The Wumpus World in INDIGOLOG: A Preliminary Report

Sebastian Sardina and **Stavros Vassos**
University of Toronto, Toronto, ON, M5S 3G4, Canada.
{ssardina,stavros}@cs.toronto.edu

Abstract

This paper describes an implementation of the the Wumpus World [Russell and Norving, 2003] in INDIGOLOG with the objective of showing the applicability of this interleaved agent programming language for modeling agent behavior in realistic domains. We briefly go over the INDIGOLOG architecture, explain how we can reason about the Wumpus World domain, and show how to express agent behavior using high-level agent programs. Finally, we discuss initial empirical results obtained as well as challenging issues to be resolved.

1 Introduction

There has been extensive work on logical formalisms for dynamic domains. Action theories in the literature address a variety of issues ranging from the specification of actions' effects and non-effects (i.e., the so-called frame problem [McCarthy and Hayes, 1969]) and the qualification problem, to more sophisticated topics such us the ramification problem, knowledge and sensing, incomplete information, concurrent action, continuous time and nondeterministic effects, to name a few. Recently, much research into reasoning about actions has been devoted to the design and implementation of agent languages and systems for Cognitive Robotics which are often built on top of existing rich formalisms of action and change. An agent is assumed to be equipped with a formal theory of the world and a high-level program describing its behavior up to some degree. This is indeed the view taken in INDIGOLOG, the last version of the University of Toronto GOLOG-like family of agent programming languages. INDIGOLOG provides a formal account of perception, deliberation, and execution within the language of the situation calculus. INDIGOLOG is implemented in PROLOG and has been used in real robotics platforms such as the LEGO MINDSTORM and the ER1 EVOLUTION robots.

In this paper, we show how to use INDIGOLOG to model the behavior of an agent living in the Wumpus World (see [Russell and Norving, 2003, Chapter 7]), a convincing and challenging abstraction of an incompletely known dynamic environment for logically reasoning agents. To that end, we explain how to practically reason in the Wumpus World by using a special kind of situation calculus-based action theories which soundly approximate incomplete knowledge by representing the dynamics of the possible values for the fluents. In addition, we explain how the behavior of an intelligent agent can be modeled with a high-level agent program that is intended to be executed incrementally. We provide empirical results which show the feasibility of our approach for this interesting scenario.

2 INDIGOLOG: An Interleaved Agent Architecture

The agent programming language to be used for modeling an agent that acts and reasons in the Wumpus World is INDIGOLOG, the most recent situation calculus based agent language in the GOLOG family. The *situation calculus* is a second-order language specifically designed for representing dynamically changing worlds [McCarthy and Hayes, 1969; Reiter, 2001]. All changes to the world are the result of named *actions* such as $moveForward$ and $pickup(x)$. A possible world history, which is simply a sequence of actions, is represented by a first-order term called a *situation*. The constant S_0 is used to denote the *initial situation* and a distinguished binary function symbol $do(a, s)$ is used to denote the successor situation to s resulting from performing action a. The features of the world are represented with (functional) *fluents*, functions denoted with a situation term as their last argument and whose values vary from situation to situation. There is also a special predicate $Poss(a, s)$ used to state that action a is executable in situation s. In the presence of sensing actions, a special function $SR(a, s)$ is used to state the sensing result obtained from executing action a in situation s. Non-sensing actions are assumed to always return 1 as their sensing outcomes. Also, to talk about both the actions and their sensing results we use the notion of a *history*, a sequence of pairs (a, μ) where a is a primitive action and μ is the corresponding sensing outcome.

Within this language, one can specify action theories that describe how the world changes as the result of the available actions in a principled and modular way (e.g., basic action theories [Reiter, 2001]). Using the theory, the agent can query the state of the world at each possible world history by solving the so-called *projection task*: given a sequence of actions together with their corresponding sensing outcomes, and a

formula $\phi(s)$ about the world, determine whether $\phi(s)$ is true in the situation resulting from performing these actions.

