Teleo-Reactive Abductive Logic Programs

Robert Kowalski and Fariba Sadri

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

Abstract. Teleo-reactive (TR) programs are a variety of production systems with a destructively updated database that represents the current state of the environment. They combine proactive behaviour, which is goal-oriented, with reactive behaviour, which is sensitive to the changing environment. They can take advantage of situations in which the environment opportunistically solves the system's goals, recover gracefully when the environment destroys solutions of its goals, and abort durative actions when higher priority goals need more urgent attention.

In this paper, we present an abductive logic programming (ALP) representation of TR programs, following the example of our ALP representation of the logic-based production system language LPS. The operational semantics of the representation employs a destructively updated database, which represents the current state of the environment, and avoids the frame problem of explicitly reasoning about the persistence of facts that are not affected by the updates. The model-theoretic semantics of the representation is defined by associating a logic program with the TR program, the sequence of observations and actions, and the succession of database states. In the semantics, the task is to generate actions so that all of the program's goals are true in a minimal model of this associated logic program.

Keywords: teleo-reactive programs, abductive logic programming, production systems, LPS.

1 Introduction

Tele-reactive (TR) programs were introduced by Nils Nilsson in a technical report in 1992 [14] and an article [15] published in 1994. In [16], a TR program is characterized as "an agent control program that robustly directs the agent towards a goal in a manner that continuously takes into account the agent's changing perceptions of the environment."

These characteristics have contributed to a growing interest in TR programs in recent years. For example: Coffey and Clark [3] propose a BDI-style agent architecture that uses teleo-reactive plans in its plan library. Marinovic et al [13] use TR programs to represent workflows and policies in pervasive healthcare. Gordon and Logan [4] use TR programs to program game agents. Gubisch et al [5] use an architecture based on TR programs for mobile robot control and apply it to soccer robots. Broda and Hogger [23] present a systematic procedure for constructing TR programs.

TR programs are written and executed like ordered production rules:

$K_1 \rightarrow a_1$
……
$K_i \rightarrow a_i$
….
$K_m \rightarrow a_m$

The list is checked from the top, and the first rule whose conditions K_i are satisfied fires, and its action a_i is executed. In effect, the conditions K_i of the i-th rule implicitly include the negations of all of the conditions of the previous i-1 rules.

Actions a_i are atomic formulae, representing a primitive action, an invocation of another TR program, or a recursive invocation of the same program with different parameters. As we argue elsewhere [7, 24], rules in production systems have the syntax of logical formulae, but do not have a logical semantics. One of the main goals of this paper is to show how to translate TR programs into logical form, in such a way that the operational semantics of TR programs is sound with respect to a model-theoretic semantics.

One of the biggest challenges of the translation is to do justice to the fact that actions can be durative, in the sense that they are executed continuously, as long as their corresponding condition remains the highest true condition in the list. When the highest true condition changes the action also changes accordingly.

In a hierarchy of TR programs, in which one program calls another, the conditions of all the programs in the hierarchy are checked continuously. The action that is executed is the one associated with the highest true condition in the "highest program in the stack of called programs".

In many cases, the program is associated with an explicit goal, which is the first condition K_1 in the list, and the associated action a_1 is nil. A program has the *regression property* if whenever the action a_i of a rule $K_i \rightarrow a_i$ is executed then an earlier condition K_j ($j<i$) will eventually be satisfied (if the environment does not intervene).

Thus TR programs can combine proactive and reactive behavior. They share with purely reactive agents the ability to react to the changes in the environment, but they can do so within the context of an explicit goal.

Depending on how well the conditions K_1-K_m cover the possible situations that can arise, TR programs are robust, in the sense that even if the executions of some actions fail or if the environment undoes some of their desired effects, the program can continue to work towards its goal. TR programs are also opportunistic, in the sense that they can take advantage of situations in which the environment solves their goals without their participation.

Nilsson [16] proposes the use of TR programs as an intermediate layer, between lower-level programs "that use a short and fast path from sensory signals to effectors" and higher-level "systems that can generate plans consisting of a sequence of intermediate-level programs".

In this paper, we show how TR programs can be given a model-theoretic semantics, by embedding them in the higher-level framework of abductive logic programming agents (ALPA) [8]. For this we take inspiration from the logic-based production system and agent language LPS [9, 10], which is also embedded in ALPA, and which combines a declarative semantics with a destructively updated database.

Thus our approach provides both a practical operational semantics for TR programs in the ALPA framework and the means to reason formally about TR programs, in the manner of Hayes [6].

Whereas Hayes' semantics is based on duration and interval logics, our semantics for TR programs uses a representation of durative actions in terms of time points, and is defined in terms of the perfect model of a locally stratified program [17, 18].

As in the case of LPS, the model-theoretic semantics is compatible with an operational semantics that employs a destructively updated database, which represents the current state of the environment. As we argue elsewhere [10], the use of a destructively updated database is necessary to overcome the computational aspects of the frame problem.

This paper assumes familiarity with the basic concepts of logic programming. However, because it is written mostly in an informal style, the reader not familiar with logic programming should be able to understand the main ideas of the paper simply by focusing on the examples. A simple introduction to logic programming and the minimal model semantics of Horn clauses is included in the appendix.

2 Abductive Logic Programming (ALP), Abductive Logic Programming Agents (ALPA) and the Logic-based Production System and Agent Language (LPS)

In this Section we provide an overview of ALP, ALPA and LPS. ALP extends logic programming by allowing some predicates, ***Ab****,* the *abducibles* or *open predicates*, to be undefined, in the sense that they can occur in the conditions of clauses, but not in their conclusions. Instead, ground atoms in the abducible predicates can be assumed, but are constrained by a set ***IC*** of *integrity constraints*. Viewed in database terms, the open predicates are potential updates that can be added to the database, and the integrity constraints are used to monitor and constrain these updates.

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***.

Several alternative semantics and proof procedures have been defined for ALP. The semantics of ALP that we use in this paper is based on the model-theoretic semantics of [7]. The key feature of the semantics is that the integrity constraints ***IC*** and any initial goal ***G*** are required to be *true* in a unique minimal model determined by the logic program ***L*** extended by assumptions in the predicates ***Ab***. For this purpose, it suffices to assume that ***L*** is a locally stratified program, which determines a unique perfect model [18], in which case ***IC*** and ***G*** can be any sentences of first-order logic (FOL). Arguments for the minimal model semantics are presented in [7].

Definition. Given an ALP framework <***L****,* ***Ab****,* ***IC***> and goal ***G*** (which can be empty, equivalent to *true*), a *solution* of the goal is a set of atomic sentences Δ in the predicates ***Ab***, such that $G \cup IC$ is *true* in the perfect model of $L \cup \Delta$.

