Proceedings of the Workshop at FAPR'96

Reasoning About Actions and Planning in Complex Environments

Ute C. Sigmund and Michael Thielscher

(Editors)

Technical Report AIDA-96-11

Darmstadt, June 1996

Fachgebiet Intellektik, Fachbereich Informatik
Technische Hochschule Darmstadt
Alexanderstraße 10
D-64283 Darmstadt
Germany

Table of Contents

REACTIVE PASCAL and the Event Calculus: A platform to program reactive, rational agents

Extended Abstract

Jacinto A. Dávila Quintero *
j.davila@doc.ic.ac.uk
Logic Programming Section
Department of Computing. Imperial College
180 Queen's Gate. London. SW7 2BZ. UK
http://laotzu.doc.ic.ac.uk
Phone: 0171-5948232 Fax: 0171-5891552

Abstract

This paper describes a language to program an "intelligent" (reactive. rational) agent as that described by Kowalski in [5]. The new programming language, called REACTIVE PASCAL, is part of a *specification platform* that can be based on either the Situation Calculus [9] or the Event Calculus [7]. Some mechanisms for common-sense reasoning are, therefore, directly available. The programmer/designer can complete a *background theory* describing the relevant dynamics of the universe in which the agent will operate. The *Elevator example* is borrowed from [8] to illustrate the expressiveness of the platform. The combination of REACTIVE PASCAL programs and a background theory then enables the agent to perform temporal reasoning such as that required for *planning*.

1 Introduction: From Structured to Logic Programming

A program can be seen as a scheme that can be used by an agent to generate plans to achieve some goal. Those plans should 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. In the work discussed here, a well-known programming language (STANDARD PASCAL) has been selected and extended with some useful tools to model those problem-solving and planning strategies. The new language inherits its semantics from

*This author is supported by a grant from CONICIT-University of Los Andes. Venezuela.

logic programming. In addition, the implementation of an interpreter for the language is specified by means of a normal logic program.

REACTIVE PASCAL is aimed at the same applications as GOLOG [8]. The language has been called *reactive* because its semantics embodies the principle of decomposition of goals into subgoals called **progression** [1]. Notice that this strategy fits nicely in an agent's architecture where planning can be interrupted at *any time* to be interleaved with execution and sensing. as described in ([5], [1]).

Our approach is different 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 in the reactive architecture described in [5] and [1]. Our approach also regards standard programming constructs as macros. However, here they are treated as special predicates or terms [2].

The syntax and semantics of REACTIVE PASCAL [3] are presented in tables 1 [4] and 2 [5] [6] respectively. 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 this initial formalization PASCAL syntax is "reduced" to the least number of structures required for structured programming (*";", "if.. then.. else..", "while"*). On the other hand. the syntax allows the representation of parallel actions through the compositional operators *par* [7] and + [8].

The semantics of REACTIVE PASCAL is defined in terms of the predicate *done* [9]. The definition can also function as an interpreter for the language. Informally, $done(A, T_o, T_f)$ reads "An action of type A is started at T_o and completed at T_f". Because the definition of *done* is a logic program. any semantics of normal logic programming can be used to give meaning to REACTIVE PASCAL programs.

[1] The first action to be performed is generated first.

[2] See *[DN-01]* 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.

[3] In addition to those syntactic rules, the system must provide translation from the "surface syntax", that the programmer will use to write atemporal *queries* and the actual logical notation.

[4] S_j means an instance of S of sub-type j

[5] PROLOG-like syntax is being used.

[6] E'_b (B') in [DN-07] is equal to E_b (B) except that all their existentially quantified variables have been renamed. This is crucial to preserve the semantics of **while**.

[7] Unlike those semantics of interleaving [4], this is a form of real parallelism. Actions start simultaneously although they can finish at different times. Notice. however, that this kind of parallelism requires another *cycle*, different from those presented in [5] and [1].

[8] used as well to express real parallelism. Actions are 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 [11].

[9] The definitions of *rigid. nonrigid* and other predicates are also required.

