ACTILOG: An Agent Activation Language

Jacinto A. Dávila

Centro de Simulación y Modelos (CESIMO)
Universidad de Los Andes. Mérida. Venezuela
jacinto@ula.ve
http://cesimo.ing.ula.ve/j̃acinto
FAX: +58 274 2402811

Abstract. ACTILOG is a language to write generalized **condition** → **action** activation rules. We propose it as an alternative and a complement to OPENLOG [6], another agent logic programming language for an abductive reasoner. We want to show how implications (conditional goals) can be used to state integrity contraints for an agent. These integrity contraints describe conditions under which the agent's goals must be reduced to plans that can be executed. For instance, a rule such as **if** A **then** B, will indicate to the agent that whenever it can prove that A is the case, it then should pursue goal B. B is normally the description of a task that must be reduced to a set of low-level, primitive actions that the agent can execute.

1 Introduction

ACTILOG is a language to write generalized **condition** → **action** activation rules. We propose it as an alternative and a complement to OPENLOG [6], another agent logic programming language for an abductive reasoner. The design is such that any semantics for abductive logic programs could be taken as the basic semantics for the programming languages ACTILOG and OPENLOG. In this way, we build upon existing formalizations of abductive reasoning and abductive logic programming[10].

Our objectives are similar to those of the IMPACT project [8] and their **taps** (Temporal Agent Programs), i.e. to program agents using the most expressive knowledge representation. However, instead of pursuing a characterization of the model-theoretic semantics, we have aimed first towards the description of the reasoning mechanisms based on abduction.

In previous work [4], [6], we suggested that the process of *activation of goals* in an agent could be understood as the derivation of unconditional goals from integrity constraints. Here, we want to show how implications (conditional goals) can be used to state integrity constraints for an agent. These integrity constraints describe conditions under which the agent's goals must be reduced to plans that can be executed. For instance, a rule such as **if** A **then** B, will indicate to the agent that whenever it can prove that A is the case, it then should pursue goal B. B is normally the description of a task that must be reduced to a set of low-level, primitive actions that the agent can execute.

ACTILOG is similar to other well-known *production-rule* languages (such as OPS5 [3]). A first difference with respect to previous work is that ACTILOG, as OPENLOG [6], relies on a general purpose representation of actions and events (i.e. a logic of actions) in the form of background theories. Temporal and common-sense reasoning about initiation and termination of properties is, as we have seen, possible within this framework.

A second important difference (with respect OPS5, in particular) is that ACTILOG is an *object-level* language[1]. It does not include syntactic constructs like **goal G** or **plan P**. These characterizations are provided by the architecture of the agent [11].

ACTILOG is intended as a language to write declarative sentences stating the relations between observations and subsequent actions to be performed in response to those observations. These sentences are regarded as *integrity constraints* for the behaviour of the agent, in close analogy to integrity constraints for information stored in a database. All the control devices required to interpret and verify integrity constraints are provided by the proof procedure that characterizes the reasoning mechanism of the agent [4], [9].

The enriched syntax of ACTILOG (with respect to languages that allow simple implications with atomic heads) supports the arrangement of the activating conditions so as to minimize redundant processing. The head of an implication can be almost any logical sentence (including implications) and thus it is possible to write, not only sentences of the form: $(A \leftarrow B) \leftarrow C$, but also sentences such as $((A \leftarrow B) \wedge (C \leftarrow D)) \leftarrow E)$, where E is a condition shared by both nested implications. This captures some of the functionalities of the RETE algorithm which has been used to improve the efficiency of the OPS5 platform (.ibid).

The following two sections describe ACTILOG in detail. Afterwards, We compare the language with OPENLOG and discuss the advantages of each.

2 Syntax of ACTILOG

The syntax of ACTILOG is presented in table 1 in a variant of the BNF form. The conventions to read the table are the same as in normal BNF. Notably, C^* represents zero or more occurrence of the category symbol C. As in OPENLOG, ACTILOG's syntax is open so that the programmer can include fluent and action names into the language. Actually, all the lower level syntactic categories, including boolean fluents, are borrowed from [6].

The top-most syntactic category is $UNIT$. A "unit" in ACTILOG gathers a set of activation rules (defined by ACT_Rule) related to a particular task. Below (in fig. 1), we give an example of ACTILOG encoding by translating the instructions for an elevator controller.

