Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Gopalan Nadathur
The University of Chicago, Department of Computer Science
1100 East 58th Street, Chicago, IL 60637, USA
E-mail: gopalan@cs.uchicago.edu

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

**Principles and practice of declarative programming** : proceedings / International Conference PPDP '99, Paris, France, September 29 - October 1, 1999. Gopalan Nadathur (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1702)
ISBN 3-540-66540-4

CR Subject Classification (1998): D.3, D.1, F.3

ISSN 0302-9743
ISBN 3-540-66540-4 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN: 10704567 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

# OPENLOG: A Logic Programming Language Based on Abduction

Jacinto A. Dávila

Centro de Simulación y Modelos (CESIMO)
Universidad de Los Andes. Mérida. Venezuela
FAX: +58 74 403873
jacinto@ing.ula.ve
http://cesimo.ing.ula.ve

**Abstract.** In this paper, we introduce a programming language for an abductive reasoner. We propose the syntax for an imperative language in the usual manner and its semantics as a mapping from the language statements to an abductive logic program. The design is such that any semantics for abductive logic programs could be taken as the basic semantics for the programming language that we propose. In this way, we build upon existing formalizations of abductive reasoning and abductive logic programming. One innovative aspect of this work is that the agent processing and executing OPENLOG programs will stay *open* to the environment and will allow for changes in its environment and assimilation of new information generated by these changes.

## 1 Introduction

Abduction is a non-valid form of reasoning in which one infers the premises of a rule given the consequent. This form of reasoning is not valid in classical first order logic since, for instance, one is not allow us to deduce the atom $b$ from the clause $h \leftarrow b$ and the atom $h$. However, in general, in the presence of $h$ and this clause, our intuition allows us to say that $b$ could well be the case. That is, we are allowed to offer $b$ as an explanation or a hypothesis for $h$ in the context of that clause when we do not have more information. This is abduction. An abductive reasoner uses abduction as one of its inference rules. Abduction enables reasoning in the absence of full information about a particular problem or domain of knowledge.

In this paper, we introduce a programming language for an abductive reasoner. We propose the syntax for an imperative language in the usual manner (summarized in table 1) and its semantics is defined as a mapping from the language statements to an abductive logic program (shown in table 2). The design is such that any semantics for abductive logic programs could be taken as the basic semantics for the programming language that we propose. In this way, we build upon existing formalizations of abductive reasoning and abductive logic programming.

A substantial effort has been made to formalize abductive reasoning. Poole's Theorist [27] was the first to incorporate the use of abduction for non-monotonic reasoning. Eshghi and Kowalski [10] have exploited the similarities between abduction and negation as failure and provided a proof procedure based on a transformation of logic programs with negation into logic programs with *abducible atoms*. de Kleer incorporates abduction into the so-called truth maintenance systems to obtain the ATMS [7]. Also, in [3], L. Console, D. Theiseider and P. Torasso analyse the relationships between abduction and deduction and define what they call an **abduction problem** as a pair $< T, \phi >$ where:

1. $T$ (the domain theory) is a hierarchical logic program[1] whose abducible atoms are the ones not occurring in the head of any clause.
2. $\phi$ (the observations to be explained) is a consistent conjunction of literals with no occurrence of abducible atoms.

A solution to the abduction problem is a set of abducible atoms that, together with $T$ , can be used to explain $\phi$.

The purpose of imposing structures such as $< T, \phi >$ upon a reasoning problem is to create **frameworks** in which the semantics of each component and its relationships with other components can be established in a declarative manner. A framework is a structure that distinguishes between types of elements in a formalization. For instance, the framework $< T, \phi, Ab >$ could be used to say that one has a theory $T$, a set of observations $\phi$ and that these observations can be explained by abducing predicates in $T$ whose names appear in $Ab$ (abducible predicates). These distinctions are then used to justify differential treatment of each type of component. In the cases considered here, for instance, abducible predicates and non-abducible predicates, so separated by the framework, are processed differently. The distinction captures the fact that the former, unlike the latter, denote uncertain or incomplete information.

The use of *frameworks* has been taken further by Kakas and Mancarella [17], Denecker and De Schreye [9], Toni [33], Fung [14] and more recently, Wetzel *et al* [36], [35] in the context of incorporating abduction into constraint logic programming. In [16] there is an overview of the first efforts to incorporate abduction into logic programs. In [13] there is a preliminary description of the abductive framework that we have used (in [6]) to formalize the reasoning mechanism of an agent. In this work, the agent is as an abductive reasoner that uses abduction to plan its actions to achieve its goals.