<table>
<tr><th colspan="4">Table 1 REACTIVE PASCAL: Syntax</th></tr>
<tr><td>Program</td><td>::=</td><td>Proc | Proc Program</td><td>A program</td></tr>
<tr><td>Proc</td><td>::=</td><td>proc $Func_{proc}$
begin Commands end</td><td>Procedure definition</td></tr>
<tr><td>Block</td><td>::=</td><td>begin Commands end</td><td>Block</td></tr>
<tr><td>Commands</td><td>::=</td><td>Block</td><td>Block call</td></tr>
<tr><td></td><td>|</td><td>$Func_{proc}$</td><td>Procedure call</td></tr>
<tr><td></td><td>|</td><td>$Func_{action}$</td><td>Primitive action call</td></tr>
<tr><td></td><td>|</td><td>Commands ; Commands</td><td>Sequential composition</td></tr>
<tr><td></td><td>|</td><td>Commands par Commands</td><td>Parallel composition</td></tr>
<tr><td></td><td>|</td><td>Commands + Commands</td><td>Strict parallel composition</td></tr>
<tr><td></td><td>|</td><td>if $Expr_{boolean}$ then Commands</td><td>Test</td></tr>
<tr><td></td><td>|</td><td>if $Expr_{boolean}$ then Commands
else Commands</td><td>Choice</td></tr>
<tr><td></td><td>|</td><td>while $Expr_{boolean}$ do Block</td><td>Iteration</td></tr>
<tr><td>Query</td><td>::=</td><td>...</td><td>Logical expressions</td></tr>
<tr><td>$Expr_j$</td><td>::=</td><td>$Func_j$(Func, Func, ..., Func)</td><td>Expressions (as function applications)</td></tr>
<tr><td>Func</td><td>::=</td><td>$Func_{proc}$</td><td></td></tr>
<tr><td></td><td>|</td><td>$Func_{action}$</td><td></td></tr>
<tr><td></td><td>|</td><td>$Func_{fluent}$</td><td></td></tr>
<tr><td></td><td>|</td><td>$Func_{boolean}$</td><td>Functors</td></tr>
<tr><td>$Func_{proc}$</td><td>::=</td><td>serve(Term), build(Term), ...</td><td>User-defined complex actions or procedures' names</td></tr>
<tr><td>$Func_{action}$</td><td>::=</td><td>nil</td><td>Null action</td></tr>
<tr><td></td><td>|</td><td>up | move(Term, Term) | ...</td><td>User-defined primitive actions' names</td></tr>
<tr><td>$Func_{fluent}$</td><td>::=</td><td>at(Term) | on(Term, $Func_{fluent}$) | ...</td><td>User-defined fluents</td></tr>
<tr><td>$Func_{boolean}$</td><td>::=</td><td>and($Func_{fluent}$, $Func_{boolean}$)</td><td></td></tr>
<tr><td></td><td>|</td><td>or($Func_{fluent}$, $Func_{boolean}$)</td><td></td></tr>
<tr><td></td><td>|</td><td>not($Func_{boolean}$)</td><td></td></tr>
<tr><td></td><td>|</td><td>$Func_{fluent}$</td><td>Boolean functions</td></tr>
<tr><td></td><td>|</td><td>Query</td><td>Tests on "rigid" information</td></tr>
<tr><td>Term</td><td>::=</td><td>Ind | Var</td><td>Terms can be individuals or variables</td></tr>
<tr><td>Ind</td><td>::=</td><td>...</td><td>Individuals identified by the user</td></tr>
<tr><td>Var</td><td>::=</td><td>...</td><td>Sorted Variables</td></tr>
</table>

Table 1: Syntax of REACTIVE PASCAL.