Another important category in the syntax is $Quants$. It stands for the subexpression in an ACT_rule that specifies which variables are existentially and universally quantified.

[1] As it is the case with OPENLOG.

Table 1 ACTILOG Language: Syntax		
Unit	::= *Set* **to** *TaskName*	*Activation Unit*
Set	::= *Act_Rule (***and** *Set)**	*Activation Set*
Act_Rule	::= *Quants* **if** *Body* **then** *Head*	*Basic Activation Rule*
Quants	::= *One_Quant**	*Quantifiers*
One_Quant	::= $\exists$ *Var*	*One Quantifier*
	| $\forall$ *Var*	
Body	::= *Condition (***and** *Body)**	*Body of an IC*
Head	::= *Disjunct (***or** *Head)**	*Head of an IC*
Disjunct	::= *Set*	
	| *Task*	
	| **false**	
Condition	::= $Func_{fluent}$ **at** *Term*	*Conditions*
	| *Task*	
	| **not** *Condition*	
	| *Query*	*Tests on "rigid" information*
Task	::= *TaskName Schedule*	*Task descriptions*
Schedule	::= *Schedule* **and** *Schedule*	*Schedules*
	| **at** *Term* | **before** *Term*	
	| **after** *Term* | **starting at** *Term*	
	| **finishing at** *Term*	
	| **starting before** *Term*	
	| **finishing before** *Term*	
	| **starting after** *Term*	
	| **finishing after** *Term*	
TaskName	::= $Func_{action}$ | $Func_{proc}$	*Action names*
	| *TaskName (***;** *TaskName)**	
	| *TaskName (***par** *TaskName)**	
$Func_{action}$	::= ...	*As in OPENLOG*
$Func_{proc}$	::= ...	*As in OPENLOG*
$Func_{fluent}$	::= ...	*As in OPENLOG*
$Func_{boolean}$	::= ...	*As in OPENLOG*
Term	::= *Ind* | *Var*	*As in OPENLOG*
Ind	::= ...	*As in OPENLOG*
Var	::= ...	*As in OPENLOG*

Table 1. Syntax of ACTILOG

Variables for which quantification is not indicated are assumed as universally quantified and their scope of quantification is the whole activation unit. This means that the scope of the variable so *implicitly* quantified *will include the scope of quantification of the other variables*. This aspect must be emphasized because it implies that existentially quantified variables will depend on those implicitly quantified variables for *skolemization*, as shown in example 1:.

Example 1. Consider the ACTILOG rule:

```
exists T1 if on(N) at T and T lt T1 then serve(N) at T1
```

It should be read as: $\forall N\ \forall T\ \exists T1(serve(N,T1) \leftarrow on(N,T) \wedge T < T1)$.

So, in clausal form one would write: $serve(N, f(N,T)) \leftarrow on(N,T) \wedge T < f(N,T)$, where $f(N,T)$ is a skolem function.

Thus, the syntax of ACTILOG takes it beyond the realm of Horn clauses extended with negation. One can now have existentially quantified variables in the *head* of the clause. The implications of this are discussed in the following section.

The other syntactic categories are better understood by the translation of the rules into integrity constraints involving the predicates $holds(P,T)$ and $done(A,T_o,T_f)$. This is the purpose of tables 3, 4 and 5 in the following sections. Before that, however, we include the semantics specification of OPENLOG for easy reference.

2.1 The Semantics of OPENLOG revisited

As in [6], the basic semantics of OPENLOG is shown in table[2] 2 by means of the predicate *done*[3]. That is, we employ a indirect mechanism: the definition of a predicate, to state the semantic. Basically, it is a mapping from our languages into normal logic programs. Thus, OPENLOG code inherits existing semantics for logic programs, including, we presume, those semantics for *positive* **taps** [8].

Recall that 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". One of the innovations in OPENLOG was 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* from the execution environment.

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*, a theory of change that, as we have shown in [6, 4], can be based on the Situation Calculus [14] or the Event Calculus [12].

[2] PROLOG-like syntax is being used.

