$, Request, Answer)) ∧
customer-request(Customer, Request) →
delete(customer-request(Customer, Request, $agency_i$)) ||
delete(company-reply($agency_i$, Request, Answer)) ||
add(agency-reply(Customer, $agency_i$, Request, Answer))

The company agent can be defined by a reactive rule and an auxiliary logic program defining the macro-action *verify-and-update*:

airline company agent:

agency-request(Agency, info(Flight)) →
available(Flight, Seats) :
add(company-reply(Agency, info(Flight), Seats)) :
delete(agency-request(Agency, info(Flight)))

agency-request(Agency, reserve(Flight, N)) →
available(Flight, Seats) :
verify-and-update(Flight, N, Seats, Answer) :
add(company-reply(Agency, reserve(Flight, N), Answer)) :
delete(agency-request(Agency, reserve(Flight, N)))

verify-and-update(Flight, N, Seats, reserve(Flight, N)) ←
N ≤ Seats : NewSeats is Seats – N :
delete(available(Flight, Seats)) :
add(available(Flight, NewSeats))

verify-and-update(Flight, N, Seats, false) ← N > Seats

The logic program defining the macro-action *verify-and-update* can be eliminated by replacing the subgoal *verify-and-update(Flight, N, Seats, Answer)* in the second reactive rule by the disjunction of the two alternative cases of the definition. The disjunction then needs to be distributed over the conjunction in the conclusion of the rule. This results in a reactive rule whose conclusion is a disjunction of conjunctions.

In MALPS, macro-actions are eliminated at the expense of introducing disjunctions into conclusions of reactive rules. This kind of elimination of macro-action definitions is not possible in the general case for logic programs containing recursion, because it can result in infinitely large disjunctions.

5 MALPS Language and Model-theoretic Semantics

5.1 The Language

The core language of MALPS consists of reactive rules of the form

conditions(X, Time) $\rightarrow$ *conclusions(X, Y, Time, OtherTimes)*

The *conditions* and *conclusions* have explicit temporal parameters, but these can be hidden in the external syntax, as exemplified in Sections 3 and 4. The *conditions* contain exactly one temporal variable *Time*, and the *conclusions* contain temporal variables *OtherTimes* that are all constrained in the conclusions to be later than the temporal variable in the *conditions*.

The *conditions* of a rule are a single query to the facts in the current state and to the events that caused the transition to the current state. *Events* include both events that are external to an agent and actions that the agent can perform. *Time* and all variables *X* that are not explicitly quantified in the conditions are universally quantified with scope the entire rule.

Queries are arbitrary formulas of FOL, possibly containing non-temporal quantified variables. These queries may contain stateless, auxiliary predicates such as inequality (and *adjacent* in the dining philosophers). For simplicity, these predicates are treated as defined extensionally by means of possibly infinitely many facts in every. In full LPS, these predicates, as well as intentional predicates and macro-actions are defined by means of finite logic programs. The extensions of the predicates are the ground facts that are true in an appropriate minimal model of the logic program defining them.

The *conclusions* of reactive rules are a disjunction of conjunctions, where each disjunct can be put in the form:

query(Y, Time') $\wedge$ *actions(Y, Time'')* $\wedge$ *rest(Y, Time', Time'', OtherTimes').*

query(Y, Time') is an FOL query (possibly empty, i.e. logically equivalent to *true*) only to the facts in the state at time *Time'*.

These disjuncts can be regarded as alternative conditional plans of actions to be made *true* at *OtherTimes* whenever the *conditions* become *true* at time *Time*. All variables *Y* and *OtherTimes* in the *conclusions* but not in the *conditions* that are not explicitly quantified in queries are existentially quantified with scope the *conclusions*.

actions(Y, Time'') is a possibly empty conjunction of atomic actions all with the same time parameter *Time''*, which is constrained in *rest(Y, Time', Time'', OtherTimes')* to be after *Time'*.

rest(Y, Time', Time'', OtherTimes') is a possibly empty conjunction of queries, atomic actions, and temporal constraints, where none of the times *OtherTimes'* are constrained to be earlier than *Time'* or *Time''*.

