IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents

Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina

Abstract IndiGolog is a programming language for autonomous agents that sense their environment and do planning as they operate. Instead of classical planning, it supports *high-level program execution*. The programmer provides a high-level nondeterministic program involving domain-specific actions and tests to perform the agent's tasks. The IndiGolog interpreter then reasons about the preconditions and effects of the actions in the program to find a legal terminating execution. To support this, the programmer provides a declarative specification of the domain (i.e., primitive actions, preconditions and effects, what is known about the initial state) in the situation calculus. The programmer can control the amount of nondeterminism in the program and how much of it is searched over. The language is rich and supports concurrent programming. Programs are executed online together with sensing the environment and monitoring for events, thus supporting the development of reactive agents. We discuss the language, its implementation, and applications that have been realized with it.

Giuseppe De Giacomo
Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Rome, Italy e-mail: `degiacomo@dis.uniroma1.it`

Yves Lespérance
Department of Computer Science and Engineering, York University, Toronto, Canada e-mail: `lesperan@cse.yorku.ca`

Hector J. Levesque
Department of Computer Science, University of Toronto, Toronto, Canada e-mail: `hector@cs.toronto.edu`

Sebastian Sardina
School of Computer Science and Information Technology, RMIT University, Melbourne, Australia e-mail: `sebastian.sardina@rmit.edu.au`

1 Motivation

Designing autonomous agents that do the right thing in changing and incompletely known environments is challenging. The agent needs to adapt to changing environment conditions and user objectives. Architectures based on classical planning can provide flexibility and adaptability, but they often end up being too demanding computationally. Our approach of *high-level program execution* [61] aims for a middle ground between classical planning and normal programming. The idea, roughly, is that instead of searching for a sequence of actions that would take the agent from an initial state to some goal state, the task is to find a sequence of actions that constitutes a legal execution of some sketchy high-level non-deterministic program involving domain specific actions and tests. As in planning, to find a sequence that constitutes a legal execution of a high-level program, it is necessary to reason about the preconditions and effects of the actions within the body of the program. However, if the program happens to be almost deterministic, very little searching is required; as more and more non-determinism is included, the search task begins to resemble traditional planning. Thus, in formulating a high-level program, the programmer gets to control the search effort required.

The high-level program execution approach to agent programming was concretely realized in the Golog programming language [62], a procedural language defined on top of the situation calculus [71, 82], a predicate logic framework for reasoning about action. Golog (the name stands for "alGOl in LOGic") provides a full set of procedural constructs including conditionals, loops, recursive procedures, as well as several nondeterministic choice constructs. The interpreter for the language uses a situation calculus action theory representing the agent's beliefs about the state of the environment and the preconditions and effects of the actions to reason and find a provably correct execution of the program.

An extension of Golog called ConGolog (Concurrent Golog) [24] was later developed to provide concurrent programming facilities. Then, more recently, in IndiGolog (incremental deterministic Golog) [26, 87, 28], the framework was generalized to allow the programmer to control planning/lookahead and support online execution, sensing the environment, and execution monitoring.

In addition to these, there have been other proposals of languages based on the high-level program execution approach. One is Thielscher's FLUX language [99], which uses the fluent calculus as its formal foundation. As well, decision theoretic versions of the approach have been proposed yielding languages such as DTGolog [14, 97, 34].

In this chapter, we will focus our presentation on IndiGolog, briefly mentioning how it differs from Golog and ConGolog as we go along. The high level program execution approach is related to work on planning with domain specific control information, such as hierarchical task network (HTN) planning [33], and planning with temporal logic control specifications [3].