## 2 An Abductive Proof Procedure

In [13], Fung and Kowalski introduce an abductive proof procedure aimed at supporting abductive reasoning on predicate logic and, in particular, on *abductive logic programs.* The iff proof procedure, as they call it, (**iffPP** hereafter), is an aggregate of the following inference rules: unfolding, propagation, splitting,

[1] A hierarchical logic program is a logic program without recursive rules.

case analysis, factoring, logical simplifications and a set of rewriting rules to deal with equalities plus the *abductive rule* described above. Fung and Kowalski also produce soundness and completeness results in [13]. We describe an implementation of **iffPP** in [6] together with some examples of how it could be used.

A proof procedure can be seen as specifying an abstract machine that transforms formulae in other formulae. It could even be seen as "an implementation independent interpreter for" the language of those formulae [34]. That is, a proof procedure determines an operational semantics for logic programs (.ibid). Thus, **iffPP** specifies an operational semantics for *abductive logic programs.* By relating this operational semantics to a programming language, one can get to program the abductive reasoner for particular applications, such as hypothetical reasoning and problem solving (e.g. planning) in specific knowledge domains and for pre-determined tasks.

In this paper, we go a step further in the definition of a programming language for the abductive reasoner defined by **iffPP**. Instead of simply relying on the procedural interpretation of (abductive) logic programs, we introduce a more conventional imperative language and explain how this can be mapped onto abductive logic programs of a special sort. These abductive logic programs lend themselves to a form of default reasoning that extends the traditional use of programming languages, i.e., the new definition supposes a re-statement of what a program is.

In the context of this research, a program is seen as a scheme that an agent uses to generate plans to achieve some specified goal. These plans ought to lead that agent to display an effective, goal-oriented behaviour that, nevertheless, caters for changes in the environment due to other independent processes and agencies. This means that, although the agent would be following a well-defined program, it would stay *open* to the environment and allow for changes in its circumstances and the assimilation of new information generated by these changes.

So defined, a program is not a closed and strict set of instructions but a list of assertions that can be combined with assertions from other programs. One advantage of this definition is that the code being executed remains *open* to updates required by changes in the circumstances of execution. The other important advantage is that it allows the executor of the program to perform a form of default reasoning. By assuming certain set of circumstances, the agent will execute certain sequence of actions. If the circumstances change, perhaps another sequence will be offered for execution.

The paper is organized as follows: The next section shows an example to illustrate the principles of abductive programming. The following section introduces the syntax and semantics of a new logic programming language for abductive programming: OPENLOG. Then, the semantics of OPENLOG and its relationship with *background theories* based on the Event Calculus [20] is explained. A discussion of the characteristics and advantages of OPENLOG is also presented before concluding with some remarks about future research.

## 3 Toy Examples of Abductive Programming with OPENLOG

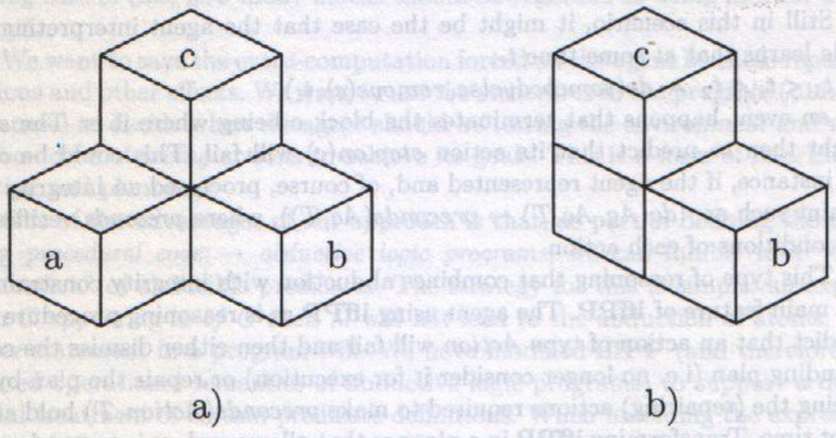

**Fig. 1.** Two Blocks-World scenarios for planning

In this section we illustrate with examples the relationship between abduction and planning based on pre-programmed routines. Consider the scenario in figure 1:

*Example 1.* An agent is presented with the challenge of climbing a mountain of blocks to get to the top. The agent can climb one block at a time provided, of course, that the block is there and at the same level (i.e. just in front). The planning problem is then to decide which blocks to climb onto and in which order. An OPENLOG procedure to guide this planning could be:

```
proc climb
  begin
    if infront( Block ) and currentlevel( Level )
       and  Block is_higher_than Level  then
      begin
        step_on( Block ) ; climb
      end
  end
```

Given the scenario in figure 1 a) and the OPENLOG code above, an abductive agent might generate the alternative plans:
$do(self, step\_on(a), t_1) \ \wedge \ t_1 < t_2 \wedge do(self, step\_on(c), t_2)$ and
$do(self, step\_on(b), t_1) \ \wedge t_1 < t_2 \wedge \ do(self, step\_on(c), t_2)$,

where $do(Agent, A, T)$ can be read as "Agent does action A at time T". This can be done by relating every OPENLOG program to an abductive logic program that refers to the predicates *do* and $<$ , and declaring those predicates as *abducibles.* This *mapping* is provided by the definition of the predicate *done* as shown in section 6 and in section 7.