An agent may have *initial goals*, which are just like the *conclusions* of reactive rules. Therefore, all variables in an initial goal (not explicitly quantified in a query in the goal) are existentially quantified, and all time variables are constrained to be after the time of the initial state.

The reactive rules can be distributed among different agents, and the agents can perform actions in the attempt to generate a model in which all of their initial goals and their reactive rules are true. An action performed by one agent becomes an external event for other agents.

In most coordination languages, coordination among agents or processes is achieved by means of a shared data space. In MALPS, coordination is achieved by means of a shared state of the environment. The operational semantics treats individual states as sets of atomic sentences *P*, without explicit time or state parameters. However, the semantics of reactive rules treats such predicates as *holds(P, T), add(P, T), delete(P, T)* as meaning that *P* is *true* in the state at time *T*, and *P* is added/deleted in the transition to the state at time *T*.

Historically, the notion of state in LPS was influenced by knowledge representations in AI, in which a state is a symbolic representation of a time slice of the real or artificial world. LPS draws upon such formalisms as the situation calculus and event calculus, in which individual *states* (or situations) are identified with sets of atomic sentences, also called *fluents* or *facts*. In LPS/MALPS, we also view such states both as states of a relational database and as a Herbrand model.

5.2 The Model-theoretic Semantics

Viewing a state $\boldsymbol{S_i}$ (without explicit time) as a Herbrand model, an atomic sentence *A* is true in $\boldsymbol{S_i}$ if and only if *A* belongs to $\boldsymbol{S_i}$. The definition of truth can be extended from atomic sentences to arbitrary sentences of FOL in the standard way. However, in the case of universally quantified sentences, we use the *substitution interpretation*, which means that $\forall X\ p(X)$ is true if and only if $p(x)$ is true for all ground terms *x* from a suitably chosen "Herbrand universe". The definition of truth for negative sentences ($\neg P$ is true if and only if *P* is not true) can be regarded as a simple case of negation as failure

Viewing a state $\boldsymbol{S_i}$ as a relational database, a query can be any formula of FOL, A ground instance $q(x)$ of a *query* $q(X)$ with free variables *X* is an *answer* to the query if and only if $q(x)$ is true in $\boldsymbol{S_i}$.

A state transition from $\boldsymbol{S_i}$ to $\boldsymbol{S_{i+1}}$ takes place when one or more events take place. These events cause some new facts to hold in $\boldsymbol{S_{i+1}}$ and some old facts in $\boldsymbol{S_i}$ not to hold in $\boldsymbol{S_{i+1}}$. We say that the new facts are *initiated* and the old facts are *terminated*. All facts that were in $\boldsymbol{S_i}$ and not terminated continue to hold in $\boldsymbol{S_{i+1}}$.

Many, if not all, formalisms for reasoning about actions and change in AI, such as the situation calculus and event calculus, employ *frame axioms*, which state that a fact that held in $\boldsymbol{S_i}$ continues to hold in $\boldsymbol{S_{i+1}}$ if it is not terminated by the events that caused the state transition. Reasoning with frame axioms is combinatorially explosive. This combinatorial explosion is one aspect of the notorious *frame problem* in AI

Although the model-theoretic semantics of LPS/MALPS is based on the notion of states as models, the operational semantics does not use the frame axioms to perform state transitions. Instead, the operational semantics adds facts that are

initiated by events and destructively deletes facts that are terminated by events. Facts that are not affected by events simply persist without needing to reason explicitly that they persist.