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

Table 2 OPENLOG – ACTILOG : 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**]
$done((\textbf{if}\ E\ \textbf{then}\ C_1)$, $T_o, T_f)$	$\leftarrow\ holdsAt(E, T_o)\ \wedge\ done(C_1, T_o, T_f)$ $\vee\ \neg holdsAt(E, T_o)\ \wedge T_o = T_f$	[**DN05**]
$done((\textbf{if}\ E\ \textbf{then}\ C_1$ **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**]
$done((\textbf{while}$ $\exists L\ (E_b(L)$ **do** $B(L))), T_o, T_f)$	$\leftarrow\ (\neg\exists L\ holdsAt(E_b(L), T_o)$ $\wedge\ T_o\ =\ T_f)$ $\vee\ \ (holdsAt(E_b(L'), T_o)$ $\wedge\ done(B(L'), T_o, T_1)$ $\wedge\ T_o < T_1$ $\wedge\ done((\textbf{while}$ $\exists L\ (E_b(L)$ **do** $B(L))\ ), T_1, T_f))$	[**DN07**]
$done((\textbf{begin}\ C\ \textbf{end})$, $T_o, T_f)$	$\leftarrow\ done(C, T_o, T_f)$	[**DN08**]
$done(\textbf{nil}, T_o, T_o)$		[**DN09**]
$holdsAt(\textbf{and}(X, Y), T)$	$\leftarrow\ holdsAt(X, T)\ \wedge\ holdsAt(Y, T)$	[**DN10**]
$holdsAt(\textbf{or}(X, Y), T)$	$\leftarrow\ holdsAt(X, T)\ \vee\ holdsAt(Y, T)$	[**DN11**]
$holdsAt(\textbf{not}(X), T)$	$\leftarrow\ \neg holdsAt(X, T)$	[**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 and ACTILOG

ACTILOG also allows for *composite task names*, using the operators "**;**" and "**par**" (and we could also add "+"). The idea is to borrow part of the definition of *done* in table 2 to deal with these. However, for the sake of simplicity we omit these operators in the semantics of ACTILOG.

3 The semantics of ACTILOG

It must be evident at this stage that OPENLOG and, now ACTILOG, are no more than "syntactic sugar" for logic (traditional logic programming in the case of OPENLOG). The exercise of defining these languages is important, however, because it helps to clarify what logical concepts are involved in programming an agent.

Thus, as with OPENLOG, to understand the meaning of any ACTILOG unit, one must restore it to its underlying logical form. Unlike OPENLOG programs however, an ACTILOG unit cannot be transformed into a normal logic program, without losing expressiveness. This is due to the fact that existential quantification is highly restricted in logic programs. We must use a richer form of logic that admits explicit quantification of variables and a more complex sentence structure.

Nevertheless, this is not a problem in our system because it is based on the **iff** abductive proof procedure[9], **iff**PP, which can accommodate a more general structure for implications (conditional goals). However, a few functionalities must be added to the specification of **iff**PP to support the agent programming language. The inference rules of the proof procedure remain the same except for **splitting of implications** and **case analysis**, which must now include a new set of conditions for their application. This rule is not applied if there are universally quantified variables in the head of an implication. The reason for this, which also applies to the rule of *case analysis* is analogous to the reason for skolemization and is better explained by example 2, a follow-up to example 1:

Example 2. Suppose that we split:

$$\forall N\ \forall T\ \exists T1((serve(N,T1) \wedge T < T1) \leftarrow on(N,T)) \tag{1}$$

We will end up with:

$$\forall N\ \forall T\ \exists T1((serve(N,T1) \wedge T < T1) \vee (\mathbf{false} \leftarrow on(N,T))) \tag{2}$$

The reason not to split the sentence in this example is that the first disjunct in the resulting sentence ($serve(N,T1)$) cannot be incorporated into the unconditional goals (as it should be), because it involves the universally quantified variable N. If one insists on doing so, the proof procedure will treat N as existentially quantified. Note that whether this yields incorrect answers depends on the rest of the formalization (in particular on the definition of *serve*).

However, it remains a problem that the system is losing the *dependency between existential and universal quantification.* One can see this by looking

back at the clausal form of the sentence in the example 2: $serve(N, f(N,T)) \leftarrow on(N,T) \wedge T < f(N,T)$, which after splitting leaves $serve(N, f(N,T))$ as a separate disjunct. The value of the second argument of *serve* is *determined*, not only by N but also by T.

Of course, nothing has been lost if one keeps the dependency by appealing to the skolem function ($f(N,T)$). However, this would imply significant modifications to the proof procedure. The use of skolemization has been attempted before (see Denecker and De Schreye's SLDNFA [7] for a system similar to **iff**PP, but that uses skolemization) and it has proved to be cumbersome and inefficient.