In classical abduction, the goal ***G*** is a set of observations ***Obs***, and a solution Δ is a set of assumptions that explain the observations. In ALP and ALPA, the goal ***G*** is

more commonly a future state of the environment, and a solution *Δ* is a set of partially ordered actions that achieve the desired state, provided the environment does not interfere. Therefore, in the general case, the set *Δ* can include both assumptions that explain observations and actions that achieve future states. However, for simplicity, in many applications of ALP and ALPA, including the application to TR programs, the observations are taken as "facts" that are dynamically added to ***L*** and that do not require any explanation.

Although in theory ***IC*** and ***G*** can be any sentences of FOL, it suffices, when ALP and ALPA are used for the semantics of LPS and TR programs, to restrict the syntax of integrity constraints in ***IC*** to conditionals of the form:

conditions → *conclusion*

where *conditions* is a conjunction of literals and *conclusion* is a conjunction of literals. All variables in such conditionals are universally quantified with scope the conditional, except for variables in the conclusion that are not in the conditions, which are existentially quantified with scope the conclusion. In the operational semantics, such conditionals are made *true,* by *reasoning forwards* to make the conclusion *true* whenever the conditions become *true*. Thus integrity constraints in the form of conditionals behave like production rules.

To embed LPS and TR programs in ALP, it suffices to restrict the syntax of goals ***G*** to goal clauses, as in logic programming. These goals have the same form as the existentially quantified *conclusions* of integrity constraints derived by forwards reasoning. In the operational semantics, such goals are made *true* by *reasoning backwards*, using logic programs to reduce goals to subgoals. A subgoal that is an atomic action can be made *true* by adding it to *Δ*.

ALP is used in ALPA for the thinking component of a BDI-like agent [19]. In ALPA, the logic program ***L*** represents the agent's beliefs, and ***G*** ∪ ***IC*** represents the agent's desires (or goals).

The thinking component of an ALP agent is a proof procedure, using forward and backward reasoning, embedded in an agent observe-think-decide-act cycle. The logic program ***L*** includes a deductive database that represents the agent's view of its environment.

In ALPA, the database is updated, as the result of observations and actions, by means of an action or event theory, such as the situation or event calculus [20, 11]. The updates are non-destructive, and via the event theory they *imply* the initiation of new facts and the termination of old facts. Moreover, they involve the computationally explosive use of frame or persistence axioms to reason that a fact persists if it is not directly affected by an update.

In contrast, production systems and most practical agent systems, employ a destructively updated database that represents only the current state of the world. They avoid the computational overheads of the frame problem by changing only those facts that are directly affected by an action or event. Facts that are not affected simply persist, without the need to reason that they persist.

LPS is based on the model-theoretic semantics and proof procedures of ALPA, but benefits from the computational advantages of employing a destructively updated database that represents only the current state of the world. The operational semantics represents facts in the database without an explicit time or state parameter, but the model-theoretic semantics is defined with respect to the entire sequence of database

states in which facts are time-stamped with explicit time parameters. This sequence is like a Kripke possible world structure in which the possible worlds (and the accessibility relation) are all combined into one model-theoretic structure.

In LPS, the model is the perfect model of the logic program:

> $\boldsymbol{P} \cup \boldsymbol{I} \cup \boldsymbol{Obs} \cup \boldsymbol{\Delta} \cup \boldsymbol{DB}$, where
>
> ***P*** is a locally stratified logic program, with explicit time parameters, which defines macro-actions in terms of sequences of atomic actions, other macro-actions, and queries to the database.
>
> ***I*** is a locally stratified logic program, with explicit time parameters, which defines the intensional and state-independent predicates of the deductive database.
>
> $\boldsymbol{Obs} = \boldsymbol{Obs_0} \cup \ldots \cup \boldsymbol{Obs_i} \cup \ldots$, where $\boldsymbol{Obs_i}$ is the set of time-stamped observations at time *i*.
>
> $\boldsymbol{\Delta} = \{a_0, \ldots, a_i, \ldots\}$ where a_i is the time-stamped action executed at time *i*.
>
> $\boldsymbol{DB} = \boldsymbol{DB_0} \cup \ldots \cup \boldsymbol{DB_i} \cup \ldots$, where $\boldsymbol{DB_i}$ is the set of time-stamped facts in the extensional part of the deductive database at time *i*.

The situation calculus successor state and frame axioms are emergent properties that are *true* in the perfect model, but are not needed in the operational semantics.

The LPS operational semantics is an observe-think-decide-act cycle in which:

> Any observed events and actions executed successfully in the last cycle are added temporarily to the database, and an event theory ***Ev*** specifies how events and actions update the extensional predicates of the database. ***Ev*** is equivalent, in effect, to the event calculus without frame axioms.
>
> For every instance of an integrity constraint whose conditions are *true* in the current state of the database, the (existentially quantified) conclusion of the constraint is added as a new goal to be made *true*.
>
> Goals are reduced to subgoals for some maximum number of steps.
>
> If there is an atomic action subgoal that can be executed in this cycle, then one such atomic action is chosen. If it is executed successfully, then it is recognised in the next cycle.

This operational semantics is sound with respect to the perfect model semantics sketched above. It is incomplete in the general case because it can make integrity constraints of the form *conditions* $\rightarrow$ *conclusion true* only by making the *conclusion true* when the *conditions* are *true*. It does not make them *true* by making the

conclusion true when the *conditions* are *false*, and it does not make them *true* by making their *conditions false*.

In LPS, observations are restricted to events that are added temporarily to the database, to aid in determining whether the conditions of an integrity constraint are true. We have excluded the use of abduction to explain observations in LPS, because it is much more complex than simply observing external events, and because abduction is not a feature of practical production systems and BDI agents.

3 An ALPA Representation of TR Programs

The translation of TR programs into ALPA is similar to our translation of LPS into ALPA [10]. We call this logic-based representation of TR programs LTR.

3.1 TR and LTR programs without an internal database.

In the simplest case, a TR program need not contain any representation of its environment. In such cases, as Rodney Brooks [2] advocates, the program uses "the world as its own model". Arguably, such TR programs without an internal database implement the "short and fast path from sensory signals to effectors" that Nilsson associates with lower-level programs.

The following example, from [6], illustrates such a TR program. It also illustrates the ability of a TR program to terminate a durative action when a higher-level condition holds and takes precedence. Here is the TR version using the Prolog convention that variables are written as a string of characters starting with an uppercase letter.

mine-pump $\{critical \leq Methane \rightarrow alarm$
$true \rightarrow operate\}$

operate $\{high < Water \vee (low < Water \wedge pump\text{-}active) \rightarrow pump$
$true \rightarrow nil\}$

Here *mine-pump* and *operate* name sub-programs, the first of which calls the second. The values of *Methane* and *Water* and the truth value of *pump-active* are observations. *alarm* and *pump* are primitive, durative actions. Note that the program is not regressive.