In the model-theoretic semantics, the sequence of states $\boldsymbol{S_0}$, $\boldsymbol{S_1}$,…, $\boldsymbol{S_i}$,…. and sets of state transforming events $\boldsymbol{e_1}$,…, $\boldsymbol{e_i}$,…., augmented with an extensional definition ***Aux*** of any auxiliary predicates such as inequality, is also viewed as a single model-theoretic structure $\boldsymbol{S = Aux \cup S_0^* \cup e_1^* \cup S_1^* \cup e_2^* \cup \ldots \cup e_i^* \cup S_i^* \cup e_{i+1}^*}$ ….. Here the facts in $\boldsymbol{S_0^*}$, $\boldsymbol{S_1^*}$,…, $\boldsymbol{S_i^*}$,…. are time stamped versions of the facts in $\boldsymbol{S_0}$, $\boldsymbol{S_1}$,…, $\boldsymbol{S_i}$,…, and similarly the events in the sets $\boldsymbol{e_1^*}$,…, $\boldsymbol{e_i^*}$,…. are time stamped versions of the events in $\boldsymbol{e_1}$,…, $\boldsymbol{e_i}$,…., i.e.

p(t_i), or alternatively *holds*(*p*, t_i), is in $\boldsymbol{S_i^*}$ if and only if *p* is in $\boldsymbol{S_i}$.
add(*p*, t_i), or *delete*(*p*, t_i), is in $\boldsymbol{e_i^*}$ if and only if *add*(*p*), or *delete*(*p*), is in $\boldsymbol{e_i}$.

In ***S***, the event $\boldsymbol{e_i}$ occurs at time t_i and the state $\boldsymbol{S_i}$ starts at time t_i. The event $\boldsymbol{e_{i+1}}$ occurs at time t_{i+1} and the state $\boldsymbol{S_i}$ ends at time t_{i+1}. Therefore, all the facts in $\boldsymbol{S_i}$ are true in ***S*** from time t_i to time t_{i+1}.

Given an initial state $\boldsymbol{S_0}$, an initial goal $\boldsymbol{G_0}$, and a sequence of sets of external events $\boldsymbol{ex_1}$,…, $\boldsymbol{ex_i}$,….the *task* for an individual agent is to generate an associated sequence of sets of actions $\boldsymbol{a_1}$,…, $\boldsymbol{a_i}$,…. such that $\boldsymbol{G_0}$ and all of the agent's reactive rules are true in the resulting model $\boldsymbol{S = Aux \cup S_0^* \cup e_1^* \cup S_1^* \cup e_2^* \cup \ldots \cup e_i^* \cup S_i^* \cup e_{i+1}^*}$ …., where $\boldsymbol{e_i = ex_i \cup a_i}$ and $\boldsymbol{S_{i+1}}$ is obtained from $\boldsymbol{S_i}$ by performing the events/actions in $\boldsymbol{e_{i+1}}$.

6 MALPS Operational Semantics

The advantage of a model-theoretic semantics is that it provides a high-level specification that can be implemented in many different ways. FOL is a good example. Its semantics can be implemented by means of such diverse proof theories as Hilbert systems, natural deduction, sequent calculus and resolution. MALPS also has a relatively simple model-theoretic semantics, which admits a great variety of operational semantics. In this section, we illustrate an operational semantics for MALPS that is similar to that of some conventional coordination languages. However, many other operational semantics are also possible.

For the MALPS operational semantics sketched in this section, the shared state is a set of possibly infinite ground atomic sentences. All events are generated by the external environment or by the actions of agents.

The agents interact with one another and with the external environment by means of an observe-think-decide-act agent cycle. For simplicity, we assume that the agent cycles are all synchronised, so that each cycle begins and ends at the same time. We assume that the duration of each cycle is some fixed duration ε, which is short enough so that actions can be executed in a timely manner, and long enough so that all the necessary queries can be evaluated within a single cycle. Therefore, if *time* is the time at the beginning of a cycle, then *time* $+\varepsilon$ is the time at the end of the cycle, and the state remains unchanged throughout the period from *time* to *time* $+\varepsilon$.

The attempted actions generated by different agents at the end of a cycle can interfere with one another. To deal with such cases, we assume the existence of an arbiter that decides which actions succeed and which actions fail.

Each agent maintains its own private goal state, which logically is a conjunction of goals, each of which is a disjunction. Each disjunct is an existentially quantified conjunction of temporally constrained queries and actions. Operationally, each goal in the goal state can be regarded as a separate thread, independent from other threads in the goal state. In particular different threads do not share any variables. Each such goal/thread is a set of alternative conditional plans. Initially, each agent has at most only one goal in its goal state, which is logically an existentially quantified disjunction, and operationally a single thread.