The Golog family of high level agent programming languages can be contrasted with the more mainstream BDI agent programming languages/architectures, such as PRS [39] and its various successors, such as AgentSpeak [78], Jason [11], Jack [17], and JAM [45], as well as the closely related 3APL [44]. These were developed as a way of enabling *abstract plans* written by programmers to be combined and used in real-time, in a way that was both flexible and robust. These BDI agent programming languages were conceived as a simplified and operationalized version of the BDI (Belief, Desire, Intention) model of agency, which is rooted in philosophical work such as Bratman's [15] theory of practical reasoning and Dennet's theory of intentional systems [31]. In the BDI paradigm, agents are viewed, as well as built, as entities whose (rational) actions are a consequence of their mental attitudes, beliefs, desires, obligations, intentions, etc. Theoretical work on the BDI model has focused on the formal specification of the complex logical relationships among these mental attitudes (e.g., [22, 79]). But more practical work in the area has sought to develop BDI agent programming languages that incorporate a simplified BDI semantics basis that has a computational interpretation.

An important feature of BDI-style programming languages and platforms is their interleaved account of sensing, deliberation, and execution [76]. In BDI systems, *abstract plans* written by programmers are combined and executed in real-time. By executing as they reason, BDI agents reduce the likelihood that decisions will be made on the basis of outdated beliefs and remain responsive to the environment by making adjustments in the steps chosen as they proceed. Because of this, BDI agent programming languages are well suited to implementing systems that need to operate more or less in real time (e.g., air traffic control and unmanned aerial vehicles (UAVs), space shuttle monitoring, search and rescue co-ordination, internet electronic business, and automated manufacturing [66, 7, 32, 9]). Unlike in classical planning-based architectures, *execution* happens at each step, and there is no lookahead to check that the selected plan can be successfully expanded and executed. The assumption is that the careful crafting of plans' preconditions to ensure the selection of appropriate plans at execution time, together with a built-in mechanism for trying alternative options, will usually ensure that a successful execution is found, even in the context of a changing environment. The approach works well if good plans can be specified for all objectives that the agent may acquire and all contingencies that may arise. However, there are often too many possible objectives and contingencies for this to be practical. Trying alternative options may not work in an environment where choices cannot be "undone." Thus supplying some form of lookahead planning in an agent programming language remains valuable provided it can be effectively controlled.

Various proposals have been made to incorporate planning (at execution time) in BDI agent programming languages. [89, 90] have proposed the CANPlan and CanPlan2 languages, that incorporate an HTN planning mechanism [33] into a classical BDI agent programming language. Earlier less formal work on this topic is reviewed in [89]. We will come back to the relationship between our high level agent programming approach and other work on agent programming languages in the final section of the chapter.

The rest of the chapter is organized as follows. In the next section, we present the syntax and semantics of IndiGolog and discuss the basis of the whole approach. In Section 3, we discuss in details our platform for high-level program execution supporting IndiGolog. In Section 4, we briefly survey some of the applications that have been developed using it. After that, we conclude by discussing the distinguishing features of our approach and issues for future work.

2 Language

2.1 Specifications and Syntactical Aspects

The Situation Calculus and Basic Action Theories

Our approach to agent programming relies on the agent being able to reason about its world and how it can change, whether for planning/lookahead, for updating its knowledge after executing an action or observing an exogenous action/event, for monitoring whether its actions are having the expected effects, etc. More specifically, we assume that the agent has a theory of action for the domain in which it operates, a theory which is specified in the situation calculus [71], a popular predicate logic formalism for representing dynamic domains and reasoning about action.

We will not go over the situation calculus in detail. We merely note the following components. There is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred. There is a distinguished binary function symbol do, where $do(a, s)$ denotes the successor situation to s resulting from performing the action a. For example, in a Blocks World, the situation term $do(put(A, B), do(put(B, C), S_0))$, could denote the situation where the agent has done the actions of first putting block B on block C and then putting block A on block B, after starting in the initial situation S_0. Relations (resp. functions) whose values vary from situation to situation, are called *fluents*, and are denoted by predicate (resp. function) symbols taking a situation term as their last argument. Thus, for example, we might have that block B was initially on the table, i.e. $OnTable(B, S_0)$, and after the agent put it on C, it no longer was, i.e. $\neg OnTable(B, do(put(B, C), S_0))$. There is also a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

Within this language, we can formulate action theories which describe how the world changes as the result of the available actions in the domain. Here, we use *basic action theories* [82], which include the following types of axioms:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms, one for each relational fluent F (resp. functional fluent f), which characterize the conditions under which $F(\mathbf{x}, do(a, s))$ holds (resp. $f(\mathbf{x}, do(a, s)) = v$) in terms of what holds in situation s; these axioms may be compiled from effects axioms, but provide a solution to the frame problem [81].
- Unique names axioms for the primitive actions.
- A set of foundational, domain independent axioms for situations Σ as in [82].