On top of these action theories, logic-based programming languages can be defined, which, in addition to the primitive actions of the situation calculus, allow the definition of complex actions. GOLOG [Levesque *et al.*, 1997], the first situation calculus agent language, offers all the control structures known from conventional programming languages (e.g., sequence, iteration, conditional, etc.) plus some nondeterministic constructs. It is due to these last control structures that programs do not stand for complete solutions, but only for sketches of them whose gaps have to be filled later, usually at execution time. CONGOLOG [De Giacomo *et al.*, 2000] extends GOLOG to accommodate concurrency and interrupts in order to accommodate the specification of reactive agent behavior. A summary of the constructs available follows:

α,	primitive action
$\phi?$,	wait or test for a condition
$\delta_1; \delta_2$,	sequence
$\delta_1 \mid \delta_2$,	nondeterministic branch
$\pi\, x.\, \delta(x)$,	nondeterministic choice of argument
δ^*,	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**,	conditional
while ϕ **do** δ **endWhile**,	while loop
$\delta_1 \parallel \delta_2$,	concurrency with equal priority
$\delta_1 \rangle\!\rangle \delta_2$,	concurrency with δ_1 at a higher priority
$\delta^{\parallel}$,	concurrent iteration
$\langle\, \vec{x} : \phi(\vec{x}) \longrightarrow \delta(\vec{x})\, \rangle$,	interrupt
$p(\vec{\theta})$.	procedure call

Note the presence of nondeterministic constructs, such as $(\delta_1 \mid \delta_2)$, which nondeterministically chooses between programs δ_1 and δ_2, $\pi\, x.\, \delta(x)$, which nondeterministically picks a binding for the variable x and performs the program $\delta(x)$ for this binding of x, and δ^*, which performs δ zero or more times. To deal with concurrency two constructs are provided: $(\delta_1 \parallel \delta_2)$ expresses the concurrent execution (interpreted as interleaving) of programs δ_1 and δ_2; $(\delta_1 \rangle\!\rangle \delta_2)$ expresses the concurrent execution of δ_1 and δ_2 with δ_1 having higher priority. Finally, for an interrupt $\langle\, \vec{x} : \phi(\vec{x}) \rightarrow \delta(\vec{x})\, \rangle$, program $\delta(\vec{t})$ is executed whenever condition $\phi(\vec{t})$ holds (see [De Giacomo *et al.*, 2000] for further details.)

Finding a legal execution of high-level programs is at the core of the whole approach. Originally, GOLOG and CONGOLOG programs were intended to be executed *offline*, that is, a complete solution was obtained before committing even to the first action. In contrast, INDIGOLOG, the next extension in the GOLOG-like family of languages, provides a formal logic-based account of interleaved planning, sensing, and action [Kowalski, 1995] by executing programs *online*. Roughly speaking, an *incremental* or *online* execution of a program finds a next possible action, executes it in the real world, obtains sensing information afterwards, and repeats the cycle until the program is finished. The semantics of INDIGOLOG is specified in terms of single-steps, using two predicates $Trans$ and $Final$ [De Giacomo *et al.*, 2000]: $Final(\delta, s)$ holds if program δ may legally terminate in situation s; $Trans(\delta, s, \delta', s')$ holds if one step of program δ in situation s leads to situation s' with δ' remaining to be executed. It is important to point out also that the execution of a program strongly relies on the projection task. For example, to execute a test for condition $\phi?$ one needs the project ϕ at the current history w.r.t. the underlying theory of action.

The fact that actions are quickly executed without much deliberation and sensing information is gathered after each step makes the approach realistic for dynamic and changing environments. However, an online execution is deterministic in the sense that there is no provision for backtracking once an action has been selected. Yet there may be two actions A_1 and A_2 for which $Trans$ holds and yet only A_2 may lead ultimately to a legal successful termination. To deal with this form of non-determinism, INDIGOLOG contains a search operator Σ to allow the programmer to specify when *lookahead* should be performed: executing $\Sigma(\delta)$ means executing δ in such a way that at each step there is a sequence of further steps leading to a legal termination. Unlike a purely offline execution, however, operator Σ allows us to control the amount of lookahead to use at each step.