To simplify the description of the operational semantics, it is useful to eliminate temporal constraints of the form $T \leq T'$ from the conclusions of conditional plans, replacing them by disjunctions $T = T' \vee T < T'$, distributing the disjunction over conjunction, generating two alternatives instead of one. To simplify the alternative containing $T = T'$, we replace every occurrence of T' in the alternative by T. To avoid over-constraining the times at which queries need to be evaluated and actions need to be executed, we do not allow concrete times t to occur in the form *holds(P, t)* or *action(t).* Instead they need to be written in the form *holds(P, T)* or *action(T),* where the time variable is constrained, in some such manner as $t \leq T \leq t + \delta$ for some suitable δ.

With these simplifying assumptions, it suffices to define the cycle for individual agents only. This cycle is similar to that of LPS:

Step 1. Let *time* be the time of the transition from the previous state $\boldsymbol{S_{i-1}}$ to the state $\boldsymbol{S_i}$, at the beginning of a cycle. The agent reasons forwards, evaluating the conditions of each one of its reactive rules:

$$conditions(X, Time) \rightarrow conclusions(X, Y, Time, OtherTimes)$$

to determine every instance *conditions(x, time)* of the conditions that is true in the current state $\boldsymbol{e_i}^* \cup \boldsymbol{S_i}^*$ viewed as a Herbrand model. For every such instance, it generates the corresponding instance *conclusions(x, Y, time, OtherTimes)* of the conclusion of the rule as a goal to be made true for some individuals *Y* at some *OtherTimes* in the future. Temporal constraints are simplified, and every such simplified conclusion is added to the agent's goal state as a separate thread.

Step 2. The agent chooses a thread and chooses an alternative plan in the thread for possible execution. It is the responsibility of the decision strategy to decide when to work on a thread and when to try which alternatives. Ideally, the decision strategy should be fair, so that each goal/thread has an opportunity to be solved. For this purpose, it is useful to view the threads as a collection of parallel processes that access and update the shared current state.

In theory, the decision strategy can employ any fair strategy for choosing alternatives in a thread. It can even try all alternatives in parallel. However, for many applications, it is often more efficient to prioritise the alternatives, and to try them one at a time. In a practical programming language, this would probably

be the default strategy, with the programmer using the order in which the alternatives are written to determine the order in which they are attempted.

Suppose the chosen plan has the form:

$$query(Y, Time') \wedge actions(Y, Time'') \wedge rest(Y, Time', Time'', OtherTimes')$$

Notice that the chosen plan can be either a new plan added in this cycle, or an old (possibly partially executed) plan left over from previous cycles.

Step 3. If *query(Y, Time')* is not empty, then the agent evaluates the query in the current state $\boldsymbol{e_i^*} \cup \boldsymbol{S_i^*}$. If the query is empty, then the agent proceeds to step 4.

Case 3a. If the query fails, then the cycle terminates with the empty action at time $time + \varepsilon$. However, the chosen plan is retained, because the query can be tried again in the future, for as long as the time variable *Time'* is not constrained to be in the past.[2]

Case 3b. If the query succeeds, with *n* answers $Y = y_1, y_2, \ldots y_n$. Then, for every such answer, a new plan of the form:

$$actions(y_i, Time'') \wedge rest(y_i, time, Time'', OtherTimes')$$

is generated, temporal constraints involving *time* are simplified, and the resulting simplified plan is added as a new alternative to the chosen thread. One such newly added alternative is selected to continue in step 4.

Step 4. The agent attempts to execute the actions: If the actions are empty, the cycle terminates with the empty action at time $time + \varepsilon$. If the actions are not empty, then suppose the selected alternative plan now has the form:

$$actions(y_i, Time'') \wedge simplified\text{-}rest(y_i, Time'', OtherTimes')$$

The agent submits all the actions $actions(y_i, Time'')$ to the arbiter at time $time + \varepsilon$.