Contrary to Brooks [2], who argues against symbolic representations, our semantics for such low-level TR programs is given by an LTR program, which consists of an integrity constraint and a locally stratified logic program. The integrity constraint is:

$observed(T) \rightarrow mine\text{-}pump(T).$

The logic program is:

$mine\text{-}pump(T) \leftarrow methane\text{-}level(M, T) \wedge critical \leq M \wedge alarm(T)$

mine-pump(T) $\leftarrow$ *methane-level(M, T)* $\wedge$ *critical* $>$ *M* $\wedge$ *operate(T)*

operate(T) $\leftarrow$ *water-level(W, T)* $\wedge$ *high* $<$ *W* $\wedge$ *pump(T)*
operate(T) $\leftarrow$ *water-level(W, T)* $\wedge$ *low* $<$ *W* $\wedge$ *pump-active(T)* $\wedge$ *pump(T)*
operate(T) $\leftarrow$ *water-level(W, T)* $\wedge$ *high* $\geq$ *W* $\wedge$ *low* $\geq$ *Water(T)*
operate(T) $\leftarrow$ *water-level(W, T)* $\wedge$ *high* $\geq$ *W* $\wedge$ $\neg$ *pump-active(T)*

Notice that the condition *high* $\geq W$ of the third rule for *operate(T)* is redundant and can be deleted. Moreover, the defined durative action *operate(T)* can be compiled away, replacing it by its definition, giving the program:

mine-pump(T) $\leftarrow$ *methane-level(M, T)* $\wedge$ *critical* $\leq$ *M* $\wedge$ *alarm(T)*
mine-pump(T) $\leftarrow$ *methane-level (M, T)* $\wedge$ *critical* $>$ *M* $\wedge$
water-level (W, T) $\wedge$ *high* $<$ *W* $\wedge$ *pump(T)*
mine-pump(T) $\leftarrow$ *methane-level (M, T)* $\wedge$ *critical* $>$ *M* $\wedge$
water-level (W, T) $\wedge$ *low* $<$ *W* $\wedge$ *pump-active T* $\wedge$ *pump(T)*
mine-pump(T) $\leftarrow$ *methane-level (M, T)* $\wedge$ *critical* $>$ *M* $\wedge$
water-level (W, T) $\wedge$ *high* $\geq$ *W* $\wedge$ *low* $\geq$ *Water*
mine-pump(T) $\leftarrow$ *methane-level (M, T)* $\wedge$ *critical* $>$ *M* $\wedge$
water-level (W, T) $\wedge$ *high* $\geq$ *W* $\wedge$ $\neg$ *pump-active(T)*

Notice that in the translation of a rule $K_{i+1} \rightarrow a_{i+1}$, the corresponding condition of the LTR clause contains the explicit negations of all the conditions of the higher-level rules $K_1 \rightarrow a_1$ ... $K_i \rightarrow a_i$. This syntactic redundancy can be avoided in the LTR syntax by employing a similar convention to that employed in the TR syntax or by using a special operator like the "cut" in Prolog. The inefficiency of re-evaluating these conditions in the operational semantics can be avoided, as it is in the TR operational semantics, in the Prolog implementation of cut, or by means of tabling [21].

The operational semantics of LTR is based on the operational semantics of ALPA, similarly to the way in which the operational semantics of LPS is based on that of ALPA. It also mimics the operational semantics of TR programs. It is illustrated by the following example, in which *critical*, *high* and *low* are 100, 20 and 10, respectively.

	methane-level	*water-level*	*pump-active*	*alarm*	*pump*
$time_1$	66	18	no	no	no
$time_2$	77	20	no	no	no
$time_3$	88	20.0001	no	no	yes
$time_4$	99	20.00001	yes	no	yes
$time_5$	99	15	yes	no	yes
$time_6$	100	12	yes	yes	no
$time_7$	110	18	no	yes	no
$time_8$	104	19	no	yes	no
$time_9$	98	19	no	no	no
$time_{10}$	98	15	no	no	no

Here the primitive actions *alarm* and *pump* are durative actions, which are turned on automatically when their conditions hold, and turned off when their conditions do not hold. For example, at $time_6$, the conditions for *pump* fail to hold, and the pump is turned off, when the conditions of the higher priority rule for sounding the alarm hold.

The declarative semantics of such a LTR program, consisting of an integrity constraint and a locally stratified program ***P*** can be understood naturally in ALPA [8] terms:

> The task is to generate a set Δ of ground actions such that the integrity constraint, in this case *observed(T) → mine-pump(T),* is true in the perfect model of ***P*** ∪ ***Obs*** ∪ Δ, where ***Obs*** is the set of all ground facts representing the input observations, including ground facts of the form *observed(t)* for every time *t* that an observation is made.

Notice that this semantics does not depend upon the nature or even the sequencing of time points. In theory, the set of time points could be uncountably large, represented for example by the set of positive real numbers. Such a semantics would be appropriate for hardware implementations using analogue electronic circuits, as suggested by Nilsson [15].

Notice also that primitive actions are assumed to take place at the same time *T* as the conditions that trigger the actions. This is an idealization, which simplifies the theory, like the assumption of friction-less motion in the laws of physics.

3.2 TR and LTR programs with an internal database.

The simple semantics above is inadequate for TR-programs that employ a destructively updated database as a representation of the current state of the environment. For such programs it is necessary to define the semantics relative to a sequence of discrete database states. Consider, for example, Nilsson's tower-building example with towers represented as lists *S,* using LISP notation, where *car(S)* is the top block on the sub-tower *cdr(S)* [16]:

make-tower(S); S is a list of blocks,
{*tower(S) → nil*
ordered(S) → unpile(car(S))
null(cdr(S)) → move-to-table(car(S))
tower(cdr(S))→move(car(S), cadr(S))
true → make-tower(cdr(S)) }

move-to-table(X); X is a block,
{on(X,table) → nil
holding(Y) → putdown(Y, table))
clear(X) → pickup(X)
true → unpile(X) }

move(X, Y); X and *Y* are blocks,
{*on(X,Y) → nil*
holding(X)∧clear(Y) → putdown(X,Y)
holding(X) → putdown(Z, table))
clear(X) ∧ clear(Y) → pickup(X)
clear(Y) → unpile(X)
true → unpile(Y) }

unpile(X); *X* is a block,
{*clear(X) → nil*
on(Y,X) → move-to-table(Y) }

The TR-program in this example manages a small deductive database, which records the current location of blocks and whether the robot is holding a block. When a primitive action, such as *putdown* is executed, its effects are sensed and the facts affected by the action are updated in the database. Facts that are not affected are not updated, so frame axioms are unnecessary. Nilsson allows for the possibility that primitive actions might affect the database directly, but he does not explain how the effects of such internal actions are specified. In LPS we use an explicit event theory ***Ev*** without frame axioms for this purpose. We also use an event theory in LTR.