2.1 An Incremental Interpreter

A logic-programming implementation of INDIGOLOG has been developed to allow the incremental execution of high-level GOLOG-like programs [Sardina, 2004]. This system is fully programmed in PROLOG and has been used to control the LEGO MINDSTORM robot, the ER1 EVOLUTION robot and other soft-bot agents. The implementation provides an incremental interpreter of programs as well as all the framework to deal with the real execution of these programs in real platforms (e.g., real execution of actions, sensing outcome readings, exogenous actions, etc.)

The architecture, when applied to the Wumpus World scenario, can be divided into the following four parts: (i) the device manager software interfacing with the real world; (ii) the evaluation of test conditions; (iii) the implementation of $Trans$ and $Final$; (iv) and the main loop. The device manager for the Wumpus World is the code responsible for simulating a real-world Wumpus World environment. It provides an interface for the execution of actions (e.g., `moveFwd`, `smell`, etc.), the retrieval of sensing outcomes, and the occurrence of exogenous events (e.g., `scream`). In our case, the world configuration will be also displayed using a Java applet. We shall now briefly go over the other three parts.

The evaluation of formulas

In order to reason about the world, we need to be able to specify the domain and be able to project formulas w.r.t. evolutions of the system, that is, to evaluate the truth of formulas at arbitrary histories. We use an extension of the classical formula evaluator used for GOLOG that is able to handle some kind of incomplete knowledge. To that end, the evaluator deals with the so-called *possible values* that (functional) fluents can take at a certain history; we say that the fluent is *known* at h only when it has only one possible value at h. For a detailed description and semantics of these type of knowledge-based theories we refer to [Vassos *et al.*, 2005; Levesque, 2005].

We assume then that users provide definitions for each of the following predicates for fluent f, action a, sensing result r, formula w, and arbitrary value v:

- `fluent`(f), f is a ground fluent;

- `action`(a), a is a ground action;
- `init`(f, v), initially, v is a possible value for f;
- `poss`(a, w), it is possible to execute action a provided formula w is known to be true;
- `causes`(a, f, v, w), action a affects the value of f: when w is possibly true, v is a possible value for f;
- `settles`(a, r, f, v, w), action a with result r provides sensing information about f: when w is known to be true, v is the only possible value for f;
- `rejects`(a, r, f, v, w), action a with result r provides sensing information about f: when w is known to be true, v is not a possible value for f.

Formulas are represented in PROLOG using the obvious names for the logical operators and with all situations suppressed; histories are represented by lists of the form $o(a, r)$ where a represents an action and r a sensing result. We will not go over how formulas are recursively evaluated, but just note that the procedure is implemented using the following four predicates: (i) `kTrue`(w, h) is the main and top-level predicate and it tests if the formula w is *known to be true* in history h; (ii) `mTrue`(w, h) is used to test if w is *possibly true* at h; (iii) `subf`(w_1, w_2, h) holds when w_2 is the result of replacing each fluent in w_1 by one its *possible values* in history h; and (iv) `mval`(f, v, h) calculates the *possible* values v for fluent f in history h and is implemented as follows:

```
mval(F,V,[]) :- init(F,V).
mval(F,V,[o(A,R)|H]) :- causes(A,F,_,_),!,
  causes(A,F,V,W), mTrue(W,H).
mval(F,V,[o(A,R)|H]) :-
  settles(A,R,F,V1,W), kTrue(W,H),!, V=V1.
mval(F,V,[o(A,R)|H]) :- mval(F,V,H),
   not(rejects(A,R,F,V,W),kTrue(W,H)).
```

So for the empty history, we use the initial possible values. Otherwise, for histories whose last action is a with result r, if f is changed by a with result r, we return any value v for which the condition w is possibly true; if a with result r senses the value of f, we return the value v for which the condition is known; otherwise, we return any value v that was a possible value in the previous history h and that is not rejected by action a with result r. This provides a solution to the frame problem: if a is an action that does not affect or sense for fluent f, then the possible values for f after doing a are the same as before.