Still in this scenario, it might be the case that the agent interpreting this code learns that at some time $t_i$:

$t_1 < t_i < t_2 \land do(somebodyelse, remove(c), t_i)$,

i.e. an event happens that terminates the block $c$ being where it is. The agent ought then to predict that its action $step\_on(c)$ will fail. This could be done, for instance, if the agent represented and, of course, processed an integrity constraint such as: $(do(Ag, Ac, T) \rightarrow preconds(Ac, T))$, where *preconds* verifies the preconditions of each action.

This type of reasoning that combines abduction with integrity constraints is the main feature of **iffPP**. The agent using **iffPP** as is reasoning procedure may predict that an action of type *Action* will fail and then either dismiss the corresponding plan (i.e. no longer consider it for execution) or repair the plan by abducing the (repairing) actions required to make $preconds(Action, T)$ hold at the right time. Transforming **iffPP** in a planner that allows replanning must be done with care, however, because, as we argue below, it may lead to "over-generation" of abducibles, i.e. to produce too many "repairing" alternatives (some of them with there own problems due to ramifications).

What we have done to tackle the original problem (transforming **iffPP** in the planner for an open agent) is to combine OPENLOG (with its solution for over-generation of abduction) with another programming language, this one based on integrity constraint, which we call ACTILOG [6]. We focus this paper on OPENLOG, due to space constraints and because integrity constraints equivalent to the one above (that involves the predicated *preconds*) can also be produced from OPENLOG code.

## 3.1 Over-Generation of Abducibles

As we said, one has to be careful with the generation of abducible predicates. Notice, for instance, that in figure 1 b) the only *feasible* plan is:

$do(self, step\_on(b), t_1) \land t_1 < t_2 \land do(self, step\_on(c), t_2)$,

because the block $a$ is not there. The agent may know about actions that cause $infront(a)$ to be the case (such as, say, $put\_block\_in\_front(a)$). It could therefore schedule one of these actions to *repair* the plan. In (the more usual) case where the agent cannot actually perform the action, the only way to prevent the scheduling of the repairing action is to perform some (non-trivial) extra computation to establish, for instance, that the agent will not be able to "move the block $a$" in the current circumstances.

This type of behaviour is what one would get from a general purpose, abductive reasoner like **iffPP**. It will "generate" all the possible combinations of abducibles actions to satisfy its goals. And these may be too many, irrelevant or impossible. Observe that this general reasoner will generate the same sets of *step_on* actions for both situations a) and b) in figure 1. Moreover, it will add actions to repair the plans (all of them) in all the possible ways (e.g. moving blocks so that *infront* holds for all of them). The problem becomes even more complex if one considers other physical or spatial effects the agent should be taking care of (like how many blocks should be regarded as being in front of the agent).

We want to save the extra-computation forced on the agent by these repairing actions and other effects. We want to use the structures in the program (`climb` in this case) to decide when the agent should be testing the environment and when it should be *abducing* actions to achieve its goals. This is a form of interleaving testing and planning.

One of the advantages of our approach is that, as part of defining the mapping *procedural code* → *abductive logic programs*, we can inhibit that "overgeneration" of abducible predicates. The strategy for this is simple: an expression $C$ appearing in $if\ C\ then$ ... will not lead to the abduction of atoms. Any other statement in a program will. We have modified **iffPP** (and therefore the related operational semantics of abductive logic programs) to support a differential treatment of certain predicate definitions. When unfolding the expression $C$, in $if\ C\ then$ ..., the involved predicates are not allowed to contribute with more abducibles, but simply to test those previously collected in order to satisfy the definitions. Thus, the expression $if\ C\ then$ ... in OPENLOG is more than a mere shorthand to a set of clauses in an abductive logic program. It is a way for the OPENLOG programmer to state which part of the code must carry out tests (on the agent's knowledge) and which must lead to actions by the agent. This strategy adds expressiveness to the programming language and makes of abduction a practical approach for the planning module of an agent [6].