Various ways of modeling sensing in the situation calculus have been proposed. One is to introduce a special fluent $SF(a, s)$ (for *sensed fluent value*) and axioms describing how the truth value of SF becomes correlated with those aspects of a situation which are being sensed by action a [58]. For example, the axiom

$$SF(senseDoor(d), s) \equiv Open(d, s)$$

states that the action $senseDoor(d)$ tells the agent whether the door is open in situation s. For actions with no useful sensing information, one writes $SF(a, s) \equiv True$. In general, of course, sensing results are not binary. For example, reading the temperature could mean returning an integer or real number. See [93] on how these can be represented.

To describe an execution of a sequence of actions together with their sensing results, one can use the notion of a *history*, i.e., a sequence of pairs (a, μ) where a is a primitive action and μ is 1 or 0, a sensing result. Intuitively, the history $\sigma = (a_1, \mu_1) \cdot \ldots \cdot (a_n, \mu_n)$ is one where actions $a_1, \ldots, a_n$ happen starting in some initial situation, and each action a_i returns sensing result μ_i. We can use $end[\sigma]$ to denote the situation term corresponding to the history σ, and $Sensed[\sigma]$ to denote the formula of the situation calculus stating all sensing results of the history σ. Formally,

$end[\epsilon] = S_0$, where ϵ is the empty history; and
$end[\sigma \cdot (a, \mu)] = do(a, end[\sigma])$.

$Sensed[\epsilon] = True$;
$Sensed[\sigma \cdot (a, 1)] = Sensed[\sigma] \wedge SF(a, end[\sigma])$;
$Sensed[\sigma \cdot (a, 0)] = Sensed[\sigma] \wedge \neg SF(a, end[\sigma])$.

We illustrate how a domain is specified by giving a partial specification of the Wumpus World domain [92]:

$$
\begin{aligned}
&LocAgent(S_0) = \langle 1,1\rangle,\\
&HasArrow(S_0),\\
&DirAgent(S_0) = right,\\
&Poss(pickGold, s) \equiv IsGold(LocAgent(s),s),\\
&DirAgent(do(a,s)) = y \ \equiv\\
&\qquad\qquad (a = turnRight \wedge DirAgent(s) = down \wedge y = left) \vee\\
&\qquad\qquad \ldots \vee (a \neq turnRight \wedge a \neq turnLeft \wedge DirAgent(s) = y).
\end{aligned}
$$

Thus, the agent is initially on the $\langle 1,1\rangle$ square, facing in the *right* direction, and it has some arrows to shoot at the Wumpus. It is possible for the agent to perform the *pickGold* action in a situation s if there is a gold coin where the agent is located in s. The direction of the agent is y in the situation that results from action a being performed in situation s if and only if the action was to turn in the right direction (i.e. clockwise) and the agent was facing *down* in s and the new direction y is *left*, or any of several other cases of the agent doing a turning action (we leave out the details), or the agent's direction was already y in s and the action a is neither turning right nor turning left.