The implementation of $Trans$ and $Final$ and the main loop

Clauses for $Trans$ and $Final$ are needed for each of the program constructs. The important point to make here is that whenever a formula needs to be evaluated, `kTrue/2` is used. So, for example, these are the corresponding clauses for sequence, tests, nondeterministic choice of programs, and primitive actions:

```
final(ndet(E1,E2),H) :-
    final(E1,H) ; final(E2,H).
trans(ndet(E1,E2),H,E,H1) :-
    trans(E1,H,E,H1).
trans(ndet(E1,E2),H,E,H1) :-
    trans(E2,H,E,H1).
final([E|L],H) :- final(E,H), final(L,H).
trans([E|L],H,E1,H1) :-
    final(E,H), trans(L,H,E1,H1).
trans([E|L],H,[E1|L],H1) :-
    trans(E,H,E1,H1).
final(?(P),H) :- !, fail.
trans(?(P),H,[],H) :- kTrue(P,H).
final(A,H) :- action(A), !, fail.
trans(A,H,[],[A|H]) :-
    action(A), poss(A,P), kTrue(P,H).
```

The top part of the interpreter deals with the execution of actions in the world. It makes use of `trans/4` and `final/2` to determine the next action to perform and to end the execution.

```
indigo(E,H):-
    handle_rolling(H), !, indigo(E,[]).
indigo(E,H):-
    exog_occur(A), !, indigo(E,[A|H]).
indigo(E,H):- final(E,H), !.
indigo(E,H):-
    trans(E,H,E1,H), !, indigo(E1,H).
indigo(E,H):-
    trans(E,H,E1,[A|H]), execute(A,H,S),!,
    indigo(E1,[o(A,S)|H]).
```

In the first clause, predicate `handle_rolling/1` checks whether the current history `H` must be rolled forward (for example, if its length has exceeded some threshold). If it does, `handle_rolling/1` performs the progression of the database and the execution continues with the empty history. In the second clause, the interpreter checks whether some exogenous action has occurred. In that case, the action in question is added to the current history and execution continues. Next, the third clause ends the execution whenever the current configuration is provably terminating. The fourth clause handles the case of transition steps that involve no action. Finally, the last clause performs an action transition step; predicate `execute(A,H,S)` is the interface to the real world and is responsible of the actual execution of action `A` in history `H`. Variable `S` is bound to the corresponding sensing outcome obtained from the environment after performing the action, and the execution program continues correspondingly. In our case, predicate `execute/3` will interface with the Wumpus World device manager simulator through TCP/IP sockets.

3 Reasoning in The Wumpus World

The Wumpus World is a well-known example for reasoning and acting with incomplete knowledge (see [Russell and Norving, 2003, Chapter 7].) According to the scenario, the agent enters a dungeon in which each location may contain the Wumpus (a deadly monster), a bottomless pit, or a piece of gold. The agent moves around looking for gold and avoiding death caused by moving into the location of a pit or the Wumpus. The agent has an arrow which she can throw as an attempt to kill the Wumpus. Also, the agent can sense the world to get clues about the extent of the dungeon, as well as the location of pits, gold pieces, and the Wumpus.

Using the functionality described in Section 2.1, we construct the domain description Π_W which captures the agent's knowledge about the world. Π_W follows closely the definitions in [Russell and Norving, 2003] except for the fact that

the agent assumes a fixed predefined size for the dungeon. The world is organized as a 8×8 rectangular grid, where `g(1,1)` is the lower-left corner and `g(1,8)` is the upper-left one. Predicate `loc(L)` holds if `L` is a valid grid location and `dir(D)` holds if `D` is one of the four directions `up`, `down`, `left` and `right`. The geometry of the grid is captured by predicate `adj/2`, which holds if the arguments represent two adjacent locations, and one binary predicate for each direction such as `up(L1,L2)`, which holds if `L2` is a the location lying one step up of `L1`.