Like the event theory in LPS, the event theory ***Ev*** in LTR specifies how externally observed events and internally generated actions update the extensional predicates of the database. ***Ev*** also specifies how the database is updated when facts involving the extensional predicates are observed.

In addition to extensional predicates, defined by ground atomic facts, the deductive database can contain intensional predicates, defined by "perceptual rules", which "create increasingly abstract predicates from simpler ones". In Nilsson's tower-building example, these rules have logic programming form. Here is the translation of these rules into a locally stratified logic program, with an explicit time parameter, using the Prolog convention that *[H|S]* represents a list with first element *H,* followed by list *S*:

tower([Block|S], T) ← ordered([Block|S], T) ∧ ¬ ∃X on(X, Block, T)
ordered([Block1, Block2|S], T) ← ordered([Block2|S],T) ∧
on(Block1, Block2, T)
ordered([Block], T) ← on(Block, table, T)
clear(Block, T) ← ¬ ∃X on(X, Block, T) ∧ ¬ holding(Block, T)

Nilsson describes a computational architecture in which such rules are executed forwards, to add new facts to the database, using a truth maintenance system to delete derived facts when the extensional facts that support them are deleted. Our declarative and operational semantics are neutral with respect to whether the definitions of the intensional predicates are executed forwards or backward. If they are executed backwards, Prolog-style, then truth maintenance is unnecessary.

Here is the translation of *make-tower*[1]. The translation of the other programs is similar:

make-tower(S, T) ← tower(S, T)
make-tower([Block|S], T) ← ¬ tower([Block|S], T) ∧ ordered([Block|S], T) ∧
unpile(Block, T)
make-tower([Block], T) ← ¬ tower([Block], T) ∧ ¬ ordered([Block], T) ∧
move-to-table(Block, T)
make-tower([Block1, Block2|S], T) ← ¬ tower([Block1, Block2|S], T) ∧
¬ ordered([Block1, Block2|S], T) ∧ tower([Block2|S], T)
∧ move(Block1, Block2, T)

[1] The conditions *null(S)* and *¬null([Block2|S])*, respectively, in the 3rd and 4th clauses are unnecessary because they are implied by the list data structure.

$$make\text{-}tower([Block|S], T) \leftarrow \neg\, tower([Block|S]), T) \wedge \neg ordered([Block|S], T) \wedge \neg\, null(S, T) \wedge \neg\, tower(S, T) \wedge make\text{-}tower(S, T)$$

Arguably, the semantics of the top-level goal of the TR program is ambiguous. Given the problem of building a specific tower, say *[a, b, c],* is the goal a *(one-off) achievement* goal, to make the tower and then terminate? Or is it a (*perpetual) maintenance* goal, to make the tower and rebuild it if and when the environment interferes with it? In LTR, the different kinds of goals can be accommodated by means of a single integrity constraint:

$$required\text{-}tower(Tower, T) \rightarrow make\text{-}tower(Tower, T)$$

where *required-tower* is an extensional predicate, instances of which are updated by using an event theory ***Ev***, which contains such clauses as:

initiates(request-tower(Tower, T), required-tower(Tower))
terminates(cancel-tower(Tower, T), required-tower(Tower))

Using the event theory, an observation of an external event *request-tower(tower, t)* adds the fact *required-tower(tower)* to the database, and an observation of an external event *cancel-tower(tower, t)* deletes the fact *required-tower(tower)* from the database.

In general, the declarative semantics of TR programs with a destructively updated database is an extension of the semantics of TR programs without a database. Informally speaking, given an initial database and sequence of sensed observations, the task is to generate a set of ground actions such that all the integrity constraints are true in the perfect model determined by the program, observations, actions and associated sequence of database states. More precisely and more formally:

> Given the representation of a TR program as a locally stratified logic program ***P*** and a set ***IC*** of one or more integrity constraints with explicit time, a time-stamped representation $\boldsymbol{DB_0}$ of the initial state of the database, a set ***I*** of definitions of intensional predicates with explicit representation of time, a sequence $\boldsymbol{Obs_0, \ldots, Obs_i, \ldots}$ of sets of time-stamped input observations, and an event theory ***Ev***,

the task is to generate a sequence of time-stamped ground actions $a_0, \ldots, a_i, \ldots$ with an associated sequence $\boldsymbol{DB_1, \ldots, DB_i, \ldots}$ of time-stamped extensional databases, such that all the integrity constraints ***IC*** are true in the perfect model of $\boldsymbol{P} \cup \boldsymbol{I} \cup \boldsymbol{Obs} \cup \{a_0, \ldots, a_i, \ldots\} \cup \boldsymbol{DB}$, where
$\boldsymbol{Obs = Obs_0 \cup \ldots \cup Obs_i \cup \ldots}$ and
$\boldsymbol{DB = DB_0 \cup \ldots \cup DB_i \cup \ldots}$.

It is possible to show that a TR style of operational semantics, using destructive database updates, is sound with respect to this semantics, using an argument similar to that for showing the soundness of the operational semantics of LPS [10].

4 Alternative ALP Representations of TR programs

We argued above that the semantics of the top-level goal of a TR program is ambiguous. Here we will see that if the top-level goal is intended as a maintenance goal, then it is often natural to represent the program in ALP form as a set of conditional integrity constraints. For example, the compiled version of the *mine-pump* example can be represented simply by three integrity constraints:

$methane\text{-}level(M, T) \wedge critical \le M \rightarrow alarm(T)$
$methane\text{-}level\ (M, T) \wedge critical > M \wedge water\text{-}level\ (W, T) \wedge high < W \rightarrow pump(T)$
$methane\text{-}level\ (M, T) \wedge critical > M \wedge water\text{-}level\ (W, T) \wedge low < W \wedge$
$pump\text{-}active\ T \rightarrow pump(T)$

In this case the logic program ***P*** defines only the auxiliary inequality predicate The semantics of subsection 3.1 for TR programs with an internal database still applies, but without the need for additional observations of the form *observed(t)*:

> The task is to generate a set Δ of ground actions such that the integrity constraints ***IC*** are all true in the perfect model of $\boldsymbol{P} \cup \boldsymbol{Obs} \cup \boldsymbol{\Delta}$, where ***Obs*** is the set of all ground facts representing the input observations.