However, one can reach a proper compromise with the following strategy: The proof procedure will preserve the dependencies between variables in the implications and will be **banned** from splitting (or doing case analysis) on any implication the head of which contains variables with **active dependencies**.

The concept of **active dependency** is simple. The dependency between $T1$ and N and T above is active if N and T, in that implication, have not been assigned known constant values. For instance, when, by propagation of $on(3,1)$, the implication above becomes $\exists T1(serve(3,T1) \leftarrow 1 < T1)$ then this can safely be handled by splitting because $T1$ is now as defined as it can be by skolemization ($T1 = f(3,1)$).

Observe that, for this strategy, the only extension required in **iff**PP is a list of "dependencies" between variables in the implications. A list which could be built by straightforward parsing of the quantifiers in the original integrity constraints. To make the process easier, we restrict the quantifiers in the ACTILOG rules to appear as shown in table 1.

All this explained, we can now show how to transform ACTILOG units into sets of integrity contraints for agent programming. The procedure is described by a normal (meta-)logic program in tables 3, 4 and 5. To simplify the presentation the syntax of the logic programs is slightly relaxed. "{}" represents both empty categories in ACTILOG and empty formulae. The predicate *append*/3 has the usual interpretation.

4 OPENLOG versus ACTILOG

OPENLOG and ACTILOG **are not** exactly alternative solutions for the same problem. We need a mechanism for the activation of goals in an agent. ACTILOG allows the triggering of tasks at anytime, even if the task itself is defined by OPENLOG procedures. In the examples above *serve* could be defined by an OPENLOG procedure, but we will still need the aforementioned ACTILOG rule to activate the goal.

If we do not allow for activation of goals (i.e. we do not want to use ACTILOG, only OPENLOG), the agent would have to have one, top-most main goal from which all the possible activities of the agent are derived. This is the solution in GOLOG[13]. Ours is closer to the *forward-chaining-like solution* in the **taps**[8], without the overhead of model-theoretic computations.

Table 3 ACTILOG translation into Integrity Constraints

$rewrite\_activa\_ic(Set$ **to** $TaskName, IC)$
$\quad \leftarrow\ transform(Set, IC)$ [**RW – ACTI**]

$transform(QVars\ FRule$ **and** $RestRules,$
$\quad NQVars(NewFRule \wedge NRestRules))$
$\quad \leftarrow\ transform(\{\}\ FRule, QVarsFR\ (NewFRule))$
$\quad \wedge\ transform(\{\}\ RestRules, QVarsRest\ (NRestRules))$
$\quad \wedge\ transform\_quantifiers(QVars, QVars')$
$\quad append(QVarsFR, QVarsRest, QVarTemp)$
$\quad append(QVars', QVarTemp, NQVars)$ [**TRSET**]

$transform(QVars$ **if** $Body$ **then** $Head,$
$\quad NQVars(NewHead \leftarrow NewBody))$
$\quad \leftarrow\ transform(\{\}\ Body, \{\}\ NewBody)$
$\quad \wedge\ transform(QVars\ Head, NQVars\ (NewHead))$ [**TRRULE**]

$transform(\{\}\ Condition$ **and** $RestConds,$
$\quad \{\}(holds(P, T) \wedge NRestCond))$
$\quad \leftarrow\ Condition = P$ **at** T
$\quad \wedge\ is\_fluent(P)$
$\quad transform(\{\}\ RestCond, \{\}\ (NRestCond))$ [**TRCOND – FL**]

Table 3. Translating ACTILOG rules into Integrity Constraints (Part 1)

Table 4 ACTILOG translation into Integrity Constraints

$transform(\{\}\ Condition\ \textbf{and}\ RestConds,$
$\quad \{\}(done(Name, T_1, T_2) \wedge LogSched \wedge NewRC))$
$\quad \leftarrow\ Condition = Name\ Schedule$
$\quad \wedge\ actionname(Name)$
$\quad \wedge\ transform\_schedule(T_1, T_2, Schedule, LogSched)$
$\quad \wedge\ transform(\{\}\ RestConds, \{\}\ (NewRC))$ [**TRCOND – ACT**]

$transform(\{\}\ \textbf{not}\ Condition, \{\}\neg(NewCond))$
$\quad \leftarrow\ transform(\{\}\ Condition, \{\}\ (NewCond))$ [**TRCOND – NOT**]