The IndiGolog Programming Constructs

Next we turn to programs. IndiGolog provides the following rich set of programming constructs (most of which are inherited from Golog [62] and ConGolog [24]):

a,	primitive action
$\phi?$,	test/wait for a condition
$\delta_1; \delta_2$,	sequence
$\delta_1 \mid \delta_2$,	nondeterministic branch
$\pi\, x.\, \delta$,	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
$\langle \phi \to \delta \rangle$,	interrupt
proc $P(\mathbf{x})$ δ **endProc**,	procedure definition
$P(\boldsymbol{\theta})$,	procedure call
$\Sigma(\delta)$,	search operator

In the first line, a stands for a situation calculus action term where the special situation constant *now* may be used to refer to the current situation (i.e. that where a is to be executed). Similarly, in the line below, ϕ stands for a situation calculus formula where *now* may be used to refer to the current situation, for example $\mathit{OnTable}(\mathit{block}, \mathit{now})$. We use $a[s]$ ($\phi[s]$) for the action (formula) obtained by substituting the situation variable s for all occurrences of now in functional fluents appearing in a (functional and predicate fluents appearing in ϕ). Moreover when no confusion can arise, we often leave out the now argument from fluents altogether; e.g. write $\mathit{OnTable}(\mathit{block})$ instead of $\mathit{OnTable}(\mathit{block}, \mathit{now})$. In such cases, the situation suppressed version of the action or formula should be understood as an abbreviation for the version with now.

Among the constructs listed, we notice the presence of nondeterministic constructs. These include $(\delta_1 \mid \delta_2)$, which nondeterministically chooses between programs δ_1 and δ_2, $\pi\, x.\, \delta$, which nondeterministically picks a binding for the variable x and performs the program δ for this binding of x, and δ^*, which performs δ zero or more times. $\pi\, x_1, \ldots, x_n.\, \delta$ is an abbreviation for $\pi\, x_1. \ldots. \pi\, x_n\, \delta$.

Test actions $\phi?$ can be used to control which branches may be executed, e.g., $[(\phi?; \delta_1) \mid (\neg\phi?; \delta_2)]$ will perform δ_1 when ϕ is true and δ_2 when ϕ is false (we use $[\ldots]$ and $(\ldots)$ interchangeably to disambiguate structure in programs). A test can also be used to constrain the value of a nondeterministically bound variable, e.g., $\pi\, x.\, [\phi(x)?; \delta(x)]$ will perform $\delta(x)$ with x bound to a value that satisfies $\phi(x)$ (or fail if no such value exists). Finally, as we discuss below, tests can also be used to synchronize concurrent processes.

The constructs **if** ϕ **then** δ_1 **else** δ_2 **endIf** and **while** ϕ **do** δ **endWhile** are the synchronized versions of the usual if-then-else and while-loop. They are synchronized in the sense that testing the condition ϕ does not involve a transition per se: the evaluation of the condition and the first action of the branch chosen are executed as an atomic unit. So these constructs behave in a similar way to the test-and-set atomic instructions used to build semaphores in concurrent programming.[1]

We also have constructs for concurrent programming. In particular $(\delta_1 \parallel \delta_2)$ expresses the concurrent execution (interpreted as interleaving) of the programs/processes δ_1 and δ_2. Observe that a process may become blocked when it reaches a primitive action whose preconditions are false or a test/wait action $\phi?$ whose condition ϕ is false. Then, execution of the program may continue provided that another process executes next. When the condition causing the blocking becomes true, the no longer blocked process can resume execution.

Another concurrent programming construct is $(\delta_1 \rangle\!\rangle\, \delta_2)$, where δ_1 has higher priority than δ_2, and δ_2 may only execute when δ_1 is done or blocked. $\delta^{\parallel}$ is like nondeterministic iteration δ^*, but the instances of δ are executed concurrently rather than in sequence.

[1] In [62], non-synchronized versions of if-then-elses and while-loops are introduced by defining: **if** ϕ **then** δ_1 **else** δ_2 **endIf** $\stackrel{\text{def}}{=} [(\phi?; \delta_1) \mid (\neg\phi?; \delta_2)]$ and **while** ϕ **do** δ **endWhile** $\stackrel{\text{def}}{=} [(\phi?; \delta)^*; \neg\phi?]$. The synchronized versions of these constructs introduced here behave essentially as the non-synchronized ones in absence of concurrency. However the difference is significant when concurrency is allowed.