In some versions of ALP [8] conditionals can occur in the conditions of logic programs. Thus we can write, for example:

$mine\text{-}pump(T) \leftarrow [methane\text{-}level(M, T) \wedge critical \le M \rightarrow alarm(T)] \wedge$
$\quad [methane\text{-}level\ (M, T) \wedge critical > M \rightarrow operate(T)]$

$operate(T) \leftarrow [water\text{-}level\ (W, T) \wedge high < W \rightarrow pump(T)] \wedge$
$\quad [water\text{-}level\ (W, T) \wedge low < W \wedge pump\text{-}active\ (T) \rightarrow pump(T)]$

This (conditional) version is equivalent to the first version in subsection 3.1, in which *mine-pump* and *operate* are defined by normal logic programs. More generally:

Theorem: $(A \rightarrow B) \wedge (C \rightarrow D) \leftrightarrow$
$(A \wedge B) \vee (C \wedge D) \vee (\neg A \wedge \neg C)$
given that $C \rightarrow \neg A$ (equivalently $A \rightarrow \neg C$).

Note that the assumption $C \rightarrow \neg A$ holds for all LTR programs.

Proof: We use the fact that $(P \rightarrow Q) \leftrightarrow (\neg P \vee Q) \leftrightarrow (\neg P \vee (P \wedge Q))$.

$(A \rightarrow B) \wedge (C \rightarrow D) \leftrightarrow$
$(\neg A \vee (A \wedge B)) \wedge (\neg C \vee (C \wedge D)) \leftrightarrow$
$(\neg A \wedge \neg C) \vee (\neg A \wedge C \wedge D) \vee (A \wedge B \wedge \neg C) \vee (A \wedge B \wedge C \wedge D)$

Note that $(\neg A \wedge C \wedge D) \leftrightarrow (C \wedge D)$ because $C \rightarrow \neg A$.
We need to show $(A \wedge B \wedge \neg C) \vee (A \wedge B \wedge C \wedge D) \leftrightarrow A \wedge B$.

But $(A \wedge B \wedge \neg C) \leftrightarrow A \wedge B$ because $A \rightarrow \neg C$, and
$(A \wedge B \wedge C \wedge D) \leftrightarrow false$ because $C \rightarrow \neg A$.

Both ways of representing TR programs as logic programs can also be written as equivalences. For example:

$$mine\text{-}pump(T) \leftrightarrow [methane\text{-}level(M, T) \wedge critical \leq M \rightarrow alarm(T)] \wedge [methane\text{-}level (M, T) \wedge critical > M \rightarrow operate(T)]$$

$$operate(T) \leftrightarrow [water\text{-}level (W, T) \wedge high < W \rightarrow pump(T)] \wedge [water\text{-}level (W, T) \wedge low < W \wedge pump\text{-}active\ T \rightarrow pump(T)]$$

This is because perfect models[2] of logic programs are minimal models, and because the if-and-only-if form of a definition is true in a minimal model if and only if the if-half of the definition is true in the model.

The equivalence of different representations makes it easier to reason about TR programs in ALP form.

5 Reasoning about LTR Programs

The following example is based on the example in [6].

Theorem: Let *max-in* and *min-out* be the maximum rate that water enters the mine, and the minimum rate that the pump removes water from the mine, respectively. Let *water-in(In, T)* and *water-out(Out, T)* express that the water enters the mine at rate *In* at time *T* and leaves the mine at rate *Out* at time *T*, respectively. Assume that:

1) $\forall In, T\ [water\text{-}in(In, T) \rightarrow In \leq max\text{-}in$ and $max\text{-}in \leq min\text{-}out]$

Assume also that the primitive action *pump* satisfies the property:

2) $\forall M, T, W, Out\ [methane\text{-}level(M, T) \wedge critical > M \wedge water\text{-}level (W, T) \wedge low < W \wedge water\text{-}out(Out, T) \wedge pump(T) \rightarrow min\text{-}out \leq Out]$

Then the defined action *mine-pump* satisfies the property:

$$\forall M, W, In, Out, T\ [methane\text{-}level(M, T) \wedge critical > M \wedge water\text{-}level (W, T) \wedge high < W \wedge water\text{-}in(In, T) \wedge water\text{-}out(Out, T) \wedge mine\text{-}pump(T) \rightarrow In \leq Out]$$

---

[2] Note that the perfect model semantics for programs of the form $G \leftarrow (A \rightarrow B)$ can be defined in terms of normal programs of the form $G \leftarrow \neg A$
$G \leftarrow A \wedge B$.

Proof: Assume that at some given time *t*,

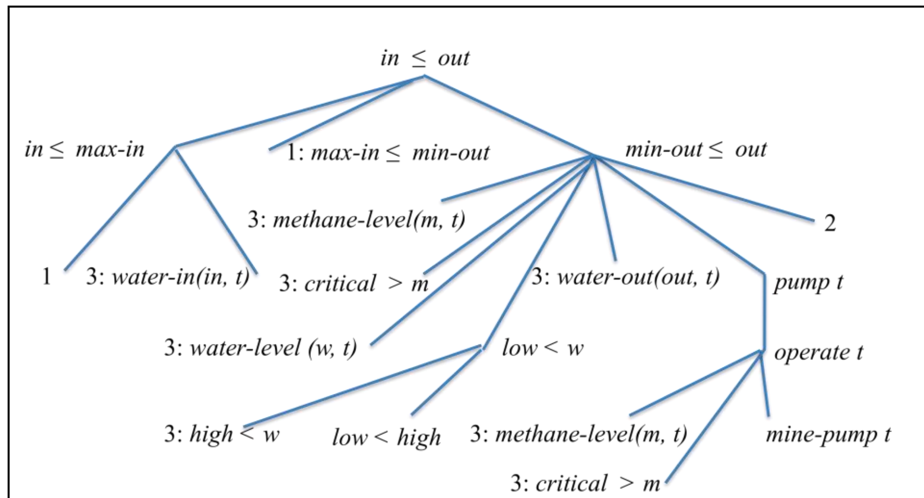
3) *methane-level(m, t)* $\wedge$ *critical* $> m$ $\wedge$ *water-level (w, t)* $\wedge$ *high* $< w$ $\wedge$
water-in(in, t) $\wedge$ *water-out(out, t)* $\wedge$ *mine-pump(t).*

It is necessary to show that *in* $\leq$ *out.*

But assumptions (1) and (3) imply *in* $\leq$ *max-in* $\leq$ *min-out.* Therefore it suffices to show *min-out* $\leq$ *out.* This follows from assumptions (2) and (3) provided we can show *pump(t)* and *low* $<$ *w*.

pump(t) follows from assumption (1), the second condition
methane-level (M, T) $\wedge$ *critical* $> M \rightarrow$ *operate(T)*
of *mine-pump T*, and the definition of *operate(T).*
low $<$ *w* follows from *low* $<$ *high* and *high* $<$ *w*.

The proof can be pictured as a tree, with the conclusion at the top and assumptions at the bottom:



6 Knowledge Representation with LTR Programs

We have shown above that LTR gives a model-theoretic semantics to TR programs, which facilitates proving TR program properties. In this section, we show that TR programs suggest a non-recursive programming style, which can also be used in LTR and ALPA more generally. We illustrate this with a path-finding example. However, the example also shows that TR programs are restricted to deterministic programs, whereas the corresponding path-finding program written in ALPA is naturally non-

deterministic.