With the inhibited platform and the code in example 1 above we state that, at that stage, the agent is just interested in testing whether $infront(A)$ actually holds for some block $A$. If the programmer decides that the agent must also build the mountain to be climbed, then she will have to write for the "climber-builder" agent a program such as this:

*Example 2.* `proc climb`
```
 begin
   if infront( Block ) and currentlevel( Level )
      and Block is_higher_than Level then
        begin
          step_on( Block ) ; climb
        end
   else
      if available( Block ) and not infront( Block ) then
         put_block_in_front( Block ) ; climb
 end
```

In this second program, when the agent has no block in front (so that the first test fails) and there is some block available in the neighbourhood, then the agent will indeed schedule (abduce) the action $put\_block\_in\_front(A)$ for execution (provided that action is a primitive action).

Thus, with inhibited abduction the agent is interleaving the "testing" of properties with the "planning" of actions. This testing is program-driven, i.e. the programs and the goals establish when the system will be testing and when it will be planning (abducing). Moreover, notice that the "testing" is not restricted to the current state of the world. Earlier actions in a plan can be used to establish that some property holds at a certain time-point. For instance, the climbing agent above may be able to deduce that after $do(step\_on(a), t_1)$, $infront(c)$ will hold.

## 4 OPENLOG: From Structured to Logic Programming

In the following, a well-known programming language (STANDARD PASCAL) is used as the basis to create a language that supports the kind of *open* problem-solving and planning behaviour mentioned above. The semantics of the resulting language (OPENLOG) is based on a logic of actions and events that caters for input assimilation and reactivity. In combination with the reactive architecture described in [6], where the interleaving of planning and execution is clearly defined, this language can provide a solution to the problem of agent specification and programming.

OPENLOG is aimed at the same applications as the language GOLOG of Levesque *et al* [21] i.e. agent programming. Our approach differs from Levesque *et al*'s in that there is no commitment to a particular logical formalism. One can employ the Situation Calculus or the Event Calculus depending on the requirements of one's architecture. However, the Event Calculus has turned out to be more expressive and useful for the reactive architecture described in [6].

Like GOLOG, our approach also regards standard programming constructs as macros. However, here they are treated as special predicates or terms[2]. There is no problem with recursive or global procedures. Procedures are like predicates that can be referred to (globally and recursively or non-recursively) from within other procedures. Interpreting these macros is, in a sense, like translating traditional structured programs into normal logic programs.

The following section 5 describes the syntax of the language which is, basically, a subset of PASCAL extended with operators for parallel execution. Section 6 explains the semantics of OPENLOG by means of a logic program (defining the predicate *done*). In section 7, we introduce the *background theories*: the temporal reasoning platform on which OPENLOG semantics in based. In [6], we illustrate the use of OPENLOG and the background theories with a more elaborated example: The Elevator Controller.

[2] See [DN01] in table 2 below: **proc** can be regarded as a two-argument predicate, the following symbol is a term, and **begin** and **end** are bracketing a more complex term.

## 5 The Syntax of OPENLOG

The syntax of OPENLOG is described in BNF form[3] in table 1.

The syntax is left "open" to accommodate, in suitable syntactic categories, those symbols designated by the programmer to represent *fluents, primitive actions* and *complex actions.* In addition to the syntactic rules, the system must also provide translations between the "surface syntax", that the programmer will use to write each *Query*, and the underlining logical notation.

In this initial formalization, PASCAL syntax is *limited* to the least number of structures required for structured programming: (*";", "if.. then.. else..", "while"*). On the other hand, the syntax supports the representation of parallel actions through the compositional operators *par* [4] and + [5].

## 6 The Semantics of OPENLOG

The semantics of the language is stated in table[6] 2 by means of the predicate *done*[7]. The definition of *done* can also function as an interpreter for the language. Declaratively, $done(A, T_o, T_f)$ reads "an action of type $A$ is started at $T_o$ and completed at $T_f$". As the definition of *done* is a logic program, any semantics of normal logic programming can be used to give meaning to OPENLOG programs.

One of the innovations in OPENLOG is that between any two actions in a sequence it is always possible to "insert" a third event without disrupting the semantics of the programming language. Axiom [DN02] formalizes this possibility. This is what we mean by plans (derived from OPENLOG programs) as *being open to updates.*

The definition of semantics in table 2 needs to be completed with a "base case" clause for the predicate *done* and the definition of *holds*. These two elements are part of the semantics, but they are also the key elements of a *background theory* $\mathcal{B}$.

[3] In the table, $S_j$ means an instance of S of sub-type j. $(A)^*$ indicates zero or more occurrences of category A within the brackets.

[4] Unlike those semantics of interleaving ([15], [24]) this is a form of real parallelism. Actions start simultaneously, although they may finish at different times. Notice that when all the actions have the same duration (or when they all are "instantaneous") this operator is equivalent to +. Also, observe that the agent architecture described in [18] only handle actions which last for one unit of time. We relax this limitation in [6].