For simplicity, assume that either all the actions succeed, or if one of them fails then they all fail. If they fail, then the cycle terminates in effect with an empty action at time $time + \varepsilon$. However, the chosen plan is retained, if the constraints on *Time''* allow the actions to be tried again in the future.

If the actions all succeed, then *Time''* is instantiated to $time + \varepsilon$. A new plan is generated:

---

[2] Simplification of temporal constraints can recognise that a plan is logically *false* and operationally unachievable, if the plan contains a subgoal of the form *holds(P, T)* or *action(T)* and a constraint of the form $T < time$, where *time* is the current time. The plan can then be deleted from the thread, because it can never be executed and made *true* in the future. If there are no alternatives left in a thread, because all of the alternatives are *false*, then the thread itself is logically *false*, the goal state is *false*, and the agent is a failure.

simplified-rest(y_i, time $+\varepsilon$, OtherTimes')

Temporal constraints involving *time* $+\varepsilon$ are simplified, and the resulting simplified plan is added as a new alternative to the chosen thread[3]. The cycle terminates at time *time* $+\varepsilon$.

However, if the resulting simplified plan is empty, and therefore logically equivalent to *true*, then, because the thread in which the plan occurs is logically the disjunction of all its alternatives, the thread itself is also logically equivalent to *true*. Because the goal state is logically the conjunction of all its threads, the thread can also be deleted from the goal state, whose truth value now depends only on the remaining threads.

Note that whether an action succeeds or fails in step 4, once an agent attempts to perform an action at a given time, there is no possibility of backtracking on the action and trying other alternatives at the same time or in the past. However, in theory, if there is sufficient time available within the cycle, other alternative plans could be chosen for evaluation, as in step 2. For simplicity, we ignore this possibility for now.

7 Related work

To the best of our knowledge, MALPS is the first attempt to provide a logic-based semantics for coordination languages. However, there are many other computing paradigms that use a similar notion of reactive rule as their core language construct. These include production systems, event-condition-action rules in active databases, BDI agent languages, and related agent languages, such as Metatem. LPS itself was motivated, in particular, by the goal of providing a logical semantics to condition-action rules in production systems.

Reactive rules are also the basic language construct of several coordination languages, including Gamma and Shared Prolog. It is interesting that Shared Prolog, along with several other logic programming based coordination languages, such as tao [12], were motivated by the Linda model of coordination, but do not have a logical semantics.

The model theoretic semantics of MALPS is similar to the possible world semantics of modal logic and of Metatem in particular. However, in Concurrent Metatem, agents communicate by message passing rather than by interacting through a shared data space.

In the possible world semantics of modal logics such as Metatem, possible worlds $\boldsymbol{S_i}$* are linked by accessibility relations, which are like state transitions in MALPS. However in MALPS, all of the possible worlds, as well as the state transitions $\boldsymbol{e_i}$*, are included in a single, classical, model theoretic structure $\boldsymbol{S}$. This

[3] The successful execution of the actions may instantiate some variables in the actions, not represented explicitly in the notation *actions(y_i, Time'')* $\wedge$ *simplified-rest(y_i, Time'', OtherTimes')*. This instantiation is a kind of feedback produced by the execution of the action, and is applied to the new plan.

simplifies the semantics and increases the expressive power of the language by making events first class objects.

The model theoretic semantics of MALPS is also similar to the possible world semantics of the non-modal language, Transaction Logic [], where transactions, which are like conditional plans, are given semantics in terms of paths in a possible worlds structure. As in LPS/MALPS, transactions are performed on a destructively updated database. The notion of FOL query in LPS/MALPS was partly inspired by queries in transaction logic.

States in LPS/MALPS are similar, not only to databases, but also to states in such AI knowledge representations languages as the situation calculus and event calculus. However, the frame problem is avoided by using destructive assignment to maintain only the current state. The situation calculus and similar AI approaches have also been used to give semantics to event-condition-action rules in active database systems [14]. However, these systems seem to inherit the frame problem of these AI approaches.