Table 2 REACTIVE PASCAL : Semantics (and implementation)		
$done(Pr,T_o,T_f)$	$\leftarrow$ **proc** Pr **begin** C **end** $\wedge\ done(C,T_o,T_f)$	[DN – 01]
$done((\ C_1 : C_2),T_o,T_f)$	$\leftarrow\ done(C_1,T_o,T_1) \wedge\ done(C_2,T_1,T_f)$	[DN – 02]
$done((\ C_1$ **par** $C_2),T_o,T_f)$	$\leftarrow\ done(C_1,T_o,T_1)\ \wedge\ done(C_2,T_o,T_f)$ $\wedge\ T_1\ \leq\ T_f$ $\vee\ done(C_1,T_o,T_f)\ \wedge\ done(C_2,T_o,T_1)$ $\wedge\ T_1\ <\ T_f$	[DN – 03]
$done((\ C_1 + C_2),T_o,T_f)$	$\leftarrow\ done(C_1,T_o,T_f)\ \wedge\ done(C_2,T_o,T_f)$	[DN – 04]
$done(($**if** E **then** $C_1),T_o,T_f)$	$\leftarrow\ holdsAt(E,T_o)\ \wedge\ done(C_1,T_o,T_f)$	[DN – 05]
$done(($**if** E **then** C_1 **else** $C_2),T_o,T_f)$	$\leftarrow\ holdsAt(E,T_o)\ \wedge\ done(C_1,T_o,T_f)$ $\vee\ \neg holdsAt(E,T_o)\ \wedge\ done(C_2,T_o,T_f)$	[DN – 06]
$done(($**while** E_b **do** $B),T_o,T_f)$	$\leftarrow\ (\neg holdsAt(E_b,T_o)\ \wedge\ T_o\ =\ T_f)$ $\vee\ \ (holdsAt(E_b,T_o)\ \wedge\ done(B,T_o,T_1)$ $\wedge\ done(($**while** E'_b **do** $B'),T_1,T_f))$	[DN – 07]
$done(($**begin** C **end**$),T_o,T_f)$	$\leftarrow\ done(C,T_o,T_f)$	[DN – 08]
$done($**nil**$,T_o,T_o)$		[DN – 09]
$holdsAt($**and**$(X,Y),T)$	$\leftarrow\ holdsAt(X,T)\ \wedge\ holdsAt(Y,T)$	[DN – 10]
$holdsAt($**or**$(X,Y),T)$	$\leftarrow\ holdsAt(X,T)\ \vee\ holdsAt(Y,T)$	[DN – 11]
$holdsAt($**not**$(X),T)$	$\leftarrow\ \neg holdsAt(X,T)$	[DN – 12]
$holdsAt(X,T)$	$\leftarrow\ nonrigid(X)\ \wedge\ holds(X,T)$	[DN – 13]
$holdsAt(Q,T)$	$\leftarrow\ rigid(Q)\ \wedge\ Q$	[DN – 14]
$nonrigid(X)$	$\leftarrow\ isfluent(X)$	[DN – 15]
$rigid(X)$	$\leftarrow\ \neg isfluent(X)$	[DN – 16]

Table 2: Semantics of REACTIVE PASCAL.

The semantics definition 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 also part of the semantics. but more important, they are the key elements of *a background theory* $\mathcal{B}$.

2 Background theories

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 inter-relationships that characterize a domain of application (including the definitions of *initiates*, *terminates* and *isfluent*).

The semantics for REACTIVE PASCAL and DD$\mathcal{B}$ 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 DI$\mathcal{B}$. The formulation presented in the following section is based on the Event Calculus [7]. Other formulations based on, for instance, the Situation Calculus [9] [10]. are equally well possible. Probably, the most important element in DI$\mathcal{B}$ is the definition of the *temporal projection predicate: holds*.

3 Background theories in the Event Calculus

The paper in which the Event Calculus (EC) was presented ([7]) offers, not only a set of inference rules, but also an ontology based on properties and the notions of initiation and termination of properties. The intuitive idea behind that formulation is: A property (in the world) holds **if** an event has happened to initiate it **and**, after the event, nothing has happened that terminates the property. We use the following axioms to formalize that:

$$holds(P,T) \leftarrow do(A,T',T_1) \wedge initiates(A,T_1,P) \wedge T_1 < T \wedge \neg clipped(T_1,P,T) \qquad \textbf{[EC1]}$$

$$clipped(T_1,P,T_2) \leftarrow do(A,T',T) \wedge terminates(A,T,P) \wedge T_1 < T \wedge T \leq T_2 \qquad \textbf{[EC2]}$$

These axioms are different from most formulations of the EC (in particular [6]) in that the well-known predicate *happens*(*Event*, *Time*) is replaced by the

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

predicate $do(Action, Starting\_Time, Finishing\_Time)$ [11].

We use a *abductive theorem prover* for interpreting REACTIVE PASCAL programs and generating plans. The execution of those plans is interleaved with their generation and also with the assimilation of inputs from the environment [5]. It is known ([2], [12], [10]) that to make an *abductive theorem prover* [13] behave as a planner, one has to define the set of abducibles, say Ab. In the present context one can make $Ab = \{do, <, \leq, =\}$. The domain-independent background theory can then be completed with the following definition (base case of *done*) [12]:

$$done(A, T_o, T_f) \leftarrow primitive(A) \wedge do(A, T_o, T_f) \quad [\mathbf{DN - EC0}]$$

By using an abductive proof-procedure (like the one by Fung [3]) with these definitions, the result of successively *unfolding* a *done* goal will be a set of *do*'s that can be regarded as the steps of the plans to achieve the goal plus a minimal set of "$\{<, \leq, =\}$" required to correctly order the *do*'s.