Finally, one may include interrupts in a concurrent program to immediately "react" to a condition becoming true. An interrupt $\langle \phi \rightarrow \delta \rangle$ has a trigger condition ϕ, and a body δ. If the interrupt gets control from higher priority processes and the condition ϕ is true, the interrupt triggers and the body is executed, suspending any lower priority processes that may have been executing. Once the interrupt body completes execution, the suspended lower priority processes may resume. The interrupt may also trigger again (when its condition becomes true). $\langle \mathbf{x} : \phi \rightarrow \delta \rangle$ is an abbreviation for $\langle \exists \mathbf{x}.\phi \rightarrow \pi \mathbf{x}.[\phi?; \delta] \rangle$. The language also allows for recursive procedures, and other convenient constructs can easily be defined, usually as abbreviations.

Finally, the *search operator* $\Sigma(\delta)$ is used to specify that lookahead should be performed over the (nondeterministic) program δ to ensure that nondeterministic choices are resolved in a way that guarantees its successful completion. When a program is not in a search block, nondeterministic choices are resolved externally from the program executor, and hence, to the executor, look like they are made in an arbitrary way. The search operator can thus be used by the programmer to control the scope of lookahead search (this is a new feature in IndiGolog [26, 87]; in Golog and ConGolog lookahead search over the whole program is automatically performed). We discuss the semantics of the search operator in the next section.

Some Examples

We illustrate how one specifies an agent's behavior in IndiGolog with some examples from the Wumpus World application [92]. If the Wumpus is known to be alive at a location l which is aligned with the agent's location, then the agent executes procedure shoot(d) with the direction d at which the Wumpus is known to be. The procedure is in charge of aiming and shooting the arrow at direction d; it is defined using a search block as follows:

```
    **proc** shoot($d$)
      $\Sigma[(\mathit{turnRight}^* \mid \mathit{turnLeft}^*);\ (\mathit{DirAgent} = d)?;\ \mathit{shootFwd}]$
    **endProc**
```

The agent's main control procedure, which is to be executed online is as follows:

```
proc mainControl
    $\langle d, l :\ \mathit{LocWumpus} = l \wedge \mathit{AliveWumpus} = \texttt{true} \wedge$
            $\mathit{Aligned}(\mathit{LocAgent}, d, \mathit{LocWumpus}) \longrightarrow \mathrm{shoot}(d) \rangle$
    $\rangle\!\rangle$
    $\langle \mathit{IsGold(LocAgent)} = \texttt{true} \longrightarrow \mathit{pickGold} \rangle$
    $\rangle\!\rangle$
    $\langle \mathit{InDungeon} = \texttt{true} \longrightarrow$
        $\mathit{smell};\ \mathit{senseBreeze};\ \mathit{senseGold};$
        $[(\neg \mathit{HoldingGold}?;\ \mathrm{explore}) \mid\ (\mathit{HoldingGold}?; \mathit{goto}(\langle 1,1 \rangle);\ \mathit{climb})] \rangle$
endProc
```

Here, we use a set of prioritized interrupts to ensure that the agent reacts immediately to threats/opportunities: if the agent comes to know that the Wumpus is in shooting range (highest priority interrupt), it interrupts whatever it was doing and immediately executes the procedure "shoot" with the appropriate direction argument; otherwise, if it comes to know that there is gold at the current location (medium priority interrupt), it immediately picks it up; otherwise, finally, if it is in the dungeon (lowest priority interrupt), it senses its surroundings and then either executes the "explore" procedure when it is not yet carrying gold or exits the dungeon otherwise. The program terminates when the conditions of all the interrupts become false, i.e., when the agent is no longer in the dungeon.