8 Conclusions

In this paper we presented a simplification and extension of the single agent language LPS to the mulit-agent language MALPS, in which agents interact among themselves and with the external environment through a shared state.

The simplification was to eliminate the logic programming component of LPS, highlighting the primary role of reactive rules. The extension included the use of the shared state as a coordination mechanism, and the distribution of the initial goals and reactive rules among different agents. The operational and model theoretic semantics were adapted from LPS, whose own model theoretic semantics was adapted in turn from the minimal model semantics of logic programming and its extension to abductive logic programming.

In MALPS, as a substitute for logic programs, we have assumed that auxiliary predicates are defined by possibly infinitely many facts. We also ignored the use of logic programs to define macro-actions. We believe that logic programs are a natural way to define both auxiliary predicates and macro-actions, but other formalisms are compatible with LPS/MALPS, provided they can be reduced to the case presented in this paper.

The MALPS operational and model-theoretic semantics allow the programming of open systems, in which agents can easily enter leave the system. To do so while respecting the model-theoretic semantics, it is necessary to ensure that the time variables in the conditions of reactive rules are restricted to the times that the agent is in to the system.

It is interesting to note that Carriero and Gelerneter [5] proposed the Linda model as a more natural model of parallelism and concurrency than the concurrent logic programming languages, which were adopted as the kernel language for the Japanese Fifth Generation Project. Arguably, the choice of concurrent logic programming, with its committed choice and lack of a true logical semantics, was one of the biggest problems of the Fifth Generation Project. Ironically perhaps, the logical semantics of MALPS presented in this paper suggests that a Linda-like

coordination model might have been a better choice.

In [5], Carriero and Gelernter argued that Linda is a "simpler, more powerful and more elegant" model of parallel programming than message-passing, concurrent object oriented programming, concurrent logic programming, and functional programming. Our experience with the development of MALPS leads us to share this view.

Acknowledgements

Many thanks to Chris Hankin and Paul Krause for helpful discussions about the topic of this paper. We are also grateful for the EPSRC internal Pathways to Impact funding that has supported the implementation of LPS and MALPS.

References

1. Arbab, F. , Ciancarini, P. , Hankin, C.L.: Coordination Languages for Parallel Programming, Parallel Computing, Vol:24 (1998)
2. Banatre, J.P. and LeMetayer, D.: The Gamma Model and its Discipline of Programming. *Science of Computer Programming*. (1990)
3. Bonner and M. Kifer.: Transaction logic programming. In Warren D. S., (ed.), Logic Programming: Proc. of the 10th International Conf., 257-279 (1993)
4. Brogi, A. and Ciancarini, P.: The concurrent language Shared Prolog. ACM Transactions on Programming Languages and Systems, 13(1): 99–123 (1991)
5. Carriero, N. and Gelernter, D.: Linda in Context. Communications of the ACM. Volume 32 Issue 4, (1989)
6. Hayes, I.J.: Towards Reasoning About Teleo-reactive Programs for Robust Real-time Systems, In Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems (SERENE), 87-94 (2008)
7. Kakas,T., Kowalski, R., Toni, F.:The Role of Logic Programming in Abduction, Hand book of Logic in Artificial Intelligence and Programming 5, Oxford University Press, 235-324 (1998)
8. 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)
9. Kowalski, R. and Sadri, F.: Abductive Logic Programming Agents with Destructive Databases, *Annals of Mathematics and Artificial Intelligence*, Volume 62, Issue 1, 129-158 (2011)
10. Kowalski, R. and Sadri, F.: Teleo-reactive Abductive Logic Programs, To appear 2012
11. Ludäscher, B., May, W. and Lausen, G.: Nested Transactions in a Logical Language for Active Rules. Logic in Databases 1996: 197-222 (1996)
12. Porto, A. and Vasconcelos, V.: Truth and action osmosis (the tao computation model). Coordination Programming: Mechanisms, Models and Semantics. 65-97 (1995)
13. Rao, A. S., Georgeff, M. P.: BDI Agents: From Theory to Practice, International Conference on Multiagent Systems - ICMAS , 312-319 (1995)
14. 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)