$transform(QVars\ Disjunct\ \textbf{or}\ RestDisj,$
$\quad NQVars(NewDisj \vee NewRD))$
$\quad \leftarrow\ transform(\{\}\ Disjunct, Vars1\ (NewDis))$
$\quad \wedge\ transform(\{\}\ RestDisj, ReVars\ (NewRD))$
$\quad \wedge\ transform\_quantifiers(QVars, LogQVars)$
$\quad append(Vars1, ReVars, QVarTemp)$
$\quad append(LogQVars, QVarTemp, NQVars)$ [**TRHEAD – OR**]

$transform(QVars\ Task,$
$\quad NQVars(LogSched \wedge done(TaskName, T_1, T_2)))$
$\quad \leftarrow\ Task = TaskName\ Schedule$
$\quad \wedge\ transform\_schedule(T_1, T_2, Schedule, LogSched)$
$\quad \wedge\ transform\_quantifiers(QVars, QVars')$
$\quad append(QVars', \{\exists T_1\ \exists T_2\}, NQVars)$ [**TRHEAD – ACT**]

Table 4. Translating ACTILOG rules into Integrity Constraints (Part 2)

Table 5 ACTILOG translation into Integrity Constraints	
$transform\_quantifiers(\{\},\{\})$	[**TRQU1**]
$transform\_quantifiers(\textbf{exists}\ V\ RestQV, \exists V\ RestQV')$ $\leftarrow\ var(V) \wedge\ transform\_quantifiers(RestQV, RestQV')$	[**TRQU2**]
$transform\_schedule(T_o, T_f, \textbf{at}\ T, T\ \textbf{le}\ T_o \wedge T\ \textbf{lt}\ T_f)$	[**TRSCH1**]
$transform\_schedule(T_o, T_f, \textbf{before}\ T, T_o\ \textbf{lt}\ T \wedge T_f\ \textbf{le}\ T)$	[**TRSCH2**]
$transform\_schedule(T_o, T_f, \textbf{after}\ T, T\ \textbf{le}\ T_o \wedge T\ \textbf{lt}\ T_f)$	[**TRSCH3**]
$transform\_schedule(T_o, T_f, \textbf{starting at}\ T, T_o\ \textbf{eq}\ T \wedge T_f\ \textbf{lt}\ T)$	[**TRSCH4**]
$transform\_schedule(T_o, T_f, \textbf{finishing at}\ T, T_o\ \textbf{lt}\ T \wedge T_f\ \textbf{eq}\ T)$	[**TRSCH5**]
$transform\_schedule(T_o, T_f, \textbf{starting before}\ T, T_o\ \textbf{lt}\ T)$	[**TRSCH6**]
$transform\_schedule(T_o, T_f, \textbf{finishing before}\ T, T_f\ \textbf{lt}\ T)$	[**TRSCH7**]
$transform\_schedule(T_o, T_f, \textbf{starting after}\ T, T\ \textbf{lt}\ T_o)$	[**TRSCH8**]
$transform\_schedule(T_o, T_f, \textbf{finishing after}\ T, T\ \textbf{le}\ T_f)$	[**TRSCH9**]

Table 5. Translating ACTILOG rules into Integrity Constraints (Part 3)

ACTILOG rules contribute to keep the agent *open* to its environment (as we show below). Thus, the programmer will normally have to use ACTILOG and OPENLOG to program the agent.

```
if currentfloor(M) at T and on(N) at T then (
        if M eq N then open par turnoff(N); close after T
    and if M lt N then addone(M,Nx); up(Nx) after T
    and if N lt M then subone(M,Nx); down(Nx) after T )
```

Fig. 1. ACTILOG rule for a simple elevator controller

The ACTILOG rule in figure 1 provides a solution for an simple *reactive* elevator-controller agent.

Observe that an ACTILOG "unit" will have neither recursive call, nor **while** statements. The iterative reasoning is generated by the architecture of the agent, i.e. by the *cycl*ing in which the whole system is engaged (as explained in [11]).

An ACTILOG unit is more *open* to the environment than a OPENLOG procedure because *cycle* will check the environment on each iteration and new information will be constantly arriving. There is less interaction with the environment when one has a **while** statement in a OPENLOG procedure which is being unfolded. By using **while**, one is introducing an iterative process in addition to (and without the benefits of interaction with the environment of) the iterative process generated by *cycle*. It is like having a loop within a loop, with the inconvenience that the "included-loop" (the *demo* predicate in [11] pro-

cessing the **while** statement) is not forced to check the environment on every iteration, as *cycle* is.