Π_W includes the following fluents, each of which captures the possible values for a dynamic element of the domain. `locA`, `dirA` and `hasArrow` represent the location of the agent, the direction that she is facing and whether she has the arrow. `noGold` is used to keep track of the number of gold pieces gathered by the agent. `isGold(L)` and `isPit(L)` capture whether there is a gold piece or a pit at location `L` and `isVisited(L)` captures whether the location has already been explored. `locW` and `aliveW` capture the location of the Wumpus and whether it is alive. Finally, `inDungeon` is used to capture whether the agent is in the dungeon. A few representative `fluent/1` clauses follow.

```
fluent(locA).  fluent(isPit(L)):- loc(L).
fluent(dirA).  fluent(isGold(L)):- loc(L).
```

Note that all fluents are functional; those that capture propositions will be defined so that only `true` and `false` may be a possible value for them. Initially, the agent is in the dungeon at location `g(1,1)` facing to the right. She possesses the arrow, but no gold pieces. There is only one possible value for the corresponding fluents and so there is complete information about the state of the agent in the empty history. On the contrary, a possible value for the location of the Wumpus is any valid grid location apart from `g(1,1)` and similarly, any location other than `g(1,1)` can possibly contain a pit or a gold piece. Also, initially the agent knows that the Wumpus is alive and that only `g(1,1)` has been explored. The `init/2` clauses for the fluents follow. We only omit the `init/2` clauses for `isGold(L)` which are identical to the ones for `isPit(L)`.

```
init(inDungeon,true).
init(locA,g(1,1)).
init(dirA,right).
init(hasArrow,true).
init(noGold,0).
init(aliveW,true).
init(locW,L):- loc(L),not L=g(1,1).
init(isPit(L),true):- loc(L),not L=g(1,1).
init(isPit(L),false):- loc(L).
init(isVisited(L),true):- L=g(1,1).
init(isVisited(L),false):- not L=g(1,1).
```

The agent can always execute the sensing actions `smell`, `senseBreeze`, and `senseGold` which give clues about the location of the Wumpus, the pits, and the gold, respectively. In order to move around, the agent can perform action `turn` which represents a change of direction by 90 degrees clockwise and `moveFwd` which represents a move, one step forward to the direction the agent is facing. This action can only be executed if it leads to a valid grid location. The agent can also perform actions `pickGold` and `shootFwd` with the intuitive meaning and preconditions. Similarly, actions `enter`, `climb` represent that the agent goes in or leaves the dungeon. Some `action/1` and `poss/2` clauses follow.

```
action(smell).
action(pickGold).
action(shootFwd).
poss(smell,true).
poss(pickGold,isGold(locA)=true).
poss(shootFwd,hasArrow=true).
```

As described in Section 2.1, the actions change the possible values of the fluents, updating in this way the agent's knowledge about the world. The fluents that represent the position of the agent, `dirA` and `locA`, are affected only by action `turn` and `moveFwd` respectively. For each of the four directions, Π_W includes an appropriate `causes/4` clause of the following form.

```
causes(turn,dirA,Y,and(dirA=up,Y=right))).
causes(moveFwd,locA,Y,
    and(dirA=up,up(locA,Y))).
```

Since there is complete information about `locA` and `dirA` initially, these clauses make sure that each fluent has exactly one possible value in all histories. Similarly, `inDungeon` is affected only by actions `enter` and `climb`, `hasArrow` by action `shootFwd`, `noGold` by action `pickGold`, and `isVisited(L)` by `moveFwd`. For these fluents too, there is complete information in all histories.