[5] used as well to express real parallelism. Actions start and finish at the same time. This allows the programmer to represent actions that interact with each other so that the finishing time of one constraints the finishing time of the other. For instance, taking a bowl full of soup with both hands and avoiding spilling [32].

[6] PROLOG-like syntax is being used.

[7] The definitions of other predicates are also required but are not problematic.

| ***Table 1 OPENLOG: Syntax*** | | |
|---|---|---|
| *Program* | ::= *Proc ( Program )** | *A program* |
| *Proc* | ::= **proc** $Func_{proc}$ **begin** *Commands* **end** | *Procedure definition* |
| *Block* | ::= **begin** *Commands* **end** | *Block* |
| *Commands* | ::= *Block* | *Block call* |
| | $\mid$ $Func_{proc}$ | *Procedure call* |
| | $\mid$ $Func_{action}$ | *Primitive action call* |
| | $\mid$ *Commands* ; *Commands* | *Sequential composition* |
| | $\mid$ *Commands* **par** *Commands* | *Parallel composition* |
| | $\mid$ *Commands* + *Commands* | *Strict parallel composition* |
| | $\mid$ **if** $Expr_{boolean}$ **then** *Commands* | *Test* |
| | $\mid$ **if** $Expr_{boolean}$ **then** *Commands* **else** *Commands* | *Choice* |
| | $\mid$ **while** $Expr_{boolean}$ **do** *Block* | *Iteration* |
| *Query* | ::= . . . | *Logical expressions* |
| $Expr_j$ | ::= $Func_j$*(Func, Func, . . . , Func)* | *Expressions* |
| *Func* | ::= $Func_{proc}$ | |
| | $\mid$ $Func_{action}$ | |
| | $\mid$ $Func_{fluent}$ | |
| | $\mid$ $Func_{boolean}$ | *Functors* |
| $Func_{proc}$ | ::= *serve( Term ), build( Term ), . . .* | *User-defined names* |
| $Func_{action}$ | ::= **nil** | *Null action* |
| | $\mid$ *up* $\mid$ *move(Term, Term)* $\mid$ . . . | *User-defined primitive actions' names* |
| $Func_{fluent}$ | ::= *at(Term)* $\mid$ *on( Term,* $Func_{fluent}$ *)* $\mid$ . . . | *User-defined fluents* |
| $Func_{boolean}$ | ::= **and**( $Func_{fluent}$ , $Func_{boolean}$ ) | |
| | $\mid$ **or**( $Func_{fluent}$, $Func_{boolean}$ ) | |
| | $\mid$ **not**( $Func_{boolean}$ ) | |
| | $\mid$ $Func_{fluent}$ | *Boolean functions* |
| | $\mid$ *Query* | *Tests on "rigid" information* |
| *Term* | ::= *Ind* $\mid$ *Var* | *Terms can be individuals or variables* |
| *Ind* | ::= . . . | *Individuals identified by the user* |
| *Var* | ::= . . . | *Sorted Variables* |