Notice that this is the case even if **while** statements can be interrupted to assimilate inputs. To achieve the same number of "tests" on the environment per unit of time, one would have to to force the program processing the **while** to suspend processing after each iteration. In ACTILOG, *cycle* defines the only iterative mechanism. No "loops within loops" can affect the interaction with the environment.

In addition, ACTILOG units can support "planning ahead". Actions will be promoted from the head of implications to *the bag of abducibles* and, after that, they will be "firing" implications and triggering subsequent actions.

There still is one more advantage in ACTILOG due to the fact the we are using the **iff** abductive proof procedure. Plans generated from ACTILOG rules, in contrast to those obtained from OPENLOG procedures, *can be made to contain a minimal set of abduced steps*. The checking of preconditions can be done in the body of the implications, where abduction is not allowed by the proof procedure. This form of precondition testing blurs the distinction between triggering conditions and proper preconditions of actions. However, by using ACTILOG only, we will not have to inhibit the abductive process to cater for "over-generation of abducibles". The problem is explained in [6].

Thus, OPENLOG and ACTILOG, in the context of abductive logic programs, *could be* alternative solutions for the same problem (i.e. both could be used to generate the same behaviour in the agent) if OPENLOG is accompanied by a mechanism to inhibit abduction. All these advantages suggest that ACTILOG is a more expressive programming framework than OPENLOG and, perhaps, that integrity constraints are equality related to traditional logic programs.

There are however, points in favour of using OPENLOG as the programming language (or even better, a combination of OPENLOG and ACTILOG. We followed this approach in the prototype).

The first advantage comes from Software Engineering. For complex tasks and domains, the set of integrity contraints can be very large and difficult to arrange as one "unit". In those circumstances, a more "modular" approach, for instance with procedures in OPENLOG, could be more advisable.

The second advantage is related to the first but is more subtle. In OPENLOG procedures, the ultimate goal being pursued can always be inferred from the code of the procedures. For instance, in the elevator example, once $on(3,1)$ triggers the goal $1 < T_1 \wedge serve(3,1,T_1)$, the goal $serve(3,T_2,T_1) \wedge 1 < T_2$ can be inferred from the other literals involved. These literals are part of the agent's goals while the agent is trying to achieve "serving the third floor by T_1". Thus, having information about which higher goal the agent is aiming to (and how much is still to be done to achieve it) in a partial plan is easier in OPENLOG.

This kind of information can be particularly useful when the system is using heuristics to guide its search process and when it is trying to decide on the importance or urgency of its goals.

But even this can be done, to some extend, in ACTILOG, although by appealing to an extra-logical resource. In the first category in table 1, a $Unit$ could be characterized by a Set and a $TaskName$ ($Unit ::= Set$ **to** $TaskName$), where $TaskName$ indicates the ultimate goal at which the integrity contraints in Set are aiming.

This is an extra-logical device because $TaskName$ is lost in the translation of ACTILOG rules into integrity contraints that define their semantics. However, if one maintains this "label" attached to the ACTILOG unit, one could identify the tasks that have been triggered and reason about their state of planning and execution.

Of course, this is not the only way of knowing about pending tasks. One could also use "state encoding", as described by Allen [1]: within the language, one would introduce the fluent $serving(N, T)$, initiated by the observation $on(N, T)$, and this would be enough for the agent to know which the on-going tasks are.

One last remark about ACTILOG and the activation of goals. Observe that, from the perspective of a reactive agent, *there may be no need* to remember which higher goal the agent is planning and acting for. For instance, in the case of the elevator controller, the agent does not need to remember $serve(3, T)$, activated by $on(3, T')$ for some $T' < T$.

If the signal stays on "outside in the environment", the agent will be able to realize that the task is still pending if it fails to reach its higher goal ($serve(3, T)$ in this case) with the first (re-)actions. It is as if the agent is using the "world as its own model" [2] and so, representations (memory) of inputs and goals (such as records of the signals and the triggered tasks) will not be necessary. In a "cooperating" environment like that, an agent needs fewer deliberative resources in order to be efficient and effective. We are exploiting this possibility in the implementation.

The following section discuss the logic of activation of goals with one example to illustrate how the reactive nature of integrity constraints can be combined with planning.