The rest of the fluents can capture incomplete information which may shrink whenever a sensing action is performed. Fluent `locW` is sensed by a `smell` action: if there is no stench (i.e. the sensing result is 0) then each of the robot's adjacent locations is not a possible value for `locW`, otherwise the opposite holds. Fluent `isGold(L)` is sensed by a `senseGold` action which settles the value of the fluent depending on the sensing result. Π_W includes the following clauses.

```
rejects(smell,0,locW,Y,adj(locA,Y)).
rejects(smell,1,locW,Y,neg(adj(locA,Y))).
settles(senseGold,1,isGold(L),true,locA=L).
settles(senseGold,0,isGold(L),false,locA=L).
```

Fluent `isPit(L)` is sensed by a `senseBreeze` action: if no breeze is sensed (i.e. the sensing result is 0), then `isPit(L)` is settled to value `false` for the locations `L` which are adjacent to the agent. Otherwise, if all the adjacent locations but one are known not to contain a pit, then the unknown one is settled to contain a pit. Note that this last rule is indeed limited for reasoning about pit locations in the sense that it is incomplete whenever there is uncertainty in more than one adjacent location. Finally, fluent `aliveW` is affected by the `shootFwd` action: if the shot was in the right direction, then the Wumpus dies.

We conclude this section with an example. In the empty history, the agent is located at `g(1,1)` and senses the stench of the Wumpus. The disjunctive knowledge about fluent `locW` then shrinks, so that the only possible values are the locations which are adjacent to the agent.

```
?- mTrue(locW=g(1,2),[o(smell,1)]),
   mTrue(locW=g(2,1),[o(smell,1)]).
Yes
```

```
?- loc(L), not adj(g(1,1),L),
   mTrue(locW=L,[o(smell,1)]).
No
```

Sensing can further limit the possible values to be exactly one. For example,

```
?- kTrue(locW=g(3,1),[o(smell,0),
   o(moveFwd,1), o(turn,1), o(moveFwd,1),
   o(turn,1), o(turn,1), o(smell,1),
   o(moveFwd,1), o(smell,0)]).
Yes
```

4 Experimental Results in the Wumpus World

In order to test the feasibility of our approach, we defined an INDIGOLOG high-level controller, which is intended to run w.r.t. the domain description given in the previous section, and experimented on several random settings of the scenario. The controller we used is quite simple; it is implemented using three concurrent interrupts at three different levels of priority, which can be described as follows:

1. If the Wumpus is known to be alive at a location l which is aligned with the agent's location, then execute procedure $shoot$ with the direction at which the Wumpus is known to be. Procedure $shoot(d)$ is in charge of aiming and shooting the arrow at direction d; it is defined using a search block as follows:

   proc $shoot(d)$
   $\Sigma(\texttt{turn}^*; (\texttt{dirA} = d)?; \texttt{shootFwd})$
   endProc
2. If there is a gold piece at the current location, then pick it up.
3. If the agent is in the dungeon, then she senses the world and proceeds to explore an unvisited location, provided it is safe and necessary to do so. Otherwise, she returns to location `g(1,1)` and climbs out of the dungeon. The exploration is done using an iterative deepening procedure, $explore$. Procedure $goto(l)$ goes to location l traversing only visited locations.

So, here is the INDIGOLOG controller in question that is to be executed online:

proc $mainControl$
$\langle d, l : \texttt{locW} = l \wedge \texttt{aliveW} = \texttt{true} \wedge$
$\quad \texttt{aligned}(\texttt{locA}, dir, \texttt{locW}) \longrightarrow shoot(d)\rangle \rangle\!\rangle$
$\langle \texttt{isGold(locA)} = \texttt{true} \longrightarrow \texttt{pickGold}\rangle \rangle\!\rangle$
$\langle \texttt{inDungeon} = \texttt{true} \longrightarrow$
$\quad \{\texttt{smell}; \texttt{senseBreeze}; \texttt{senseGold}$
$\quad \{?(\texttt{noGold} = 0); explore\} \mid \{goto(\texttt{g(1,1)}); \texttt{climb}\}\}\rangle$
endProc

We performed a series of experiments where the world is an 8×8 grid. A setting (n, p) represents a random configuration with n gold pieces and a probability p of a location containing a pit. For each of these settings, we tested our controller in 300 random scenarios. The experiments verify that the agent always exits from the dungeon alive.