**Table 1.** The Syntax of OPENLOG.

**Table 2 OPENLOG : Semantics and interpreter**

| | | |
|---|---|---|
| $done(Pr, T_o, T_f)$ | $\leftarrow$ **proc** $Pr$ **begin** $C$ **end** $\wedge\ done(C, T_o, T_f)$ | **[DN01]** |
| $done((C_1 ; C_2), T_o, T_f)$ | $\leftarrow done(C_1, T_o, T_1) \wedge T_1 < T_2 \wedge done(C_2, T_2, T_f)$ | **[DN02]** |
| $done((C_1$ **par** $C_2), T_o, T_f)$ | $\leftarrow done(C_1, T_o, T_1) \wedge done(C_2, T_o, T_f) \wedge T_1 \leq T_f \vee done(C_1, T_o, T_f) \wedge done(C_2, T_o, T_1) \wedge T_1 < T_f$ | **[DN03]** |
| $done((C_1 + C_2), T_o, T_f)$ | $\leftarrow done(C_1, T_o, T_f) \wedge done(C_2, T_o, T_f)$ | **[DN04]** |
| $done(($**if** $E$ **then** $C_1), T_o, T_f)$ | $\leftarrow holdsAt(E, T_o) \wedge done(C_1, T_o, T_f) \vee \neg holdsAt(E, T_o) \wedge T_o = T_f$ | **[DN05]** |
| $done(($**if** $E$ **then** $C_1$ **else** $C_2), T_o, T_f)$ | $\leftarrow holdsAt(E, T_o) \wedge done(C_1, T_o, T_f) \vee \neg holdsAt(E, T_o) \wedge done(C_2, T_o, T_f)$ | **[DN06]** |
| $done(($**while** $\exists L\ (E_b(L)$ **do** $B(L))), T_o, T_f)$ | $\leftarrow (\neg\exists L\ holdsAt(E_b(L), T_o) \wedge T_o = T_f) \vee (holdsAt(E_b(L'), T_o) \wedge done(B(L'), T_o, T_1) \wedge T_o < T_1 \wedge done(($**while** $\exists L\ (E_b(L)$ **do** $B(L))), T_1, T_f))$ | **[DN07]** |
| $done(($**begin** $C$ **end**$), T_o, T_f)$ | $\leftarrow done(C, T_o, T_f)$ | **[DN08]** |
| $done($**nil**$, T_o, T_o)$ | | **[DN09]** |
| $holdsAt($**and**$(X, Y), T)$ | $\leftarrow holdsAt(X, T) \wedge holdsAt(Y, T)$ | **[DN10]** |
| $holdsAt($**or**$(X, Y), T)$ | $\leftarrow holdsAt(X, T) \vee holdsAt(Y, T)$ | **[DN11]** |
| $holdsAt($**not**$(X), T)$ | $\leftarrow \neg holdsAt(X, T)$ | **[DN12]** |
| $holdsAt(X, T)$ | $\leftarrow nonrigid(X) \wedge holds(X, T)$ | **[DN13]** |
| $holdsAt(Q, T)$ | $\leftarrow rigid(Q) \wedge Q$ | **[DN14]** |
| $nonrigid(X)$ | $\leftarrow isfluent(X)$ | **[DN15]** |
| $rigid(X)$ | $\leftarrow \neg isfluent(X)$ | **[DN16]** |

**Table 2.** The Semantics of OPENLOG

## 7 Background Theories

Roughly, a *background theory* ($\mathcal{B}$) is a formal description of actions and properties and the relationships between action-types and property-types.

A background theory consists of two sub-theories: A set of *domain independent axioms* (DI$\mathcal{B}$) (notably the base case of *done* and the definition of *holds*) stating how actions and properties interact. These domain independent axioms also describe how persistence of properties is cared for in the formalism.

The other component of the background theory is a set of *domain dependent axioms* (DD$\mathcal{B}$), describing the particular properties, actions and interrelationships that characterize a domain of application (including the definitions of *initiates*, *terminates* and $isfluent$ ).

The semantics for OPENLOG can be isolated from the decision about what formalism to use to represent actions and to solve the frame problem (the problem of persistence of properties) in the background theory. Formulations based on the Event Calculus [20] and on the Situation Calculus [22][8] are equally well possible. The following one is based on the Event Calculus.

Probably, the most important element in a background theory is the definition of the *temporal projection predicate: holds*.

### 7.1 The Projection Predicate in the Event Calculus

$$\begin{aligned} holds(P,T) \leftarrow\ & do(A,T',T_1) \land initiates(A,T_1,P) \\ & \land T_1 < T \land \neg clipped(T_1,P,T) \qquad \textbf{[EC1]} \\ clipped(T_1,P,T_2) \leftarrow\ & do(A,T',T) \land terminates(A,T,P) \\ & \land T_1 < T \land T \leq T_2 \qquad \textbf{[EC2]} \end{aligned}$$

These axioms are different from most formulations of the EC (in particular [19]) in that the well-known predicate $happens(Event, Time)$ is replaced by the predicate $do(Action, Starting\_Time, Finishing\_Time)$[9].

### 7.2 The Base-Case of *done* in the Event Calculus

As we said before, we use **iffPP** for interpreting OPENLOG programs and generating plans. The execution of those plans is interleaved with their generation and also with the assimilation of inputs from the environment([18], [6]). It is known ([11], [31], [25]) that to make an *abductive theorem prover* [33] behave as a planner, one has to define properly the set of abducibles, say *Ab*. In the

[8] in this case with certain sacrifice in expressiveness, however. The operators + and *par* would have to be excluded from the language as it is.

[9] The intention is to have the name of the agent also represented by a term in the predicate: $do(Agent, Action, Starting\_Time, Finishing\_Time)$. For the sake of simplicity, however, the term for agents is omitted here.

present context one can make $Ab = \{do, <, \leq, =\}$. The background theory can then be completed with the following definition (the base case of *done*):

$$done(A, T_o, T_f) \leftarrow primitive(A) \wedge do(A, T_o, T_f) \text{ [DNEC0]}$$

Notice that we do not include the predicate $preconds(A, To)$ in [DNEC0]. Strictly speaking, one should be "testing" the preconditions of action A at this point. We, however, leave to the programmer the job of testing preconditions within OPENLOG code (i.e. *if C then..* expressions).