Consider the following recursive logic program for path-finding with time represented explicitly:

$$go\text{-}to(X, T, T) \leftarrow at(X, T)$$
$$go\text{-}to(X, T_1, T_2) \leftarrow \neg at(X, T_1) \wedge at(Y, T_1) \wedge towards(Y, Z, X) \wedge move(Y, Z, T_1, T) \wedge go\text{-}to(X, T, T_2)$$

Here $go\text{-}to(X, T_1, T_2)$ represents the "macro-action" of going to X from time T_1 to time T_2. The predicate $towards(Y, Z, X)$ non-deterministically identifies a place Z that is next to Y and in the direction from Y towards X. There could be several alternative such places Z. So an agent would need to choose between them, perhaps by planning ahead to find a path that ends in X, and then moving along that path to get to X.

TR programs suggest a non-recursive way of writing a similar logic program:

$$go\text{-}to(X, T) \leftarrow at(X, T)$$
$$go\text{-}to(X, T) \leftarrow \neg at(X, T) \wedge at(Y, T) \wedge towards(Y, Z, X) \wedge move(Y, Z, T)$$

or in the *conditional* representation

$$go\text{-}to(X, T) \leftarrow [\neg at(X, T) \wedge at(Y, T) \rightarrow towards(Y, Z, X) \wedge move(Y, Z, T)]$$

The variable Z, which does not occur in the conclusion $go\text{-}to(X, T)$ of the clause, is existentially quantified in the conditions of the clause.

Both of these non-recursive formalisations need to be augmented with an integrity constraint such as $required\text{-}destination(Place, T) \rightarrow go\text{-}to(Place, T)$, where *required-destination* is an extensional predicated updated by observations of events, say, *request-destination* and *cancel-destination*. This is similar to the predicate *required-tower* and the events *request-tower and cancel-tower*, in the case of the tower building example, earlier.

It is not possible to represent the non-recursive logic program directly as a TR program. Suppose we try to represent it as:

$$go\text{-}to(X)\ \{at(X) \rightarrow nil,\quad at(Y) \wedge towards(Y, Z, X) \rightarrow move(Y, Z)\}$$

Then what is the implicit quantification of the variable Z? Is it that the agent should move to every place Z that is towards X from Y? Or does it mean that the agent should non-deterministically find one such place Z and move towards it? The alternative representation:

$$go\text{-}to(X)\ \{at(X) \rightarrow nil,\quad at(Y) \rightarrow towards(Y, Z, X) \wedge move(Y, Z)\}$$

is not allowed in the syntax of TR programs, and is not catered for in the operational semantics of TR programs.

Notice that the non-recursive style can also be used for the tower-building program. Here is the top-level of a non-recursive LTR program:

make-tower(S, T) ← tower(S, T)

make-tower([Block|S], T) ← ¬ tower([Block|S], T) ∧ ordered([Block|S], T) ∧
unpile(Block, T)

make-tower([Block], T) ← ¬ tower([Block], T) ∧ ¬ ordered([Block], T) ∧
move-to-table(Block, T)

make-tower(S, T) ← ¬ tower(S, T) ∧ ¬ ordered(S, T) ∧
append(S1, [B1, B2|S2], S) ∧ ordered([B2|S2], T) ∧
¬ on(B1, B2, T) ∧ move(B1, B2, T)

make-tower(S, T) ← ¬ tower(S, T) ∧ ¬ ordered(S, T) ∧
append(S1, [B], S) ∧ ¬ ordered([B], T) ∧
move-to-table(B, T)

The condition *append(S1, [B1, B2|S2], S)* splits *S* into a tower *[B2|S2]* that has already been built and the part that remains to be built.

In the next section we describe LPS, with a view towards embedding both LTR and LPS in a more expressive and more powerful ALPA framework with a destructively updated database.

7 The Logic-Based Production System and Agent Language LPS

LPS [9, 10] combines logic programs and production systems in a logical framework based on ALP. The relationship between LPS and ALP is analogous to the relationship between LTR and ALP.

Both TR programs and LPS employ a destructively updated database and a syntax without time, but can be given a model-theoretic semantics by translating them into ALPA with an explicit representation of time. In both cases, the semantics is defined relative to the time-stamped sequence of observations, actions and database states.

TR programs and LPS differ in their ontologies for actions. In TR programs, actions are executed duratively, for as long as their associated conditions continue to hold. The effects of actions, like the level of water in a mine or a robot's location, are sensed as observations, rather than derived by means of an event theory. These effects can vary continuously as a function of the durations of actions, as in the case of pumping, which affects the level of water, and moving forwards, which affects the robot's location.

In LPS, actions are discrete events, which transform one state of the world (or the database) into another. The effects of actions and other events are defined, as in the situation calculus [20] and event calculus [11], by an event theory, which specifies the extensional predicates that are initiated or terminated by events, and the preconditions

that must hold for actions to be possible. LPS uses the event theory (but without frame axioms) to update the database.

Perhaps the biggest difference between TR programs and LPS is that LPS allows definitions of sequences of database state transitions. For example, the recursive definition of the macro-action *go-to* in the previous section could be written in LPS in the form:

$$go\text{-}to(X) \leftarrow at(X)$$
$$go\text{-}to(X) \leftarrow \neg\, at(X) : at(Y) : towards(Y, Z, X) : move(Y, Z) : go\text{-}to(X)$$

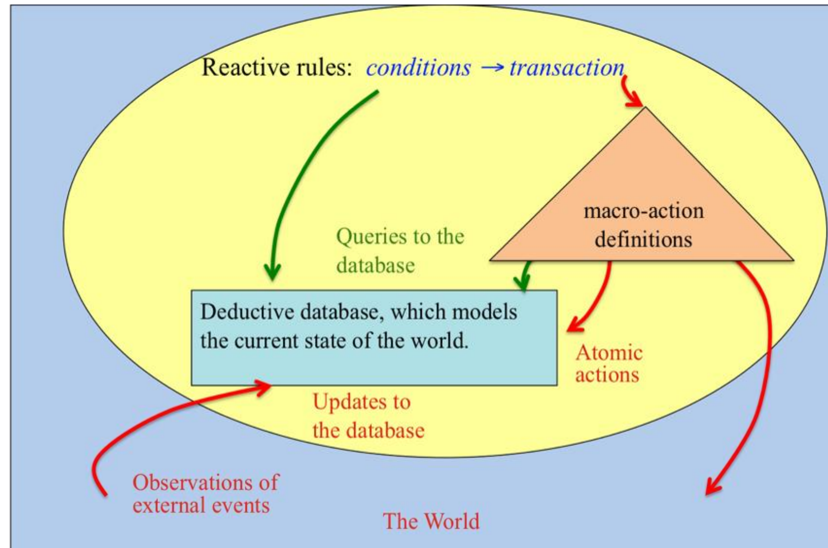
The meaning of the sequential connective **:** is given by the translation into ALP, which we have already seen in the last section:

$$go\text{-}to(X, T, T) \leftarrow at(X, T)$$
$$go\text{-}to(X, T_1, T_2) \leftarrow \neg at(X, T_1) \wedge at(Y, T_1) \wedge towards(Y, Z, X) \wedge move(Y, Z, T_1, T) \wedge go\text{-}to(X, T, T_2).$$