Figure 1 summarizes our evaluation metrics for some representative settings. The column "Gold" specifies the number of the scenarios where the agent managed to get gold. The column "Imp." specifies the number of scenarios in which a pit or the Wumpus was located just next to the initial location of the agent, and hence the problem was unsolvable for the agent.[1] The rest of the columns show the average of the reward, the number of moves, and the time it took the agent to exit the dungeon. Time is the total running time of the INDIGOLOG controller in seconds and rounded; this includes the deliberation time, as well as the time needed for INDIGOLOG to execute the simulated actions. The reward is calculated as follows: move actions cost -1, a shoot action costs -10, getting the gold is +1000, and dying is -1000. Each number in parenthesis is the corresponding average calculated only over the scenarios which are not "impossible" in the aforementioned sense.

CONF	GOLD	IMP	REWARD	MOVES	TIME
(1, .10)	168	68	584 (737)	111 (140)	21 (26)
(1, .15)	93	113	342 (514)	74 (112)	15 (23)
(1, .20)	59	138	235 (390)	44 (76)	9 (16)
(1, .30)	24	188	122 (218)	24 (54)	6 (14)
(1, .40)	14	229	93 (210)	15 (38)	3 (9)
(2, .10)	162	77	574 (746)	69 (89)	17 (23)
(2, .15)	131	90	472 (630)	57 (78)	14 (19)
(2, .20)	85	138	320 (527)	42 (73)	10 (18)
(2, .30)	55	180	226 (428)	22 (42)	5 (11)
(2, .40)	24	221	125 (278)	14 (38)	3 (10)
(4, .10)	185	86	654 (878)	42 (55)	11 (15)
(4, .15)	149	103	534 (770)	38 (55)	10 (15)
(4, .20)	114	139	422 (719)	29 (48)	7 (12)
(4, .30)	69	188	273 (571)	19 (38)	4 (10)
(4, .40)	49	227	208 (596)	13 (33)	3 (9)

Figure 1: Experimental results for controller $mainControl$.

CONF	GOLD	IMP	REWARD	MOVES	TIME
(1, .10)	167	68	574 (724)	163 (207)	10 (13)
(1, .15)	92	113	332 (500)	105 (160)	4 (6)
(1, .20)	57	138	220 (363)	70 (121)	1 (2)
(1, .30)	24	188	117 (209)	36 (80)	0 (1)
(1, .40)	12	229	82 (172)	21 (56)	0 (0)
(2, .10)	162	77	566 (736)	104 (137)	5 (7)
(2, .15)	131	90	465 (615)	86 (119)	3 (4)
(2, .20)	85	138	314 (519)	58 (100)	1 (1)
(2, .30)	54	180	220 (415)	29 (56)	0 (1)
(2, .40)	24	221	122 (272)	21 (57)	0 (0)
(4, .10)	185	86	649 (877)	62 (83)	2 (3)
(4, .15)	149	103	531 (765)	51 (73)	1 (1)
(4, .20)	113	139	416 (710)	39 (63)	0 (1)
(4, .30)	69	188	269 (563)	28 (58)	0 (0)
(4, .40)	48	227	203 (581)	17 (45)	0 (0)

Figure 2: Experimental results using FLUX.

As the probability p for the pits increases, the scenarios which are not impossible become rare and exploring becomes difficult because there are not many locations which the agent can conclude that they are safe to go. For scenarios with 1 gold piece, which are not impossible, the effectiveness of the agent in getting the gold ranges from 73.9% for $p = 0.10$ down to 23.1% for $p = .40$. As expected, as p increases, the average reward and the average number of moves decrease.

[1] Our agent is risk-averse: when no safe exploration is possible, the agent exits the dungeon.

Note that the average running time drops as p increases, but this is merely because in many scenarios the agent has to give up exploring very early. A safer metric for running time is the average time calculated over the scenarios which are not impossible; this also decreases because of the same reason, but it is more reasonable as a metric for the time the agent needs to fully explore the world.

Finally, as the number of gold pieces increase in the grid, the effectiveness of the agent increases also, both in respect to the average reward and in respect to the average time. This is because when there are more than one gold pieces in the grid, the chances that there exists a risk-averse path to one of them increase. As one can observe, this also mitigates the effect of having pits with high probability.