### 7.3 How to Achieve the Inhibition of Abduction

As can be seen, the projection predicate *holds* is involved in the interpretation of every conditional expression in OPENLOG. Thus, to inhibit abduction, we simply establish that no *do* atom "derived" by unfolding a *holds* atom will be abduced. In this way, the *holds* predicate is used for "testing", whereas the base case of *done* is used for generation of plans, as we explained above.

## 8 Discussion

OPENLOG is a logic programming language that can be used to write procedural code which can be combined with a declarative specification of a problem domain (a background theory).

To define the language, logical characterization has been given to the traditional programming structures (**if then else, while, ;, ...** ) in such a way that any program written with those structures can be translated into a set of logical sentences.

This mapping from procedural code to logical sentences is not only sought for the sake of clarity. The logic chosen to provide semantics for the procedural structures can also be used to specify a theory of actions that models dynamic universes[6]. This theory of actions can be based on Kowalski and Sergot's Event Calculus [20], a logical formalism with an ontology based on events and properties that can be initiated and terminated by events. The Event Calculus provides a solution to the Frame Problem and also permits the efficient representation of concurrent activities and continuous domains. This has permitted the extension of the capabilities of standard PASCAL to allow for the description of parallel actions in OPENLOG programs.

Thus, the designer/programmer is offered a specification-implementation language that can be used to model complex universes and also to write high-level algorithms to guide the activities of agents acting in a dynamic environment.

As in other logic programming languages, programs in OPENLOG are processed by a theorem prover. Unlike in other approaches, however, programs in OPENLOG are intended to be interpreted[10] rather than compiled[11]. The reason

[10] As in JAVA [23] and other commercial products, where code is pre-compiled to an intermediate form to be read by an interpreter/executive.

[11] As in Situated Agents [29] and GOLOG [21]

for this is crucial. The process of planning (the theorem prover transforming goals into plans) must be interleaved with the execution of those plans and the inputting and assimilation of observations. One has to expect many modifications and amendments of the plans. The system as a whole will process inputs as soon as it can, increasing its chances of an opportune response (normally by an minor adjustment to its plans as illustrated in [6]). The first practical consequence of this is that the system will generate and use *partial plans* which it will refine progressively as its knowledge of the environment increases. This is a crucial difference between OPENLOG's aims and those of a similar logic-based programming language: GOLOG [21]. We have explored the similarities and differences between GOLOG and a previous version of OPENLOG in [5].

Partial planning may seem atypical in the current context because theorem provers are normally backward-reasoning mechanisms. An interesting aspect of the representation here discussed is that it supports planning by searching the time line in a forward direction. This is called *progression*. The representational strategy that supports this form of planning is not new. It is at the core of a well known device to specify grammars and to program their parsers: Definite Clause Grammar or DCG [26]. OPENLOG programs are like DCG*s* in that they both are higher level macros that can be completely and unambiguously translated into logic programs. Unlike DCG however, OPENLOG provides for negative literals.

There is another critical difference between OPENLOG and DCG. In DCGs, the "state of the computation" (which in that case contains the sentence being parsed) is carried along through arguments as is common in stream logic programming. This has the inconvenience of requiring the explicit representation of all objects in the application domain and is, therefore, cumbersome and limiting (we tested the approach in the prototypical implementation of pathfinder reactive automatas that do forward planning, reported in [4]). Background theories are a flexible and powerful alternative to this approach.

## 9 Conclusions and Further Research

OPENLOG is a logic programming language. In OPENLOG one can write procedural code combined with a declarative specification of a dynamic domain (a background theory) to guide an agent at problem-solving in that domain.

The interpreter of OPENLOG is an abductive proof procedure which can be used to implement the planning module of an agent [6]. One innovative aspect of this work is that the agent processing and executing OPENLOG programs will stay *open* to the environment and will allow for changes in its environment and assimilation of new information generated by these changes.

Another novelty in this work is that we use a logic program (the definition of *done* and the other predicates) to specify the semantics of an imperative programming language. The semantics is provided as a mapping that links the semantics of the imperative code with any semantics for abductive logic programs. The definition of *done* has some other operational advantages. It can serve as an interpreter for OPENLOG, thus providing its operational semantics as well. And it can be used to "inhibit" the abductive proof procedure and prevent the over-generation of abducibles which would make of abduction an impractical approach for building the planning module of an agent.

We are exploring the relationship between OPENLOG and programming with integrity constraints [6]. Also in [6], "the Elevator example" is borrowed from [21] and is developed in with OPENLOG. We plan to use OPENLOG as the programming language for each agent in a platform to simulate multi-agents systems.

## Acknowledgments

This research was supported by a grant from CONICIT-University of Los Andes, Venezuela. The author would like to thank Bob Kowalski, Fariba Sadri and Murray Shanahan for many interesting discussions and the referees for their useful comments.