5 Activation of goals for planning

The purpose of activating a goal is to have the agent plan actions to achieve it. As we discussed above, sometimes the environment is such that the agent does not need to plan. In those cases, reactivity becomes more important in producing sensible behaviour, and then simple integrity constraint or ACTILOG rules are sufficient to generate that behaviour.

However, the "reactive" use of integrity constraints to activate goals could be a source of inadequate or improper behaviour. This could be the case, for instance, if the agent continues executing a plan that it has devised to achieve an "activated" goal, even though the "activating" conditions have ceased to hold.

To illustrate this, let us use the context of the following example. Imagine that the goal:

$$\exists T_1 \; \exists T2 \; (0 < T_1 \; \wedge \; serve(2, T_1, T_2) \;) \tag{3}$$

has been activated from the implication:

$$\exists T_1\ \exists T2\ (T < T_1\ \wedge\ serve(N, T_1, T_2) \leftarrow\ obs(on(N), T)\ ) \tag{4}$$

by the input: $obs(on(2), 0)$

Also imagine that half-way through the execution of the corresponding plan, the signal at floor 2 is turned off. The agent observes this, because it has interrupted its reasoning to try the first action of the plan, and the information about the new status of the signal arrives as "feedback".

It would be incorrect[4] for the elevator to keep executing this plan as its motivating condition (that the signal was "on" and the floor ought to be served) has vanished.

The problem is that the elevator (executing an OPENLOG "serve" procedure and with the integrity constraint 4 above) has no means of deducing that the plan is now unnecessary and must be abandoned, until it actually tries the *turnoff* action (which will fail because the signal is not "on").

We can solve this problem in several ways with our agent architecture. We discuss one general[5] and one specific solution below.

A general solution is to include this version of the axiom [DNEC0] which includes an explicit test of all the preconditions of all the primitive actions, like this:

$$\begin{aligned} done(A, T_o, T_f) \leftarrow\ & primitive(A)\ \wedge\ preconds(A, T_o) \\ & \wedge T_o \leq T_f\ \wedge\ do(A, T_o, T_f) \qquad\qquad [\mathbf{DNEC0'}] \end{aligned}$$

The axiom [DNECO'] would allow for the "clipped" constraint to be produced and used by the planner to falsify the plan. If the agent completes that plan up to the point where the preconditions of *turnoff* are reasoned about, it will "realize" (before trying to execute it) that the action *turnoff(2)* is going to fail (precisely because the elevator assumes that the signal will not be "on" at that floor). This is the reason to drop the plan.

Notice that we are assuming here that either some action of the plan has been executed or the planner has access to some mechanism to handle inequalities and time-constraints involving the current time. As we said in that section, this inequality-handling mechanism could be combined with a mechanism to evaluate agent's action preferences. One could also maintain an explicit record of the goal that has been activated and its activating condition as "contextual" information.

That "general" solution to the problem of activating conditions that ceased to hold (leaving "triggered" plans *without justification for their execution*) could be expected to be inefficient. This is because the planner needs to "complete" the plan up to the point where the constraints on the preconditions of the actions are made explicit (e.g. the constraint **false** $\leftarrow$ $clipped(0, on(2), T_3)$ must be derived by the planner, before it can be used to test whether the precondition persists).

[4] with respect to an idealised model of perfect rationality with no resource constraints for reasoning.

[5] General solution for those cases when the "motivating" condition (e.g. *on(2)* above) is also the precondition of some action in the plan (as in the case of *turnoff(2)* above).

One could improve the efficiency of the planner by providing a more precise and informative integrity constraint to activate the "serve" goal. This would be a specific solution because it uses knowledge specific to the problem. For instance, after introducing a new abducible predicate[6] *serving*, the constraints:

$$\begin{aligned}
&\exists T_1\ \exists T_2\ (\ (T < T_1\ \wedge\ serving(N,T,T_2)\ \wedge\ serve(N,T_1,T_2)\ ) \\
&\qquad\qquad \leftarrow\ obs(on(N),T)\ ) \\
&\wedge\ (\mathbf{false} \leftarrow\ (serving(N,T_3,T_4)\ \wedge\ do(S,\mathit{turnoff(N)},T_0,T_f) \\
&\qquad\qquad \wedge\ S \neq self\ \wedge\ T_3 \leq T_0\ \wedge\ T_f \leq T_4\ )\ )
\end{aligned}$$

will have any plan to achieve the goal $serve(N,T_1,T_2)$ falsified, if an event that switches the signal off (presumably other agent doing it) is observed before the plan is executed by *this* agent[7]

Thus, integrity constraints do support some basic, rational behaviour in a multi-agent, dynamic environment. Whether they can be extended to cater for more complex cases of coordination and cooperative behaviour requires further investigation.