5 Discussion

In this article, we have shown how to model an intelligent agent acting in the Wumpus World by using the INDIGOLOG agent programming language.

The only available Cognitive Robotics implementation of the Wumpus World that we are aware of is that of FLUX [Thielscher, 2004]. FLUX is a constraint logic programming method in which agents could be programmed relative to fluent calculus action theories. The fluent calculus [Thielscher, 2000] extends the situation calculus by making explicit the notion of *states*—a snapshot of the environment characterized by the set of fluents that are true in it and some extra constraints representing incomplete knowledge. By appealing to states, FLUX uses *progression*, rather than regression, as the computational mechanism for answering queries about the world and the agent's knowledge. By using progression and appealing to constraint programming, FLUX offers a computationally attractive framework for implementation of agents with incomplete information.

The results of running the available FLUX Wumpus implementation on exactly the same scenarios used for INDIGOLOG are shown in Figure 2. As one can observe, the empirical results do not seem to differ much from the INDIGOLOG ones and many differences may, in fact, only reflect the different strategies used in the agent controllers rather than the actual programming framework (e.g., FLUX seems to perform more actions). Nonetheless, we think there are two major issues worth mentioning. First, the FLUX implementation strongly relies on constraint solvers. This makes the agent reasoning substantially faster than when plain Prolog technology is used, as with our current INDIGOLOG implementation. We believe that this could be even more explicit when larger grids are used and, hence, it suggests the convenience of investigating how constraint programming can be incorporated in INDIGOLOG too. Notice that, like ours, the FLUX implementation also assumes a fixed grid size. Second, it is difficult, however, to compare the formal accounts of execution in FLUX and INDIGOLOG. This is mainly because the FLUX controller is no more than a constraint logic program; FLUX does not come with a well-defined notion of what an agent program execution is (e.g., in terms of single-step semantics) and there is no *formal* interleaved account of sensing, deliberation, and execution. The recent work [Schiffel and Thielscher, 2005] is a first attempt towards that. Finally and unlike FLUX, the INDIGOLOG framework is able to accommodate exogenous actions such as the scream of the Wumpus when it dies.

Many issues remain open for further investigation. The most important one is to study how our implementation scales up with larger grids and, more importantly, how to model an agent who does not know the grid size initially but can obtain information about its limits by bumping into the walls. We would also like to develop other controllers and compare them empirically. In particular, we think that in order to handle the intrinsic incompleteness in the reasoning, the agent may try to explore the grid more than once if necessary.

Acknowledgments

We thank Hector Levesque for his valuable help.

References

[De Giacomo *et al.*, 2000] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[Kowalski, 1995] R. A. Kowalski. Using meta-logic to reconcile reactive with rational agents. In K. R. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*, pages 227–242. MIT Press, 1995.

[Levesque *et al.*, 1997] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[Levesque, 2005] Hector Levesque. Planning with loops. In *Proc. of ICJAI'05*, 2005. To appear.

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

[Reiter, 2001] Raymond Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[Russell and Norving, 2003] Stuart Russell and Peter Norving. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[Sardina, 2004] Sebastian Sardina. *IndiGolog: An Integrated Agent Arquitecture: Programmer and User Manual*. University of Toronto, 2004.

[Schiffel and Thielscher, 2005] S. Schiffel and M. Thielscher. Interpreting golog programs in flux. In *7th International Symposium On Logical Formalizations of Commonsense Reasoning, The*, 2005.

[Thielscher, 2000] Michael Thielscher. The fluent calculus. Technical Report CL-2000-01, Computational Logic Group, AI Institute, Dept. of Computer Science, Dresden University of Technology, April 2000.

[Thielscher, 2004] Michael Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 2004.

[Vassos *et al.*, 2005] Stavros Vassos, Sebastian Sardina, and Hector Levesque. A feasible approach to disjunctive knowledge in the situation calculus. In preparation, 2005.