## References

1. James F. Allen. Temporal reasoning and planning. In J. F. Allen, H. Kautz, R. Pelavin, and J. Tenenberg, editors, *Reasoning About Plans.* Morgan Kauffmann Publishers, Inc., San Mateo, California, 1991. ISBN 1-55860-137-6.
2. E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Menlo Park, CA, 1985.
3. L. Console, T.. Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 2(5):661–690, 1991.
4. Jacinto A. Dávila. Knowledge assimilation in multi-agents system. Master's thesis, Imperial College, London, September 1994.
5. Jacinto A. Dávila. A logic-based agent. Technical report, Imperial College, London, February 1996.
6. Jacinto A. Dávila. *Agents in Logic Programming.* PhD thesis, Imperial College, London, May 1997.
7. J. de Kleer. An assumption-based tms. *Artificial Intelligence*, 32, 1986.
8. Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *AAAI 88: The Seventh National Conference on AI*, volume 1, Saint Paul, Minnesota, August 1988.
9. M. Denecker and D. De Schreye. Sldnfa: an abductive procedure for abductive logic programs. The journal of logic programming. 1995.
10. K. Eshghi and R. Kowalski. Abduction compare with negation as failure. In G. Levi and M. Martelli, editors, *Proceedings of the International Conference on Logic Programming*, pages 234–255, Lisbon, Portugal, 1989. MIT Press.
11. Kave Eshghi. Abductive planning with event calculus. In *Proceedings 5th International Conference on Logic Programming*, 1988. pg. 562.
12. C.A. Evans. Negation as failure as an approach to the Hanks and McDermott problem. In F.J. Cantu-Ortiz, editor, *Proc. 2nd. International Symposium on Artificial Intelligence*, Monterrey, México, 1989. McGraw-Hill.

13. T Fung and R Kowalski. The iff proof procedure for abductive logic programming. The Journal of logic programming, 33(2): 151-178, 1997.
14. Tze Ho Fung. *Abduction by deduction.* PhD thesis, Imperial College, London, January 1996.
15. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
16. A.C. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
17. A.C. Kakas and P Mancarella. Abductive logic programming. In W. Marek, A. Nerode, D. Pedreschi, and V.S. Subrahmanian, editors, *Proc. NACLP Workshop on Non-monotonic Reasoning and Logic Programming*, Austin, Texas, 1990.
18. Robert Kowalski. Using metalogic to reconcile reactive with rational agents. In K. Apt and F. Turini, editors, *Meta-Logics and Logic Programming.* MIT Press, 1995. (Also at http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/recon-abst.html).
19. Robert Kowalski and Fariba Sadri. The situation calculus and event calculus compared. In M. Bruynooghe, editor, *Proc. International Logic Programming Symposium*, pages 539–553. MIT Press, 1994. (Also at http://www-lp.doc.ic.ac.uk/UserPages/staff/fs/ilps94.html).
20. Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.
21. H. Levesque, R. Reiter, Y. Lespérance, L. Fangzhen, and R. B. Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, (31):59–84, 1997.
22. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.
23. Sun Microsystems. Hotjava home page. http://java.sun.com/.
24. Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
25. Lode Missiaen, Maurice Bruynooghe, and Marc Denecker. Chica, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5(5):579–602, October 1995.
26. F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis-a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.
27. D. Poole. Explanation and prediction: an architecture for default and abductive reasoning. *Computational Intelligence Journal*, 5:97–110, 1989.
28. David Poole. Logic programming for robot control. In Chris S. Mellish, editor, *Proc. International Joint Conference on Artificial Intelligence*, pages 150–157, San Mateo, California, 1995. Morgan Kaufmann Publishers, Inc.
29. Stanley J. Rosenschein and Leslie Pack Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73:149–173, February 1995.
30. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs - New Jersey, 1995.
31. Murray Shanahan. Prediction is deduction but explanation is abduction. In N.S. Sridharan, editor, *Proc. International Joint Conference on Artificial Intelligence*, pages 1055–1060. Morgan Kaufmann, Detroit. Mi, 1989.
32. Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.
33. Francesca Toni. *Abductive Logic Programming.* PhD thesis, Imperial College, London, July 1995.
34. M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 4(4):733–742, 1976.

35. Gerhard Wetzel. *Abductive and Constraint Logic Programming.* PhD thesis, Imperial College, London, March 1997.
36. Gerhard Wetzel, Robert Kowalski, and Francesca Toni. A theorem-proving approach to clp. In A. Krall and U. Geske, editors, *Workshop Logische Programmierung*, number 270, pages 63–72. GMD-Studien, September 1995.