The action $move(Y, Z, T_1, T)$ transforms the database state at time T_1 to the next state at time T. The event theory updates the database by deleting the old location *at(Y)* and adding the new location *at(Z).*

Database transitions in LPS are similar to transactions in Transaction Logic [1]. In both cases, transactions are alternating sequences of queries and actions. An action can be an atomic action, which directly updates the external environment and/or the internal database, or a macro-action, which is the name of a transaction. In this respect, LPS is also similar to Golog [20]. However, like Golog and unlike Transaction Logic, transactions in LPS are not atomic and cannot be rolled back, although this is a feature that could be added to LPS.

The top-level of an LPS program consists of reactive rules, which are like condition-action rules in production systems and like event-condition-action rules in active database systems. However in LPS, reactive rules have the semantics of integrity constraints in ALP, and their conclusions can be transactions (or equivalently, macro-actions). The structure of an LPS program can be pictured roughly like this:



Given their compatible syntax and operational and declarative semantics, LTR and LPS can be combined in a unified language inheriting the benefits of the separate languages. One way of combining them is to embed LTR programs in LPS programs. For example, here is the definition, translated into ALP, of a macro-action that calls the teleo-reactive program *make-tower* to construct a tower and terminate when the tower is completed:

$$achieve\text{-}tower\ (S, T, T) \leftarrow tower(S, T)$$
$$achieve\text{-}tower\ (S, T_1, T_2) \leftarrow \neg\ tower(S, T_1) \wedge make\text{-}tower(S, T_1) \wedge achieve\text{-}tower(S, T_1{+}1, T_2)$$

Similarly the teleo-reactive LTR program *go-to* of the last section can be embedded in an LPS macro-action definition:

$$get\text{-}to(X, T, T) \leftarrow at(X, T)$$
$$get\text{-}to(X, T_1, T_2) \leftarrow \neg at(X, T_1) \wedge go\text{-}to(X, T_1) \wedge get\text{-}to(X, T_1{+}1, T_2)$$

to move towards a place, and terminate when reaching it. In both the case of *achieve-tower* and the case of *get-to,* the goal is achieved “teleo-reactively”, taking advantage of any favourable changes in the environment and recovering gracefully from any unfavourable changes.

8 Conclusions: ALPA as a Unifying Framework

The translations of LPS and TR programs into ALPA illustrate the broader potential of ALPA to unify different knowledge representation formalisms. The contribution of LPS is that it shows how to give a model-theoretic semantics to programs that maintain a destructively updated database. But this contribution can be generalised to any representation in ALPA that similarly maintains such a database.

Complex event recognition and processing (CEP) [12] is another programming paradigm that could benefit from such a treatment. In the same way that macro-actions can be defined in terms of queries and actions, complex events can be defined in terms of conditions and simpler events. Conditions can be evaluated either by querying sensory inputs or by querying the database. Atomic events can be recognised by input observations. Here are two examples from [22], written in ALP form:

shoplift (Item, T1, T2) $\leftarrow$ *shelf-reading(Item, T1)* $\wedge$ *exit-reading(Item, T2)* $\wedge$
$\neg$ *(check-out-reading(Item, T)* $\wedge$ *T1* $\leq$ *T* $\wedge$ *T* $\leq$ *T2)* $\wedge$ *T2 – T1 < 12 hours*

overdose(Person, antibiotics, T1, T2) $\leftarrow$ *ingest(Person, Medicine1, Dosage1, T1)* $\wedge$
ingest(Person, Medicine2, Dosage2, T2) $\wedge$ *antibiotics(Medicine1)* $\wedge$
antibiotics(Medicine2) $\wedge$ *Dosage1+Dosage2 >1000* $\wedge$ *T2 – T1 < 4 hours*

The first identifies the occurrence of a complex event of shoplifting when an item that was on shelf is removed from the store without being checked out. The second identifies a complex event of overdosing when a person has taken more than 1000 units of antibiotics in less than 4 hours in two doses.

Notice that, to specify such complex events, we need to be able to represent temporal constraints on the times that conditions hold and events are observed. However, with the ability to represent such constraints, it is easy to specify a partial ordering among the conditions and events that make up a complex event.

Notice also that the ALPA framework allows not only defining (and thus identifying) complex events, but also reacting to them (via ALPA integrity constraints) with transactions.

Investigating complex event processing further is part of our future work. Other topics we plan to consider in the future include formalisation and proof of other properties of LTR programs, such as “progress” towards achieving goals, and formal characterisations of how an LTR agent “recovers” and “re-plans” after environmental interference, without the need for explicit inclusion of such features in the representation.

We have a prototype implementation of LPS, which is sufficiently general that it can run programs written in LTR. Exploring the scalability of the implementation for more substantial examples is also part of future work.

Afterword. It is a pleasure to dedicate this paper to Marek, whose friendship and intellectual inspiration have been a great support over many years.

Acknowledgments. Many thanks to Keith Clark for awakening our interest in TR programs, and to Clive Spencer and Alan Westwood from Logic Programming Associates for many useful discussions and for their work on the prototype implementation of LPS and its application to LTR. Thanks also to the anonymous referees.

References

1. Bonner and M. Kifer.: Transaction logic programming. In Warren D. S., (ed.), Logic Programming: Proc. of the 10th International Conf., 257-279 (1993)
2. Brooks, R.A., Intelligence Without Representation, Artificial Intelligence 47, 139-159 (1991)
3. Coffey, S., and Clark, K. L., A Hybrid, Teleo-Reactive Architecture for Robot Control, In Proceedings of the Second International Workshop on Multi-Agent Robotic Systems (MARS-06) (2006)
4. Gordon, E. and Logan, B., Game Over: You have been beaten by a GRUE, In Challenges in Game Artificial Intelligence: AAAI Workshop (2004)
5. Gerhard Gubisch, Gerald Steinbauer, Martin Weiglhofer and Franz Wotawa A., Teleo-Reactive Architecture for Fast, Reactive and Robust Control of Mobile Robots, In IEA/AIE 2008 Wroclaw, Poland, Lecture Notes in Computer Science, Volume 5027, 541-550 (2008)
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. Kowalski, R.: Computational Logic and Human Thinking: How to be Artificially Intelligent, Cambridge University Press (2011)
8. Kowalski, R. and Sadri, F.: From Logic Programming Towards Multi-agent Systems, *Annals of Mathematics and Artificial Intelligence*, Volume 25, 391-419 (1999)
9. 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)
10. 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)
11. Kowalski, R. and Sergot, M.: A Logic-based Calculus of Events. In New Generation Computing, Vol. 4, No.1, 67-95 (1986). Also in The Language of Time: A Reader (eds. Inderjeet Mani, J. Pustejovsky, and R. Gaizauskas) Oxford University Press (2005)
12. Luckham, D.: The Power of Events - An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison-Wesley (2002)
13. Marinovic, S., Twidle, K., Dulay, N., Teleo-reactive Workflows for Pervasive Healthcare, First IEEE PerCom Workshop on Pervasive Healthcare (2010)
14. Nilsson, N., *Toward Agent Programs with Circuit Semantics*, Technical Report STAN-CS-92-1412, Stanford University Computer Science Department (1992)