6 Conclusions and further work

This article and the previous one [6] have presented a family of extended logic programming languages to program an agent. The characteristic common to all these languages is that their sentences have an unambiguous translation into subsets of first order logic. In the case of OPENLOG, the translation has a more restrictive output, yielding *normal logic programs*. In the case of ACTILOG the translation is into a form that supports sentences formalizing integrity contraints, that can be used to guide the process of activation of goals in the agent. Both programming languages have an operational semantics closely related to the specification of an abductive proof procedure [9].

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). ACTILOG complements that language by providing for integrity constraint and activation rules for the agent. However, ACTILOG can also be used to state integrity constraints as an alternative representation of procedural descriptions.

We plan to complete the family of language to program agents, with means to represent agent preferences and priorities. We will also offer this family of programming languages in a platform to simulate multi-agents systems [5].

---

[6] This means that the *bag of abduced atoms* will contain $\{do, =, <, obs, serving\}$. The introduction of *serving* could be regarded as an instance of "state-encoding" as discussed by Allen in [1] and also mentioned in the previous section.

[7] Here we also assume that there is a mechanism to deduce that the turning off of the signal does occur after the instant when the goal is activated ($T_3 \leq T_0$) and before the plan is completed ($T_f \leq T_4$).

7 Acknowledgments

This work has been partially funded by Fonacit-CDCHT-University of Los Andes, projects I-524-95-02-AA, I-667-99-02-B and S1-2000000819.

References

1. James F. Allen, *Temporal reasoning and planning*, Reasoning About Plans (J. F. Allen, H. Kautz, R. Pelavin, and J. Tenenberg, eds.), Morgan Kauffmann Publishers, Inc., San Mateo, California, 1991, ISBN 1-55860-137-6.
2. Rodney Brooks, *Intelligence without representation*, Artificial Intelligence (1991), 139–159.
3. Lee Brownston, *Programming expert systems in ops5*, Addison-Wesley Inc., USA, 1985.
4. Jacinto Dávila, *Agents in logic programming*, Ph.D. thesis, Imperial College, London, UK, 1997.
5. Jacinto Dávila and Mayerlin Uzcátegui, *Galatea: A multi-agent simulation platform*, The best of AMSE (C. Berger-Vachon and A.M. Gil lafuente, eds.), http://www.amse-modelling.org/Periodical_AMSE.html, AMSE, Barcelona, Spain, 2000.
6. Jacinto A. Dávila, *Openlog: A logic programming language based on abduction*, Lecture Notes in Computer Science (Proceedings of PPDP'99. Pars,France) Gopalan, Nadathur (Ed.). Springer. ISBN 3-540-66540-4. **1702** (1999).
7. M. Denecker and D. De Schreye, *Sldnfa: an abductive procedure for normal abductive programs*, Proc. International Conference and Symposium on Logic Programming (1992), 686–700.
8. Jürgen Dix, Sarit Kraus, and V.S Subrahmanian, *Temporal agent programs*, Artificial Intelligence (2001), no. 127, 87–135.
9. T.H Fung and R. Kowalski, *The iff proof procedure for abductive logic programming*, Journal of Logic Programming (1997).
10. A. C. Kakas, R. Kowalski, and F. Toni, *Handbook of logic in artificial intelligence and logic programming 5*, ch. The Role of Abduction in Logic Programming, pp. 235–324, Oxford University Press, 1998.
11. R Kowalski and F. Sadri, *From logic programming towards multi-agent systems*, Annals of Mathematics and Artificial Intelligence **25** (1999), 391–419.
12. Robert Kowalski and Marek Sergot, *A logic-based calculus of events*, New Generation Computing **4** (1986), 67–95.
13. H. Levesque, R. Reiter, Y. Lespérance, L. Fangzhen, and R. B. Scherl, *Golog: A logic programming language for dynamic domains*, (1995), (Also at http://www.cs.toronto.edu/čogrobo/).
14. J. McCarthy and P. Hayes, *Some philosophical problems from the standpoint of artificial intelligence*, Machine Intelligence **4** (1969), 463–502.