4 The Elevator Example

This example is borrowed from [8] where a GOLOG program is offered as a solution. Our solution is a program written in REACTIVE PASCAL.

The purpose of the program is to control an elevator. The problem of controlling elevators has been attacked by control engineers in many ways. Yet, it still seems to be an open problem because of the diversity of optimality criteria. There are several variables that can be optimized. Observe that this has to be done constantly over the working hours of the device, while the elevator keeps providing an adequate service for a highly uncertain set of *clients*.

In order to build the controller-program, [8] employs several abstractions that we preserve. The elevator is an agent that **can perform** the following primitive actions: *up(N)*: go up to floor N, *down(N)*: go down to floor N, *turnoff(N)*: wwitch off the call signal at floor N, *open*: open the door, and *close*: close the door. In addition, the agent **knows** about the following fluents: *currentfloor(C)*: the current floor is C [13], *on(N)*: the signal-call is on at floor N.

The following REACTIVE PASCAL program (ELE_PASCAL) is equivalent to the GOLOG program in [8] (pg. 10) [14]:

[11] 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.

[12] The definition of *primitive* must also be provided by the designer. It should correspond to the list of low-level, indivisible actions that the agent can perform.

[13] Levesque et al. use $current\_floor(S) = M$ to say that the current floor is M in situation S. Instead of that, we say $holds(currentfloor(M), S)$.

[14] Note that $addone(X, Y) \equiv assign(Y, X + 1)$ and $subone(X, Y) \equiv assign(Y, X - 1)$. It is assumed that there is a built-in mechanism to perform the mathematical operations.

```
proc serve( N )
  begin
    if currentfloor( C ) then
      if C=N then
        begin
          turnoff( N ) par open ; close
        end
      else
        if C<N then
           begin
             addone( C, Nx ); up( Nx ); serve( N )
           end
        else
           begin
             subone( C, Nx ); down( Nx ); serve( N )
           end
  end

proc control
  begin
    while on(N) do begin serve(N) end ;
    park
  end

proc park
  begin
    if currentfloor(C) then
      if C=0 then
        open
      else  begin down(0); open  end
  end
```

Proposition 1 *Let ELE-PLAN be { do(self,up(5),t_4,t_5), do(self,turnoff(5),t_6,t_7), do(self,open,t_6,t_8), do(self,close,t_9,t_{10}), do(self,down(4),t_{11},t_{12}), do(self,down(3), t_{12},t_{13}), do(self,open,t_{14},t_{15}), do(self,turnoff(3),t_{14},t_{16}), do(self,close,t_{17},t_{18}), done(self,park,t_{18},t_{100}) }.*

Let INEQ = $\{t_0 < \ldots < t_{100}\}$. Let ELE_T be the conjunction of ELE_H, ELE_PASCAL, ELE_DDB, EC1, EC2, INEQ and DONE [15]*, then:*

$$ELE\_T \cup ELE\_PLAN \ \vdash_{\mathbf{iff}} \ done(control, t_4, t_{100}) \qquad \text{[ELEVA]}$$

Proof: See Appendix (In the full paper).
The full paper discusses this program and compares it with the GOLOG version. The final section in the paper summarizes the contribution of the paper in the context of on going research on logic-based agents

[15] All the symbols and the rest of the example are explained in the full paper.

References

[1] Jacinto A. Dávila Quintero. A logic-based agent. Technical report, Imperial College, London, February 1996.

[2] Kave Eshghi. Abductive planning with event calculus. In *Proceedings 5th International Conference on Logic Programming*, 1988. pg. 562.

[3] Tze Ho Fung. *Abduction by deduction*. PhD thesis, Imperial College, London, January 1996.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] 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).

[6] 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).

[7] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[8] H. Levesque. R. Reiter, Y. Lespérance, L. Fangzhen, and R. B. Scherl. Golog: A logic programming language for dynamic domains. *(forthcomming)*. (Also at http://www.cs.toronto.edu/~ cogrobo/).

[9] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*. 4:463–502. 1969.

[10] 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.

[11] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. (forthcomming).

[12] 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.

[13] Francesca Toni. *Abductive Logic Programming*. PhD thesis, Imperial College, London. July 1995.