15. Nilsson, N.J., Teleo-reactive Programs for Agent Control, Journal of Artificial Intelligence Research cs.AI/9401, 139-158, January (1994)
16. Nilsson, N.J., Teleo-reactive Programs and the Triple-tower Architecture, Electronic Transactions on Artificial Intelligence, Vol. 5, Section B, 99-110 (2001)
17. Przynusinski, T., On the Declarative and Procedural Semantics, Journal of Automated Reasoning 5: 167-295 (1989)
18. Przymusinski, T.: On the Declarative and Procedural Semantics of Stratified Deductive Databases. In J. Minker, editor, Foundations of Deductive Databases and Logic Programming, Morgan-Kaufman, Los Altos, CA, 193–216 (1988)
19. Rao, A. S., Georgeff, M. P.: BDI Agents: From Theory to Practice, International Conference on Multiagent Systems - ICMAS , 312-319 (1995)
20. Reiter, R.: Knowledge in Action. MIT Press (2001)
21. Sagonas, K. F., Swift,T., Warren, D. S.: XSB as an Efficient Deductive Database Engine, Sigmod Record , vol. 23, no. 2, 442-453 (1994)
22. Wu, E., Diao, Y., Rizvi, S.: High-Performance Complex Event Processing Over Streams, SIGMOD, Chicago, Illinois, USA, 407-418 (2006)
23. Broda, K., Hogger, C.J.: Designing and Simulating Individual Teleo-Reactive Agents, Poster Proceedings, 27th German Conference on Artificial Intelligence (2004)
24. Kowalski, R., and Sadri, F.: Integrating Logic Programming and Production Systems in Abductive Logic Programming Agents, In Web Reasoning and Rule Systems (eds. A. Polleres and T. Swift) Springer, LNCS 5837 (2009)
25. van Emden, M. and Kowalski, R.: The Semantics of Predicate Logic as a Programming Language, in *JACM*, Vol. 23, No. 4, 733-742 (1976)

Appendix

Brief Introduction to Logic Programming

Logic programs are collections of sentences in the logical form of conditionals:

> ***if*** *conditions* ***then*** *conclusion*, also written
> *conclusion* ← *conditions*.

Such conditionals (also called *clauses*) combine a *conclusion*, which is an atomic formula, with *conditions*, which are a conjunction of atomic formulas or negations of atomic formulas. A clause without negative conditions is called a *definite clause*.

The number of *conditions* in a clause can be zero, in which case the clause is written without the implication sign ←, simply as *conclusion.* If the number of *conditions* is not zero, then the clause is also sometimes called a *rule*.

All variables in a clause are implicitly universally quantified with scope the clause. Clauses with no variables are called *ground clauses*. Ground clauses with zero conditions are also called *facts*.

Logic programs can be viewed as *definitions* of the predicates occurring in the conclusions of clauses. These definitions are used to solve *goal clauses*, which are existentially quantified conjunctions of atomic formulas or negations of atomic formulas. A *definite goal clause* is a goal clause without negative conditions. Both definite clauses and definite goal clauses are also called *Horn clauses.*

The use of logic programs to solve goals can be viewed in both programming and database terms. Viewed in programming terms, logic programs compute values of the existentially quantified variables in goal clauses. Viewed in database terms, they derive answers to goal clauses, viewed as database queries.

Given a logic program L, to solve a goal clause G, it is necessary to find a variable-free (i.e. ground) instance G' of G such that G' holds with respect to L. In the case of a definite clause program L, there are two equivalent semantic notions of what it means for a sentence S to hold with respect to L:

L logically entails S, i.e. S is true in all models of L.
S is true in the unique minimal model of L.

The minimal model semantics has a number of advantages, which are detailed for example in [7].

The *minimal model* [25] of a definite clause program L is equivalent to the set of all facts that can be derived from L by repeatedly applying the two inference rules of instantiation and *modus ponens*, until no new facts can be derived. *Instantiation* replaces all occurrences of a variable in a clause by a ground term constructed from the constants and function symbols of the language. *Modus ponens* derives the *conclusion* of a clause *conclusion* $\leftarrow$ *conditions* from the *conditions* given as facts.

In the case of non-Horn clauses that are locally stratified [18], the minimal model semantics has a natural generalisation to *perfect models*. For simplicity, consider the case of a ground clause program L with two strata, determined by partitioning the set of all ground atoms A of the language into two disjoint sets (or strata) A_0 and A_1. L is *locally stratified* if L is the union $L_0 \cup L_1$ of two disjoint sets of clauses:

L_0 consists of clauses whose conclusion and positive conditions belong to A_0, and that have no negative conditions.

L_1 is the set of all the clauses in L whose conclusion belongs to A_1, whose positive conditions belong to $A_0 \cup A_1$, and whose negative conditions have atoms in A_0. Thus no clause in L contains a negative condition in A_1.

The perfect model of L is the union $M_0 \cup M_1$ of two minimal models:

M_0 is the minimal model of the definite clause program L_0.
M_1 is the minimal model of the definite clause program L_1' obtained from L_1 by evaluating in M_0 both the positive and negative conditions of clauses in L_1 whose atoms are in A_0. i.e. L_1' contains a clause of the form *conclusion* $\leftarrow$ *conditions*$_1$ iff L_1 contains a clause of the form *conclusion* $\leftarrow$ *conditions*$_1$ *and conditions*$_2$, where the atoms in *conditions*$_1$ are all in A_1, the atoms in *conditions*$_2$ are all in A_2, and the *conditions*$_2$ are all true in M_0.

This definition can be generalised in three ways: 1) from ground logic programs to programs containing variables, by adding universal instantiation; 2) to an unbounded number of strata: 0, 1, …. and 3) from conditions that are negative atoms at lower strata to conditions that are arbitrary formulas in the vocabulary of lower strata.