To further illustrate how the search operator can be used, consider the following example (adapted from one in [57]) of an iterative deepening search procedure to find a robot delivery schedule/route that serves all clients and minimizes the distance traveled by the robot; one calls the procedure using Σ(minimizeDistance(0)):

```
proc minimizeDistance($dist$)
    serveAllClientsWithin($dist$)        % try to serve all clients in at most $dist$
     | minimizeDistance($dist + 1$)      % otherwise increment $dist$
endProc

proc serveAllClientsWithin($dist$)
    $((\neg\exists c)\,ClientToServe(c))?$          % when all clients served, exit
     |                                              % otherwise pick a client
     $\pi\, c, d.[\ (ClientToServe(c) \wedge DistanceTo(c) = d \wedge d \leq dist)?;$
            goTo($c$); $serve(c)$;                      % serve selected client
            serveAllClientsWithin($dist - d$)]          % serve remaining clients
endProc
```

Note that in "minimizeDistance," we rely on the fact that the IndiGolog implementation tries nondeterministic branches left-to-right, in Prolog fashion. It is possible to define a "try δ_1 otherwise δ_2" construct that eliminates the need for this assumption.

As a final example of the use of the search operator, consider the following procedures, which implement a generic iterative deepening planner (adapted from [82]):

```
proc IDPlan($maxl$) % main iterative deepening planning procedure
    IDPlan2($0, maxl$)
endProc

proc IDPlan2($l, maxl$)
    BDFPlan($l$)                 % try to find a plan of length $l$
     | [$(l < maxl)?$; IDPlan2($l + 1, maxl$)] % else increment $l$ up to $maxl$
endProc
```

```
**proc**BDFPlan($l$) % a bounded depth first planning procedure
    ($Goal$)? |
    [$(l > 0)$?; $\pi\, a.(Acceptable(a))$?; $a$; BDFPlan($l-1$)]
**endProc**
```

One calls the planning procedure using $\Sigma(\text{IDPlan}(N))$ where N is a plan length bound; *Goal* is a defined predicate specifying the goal and *Acceptable* is another defined predicate that can be used to filter what actions are considered in a given situation.

2.2 Semantics and Verification

Reasoning about Action: Projection via Regression and Progression

Our "high level program execution" framework requires reasoning about action. The executor must reason to check that its actions are possible and to determine whether the tests in the high-level program hold. This reasoning is required whether the agent is actually executing the program online or performing lookahead/planning to find a successful execution offline. So let's begin by discussing reasoning about action.

There are two well known reasoning tasks that our executor must perform. The main one is called the (temporal) *projection task*: determining whether or not some condition will hold after a sequence of actions has been performed starting in some initial state. The second one is called the *legality task*: determining whether a sequence of actions *can* be performed starting in some initial state. Assuming we have access to the preconditions of actions, legality reduces to projection, since we can determine legality by verifying that the preconditions of each action in the sequence are satisfied in the state just before the action is executed. Projection is a very basic task since it is necessary for a number of other larger tasks, including planning and high-level program execution, as we will see later in the chapter.

We can define projection in the situation calculus as follows: given an action theory $\mathcal{D}$, a sequence of ground action terms, $\mathbf{a} = [a_1, \ldots, a_n]$, and a formula $\phi[s]$ that is uniform in s (i.e. roughly where the only situation term that appears is s), the task is to determine whether or not

$$\mathcal{D} \models \phi[do(\mathbf{a}, S_0)].$$

Reiter [81] has shown that the projection problem can be solved by *regression*: when $\mathcal{D}$ is an action theory (as specified earlier), there is a regression operator $\mathcal{R}$, such that for any ϕ uniform in s,

$$\mathcal{D} \models \phi[do(\mathbf{a}, S_0)] \quad \text{iff} \quad \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models \phi'[S_0],$$

where $\mathcal{D}_{S_0}$ is the part of $\mathcal{D}$ that characterizes S_0, $\mathcal{D}_{una}$ is the set of unique name axioms for primitive actions, and $\phi' = \mathcal{R}(\phi, \mathbf{a})$. So to solve the projection problem, it is sufficient, to regress the formula using the given actions, and then to determine whether result holds in the initial situation, a much simpler entailment.

Regression has proved to be a powerful method for reasoning about a dynamic world, reducing it to reasoning about a static initial situation. However, it does have a serious drawback. Imagine a long-lived agent that has performed thousands or even millions of actions in its lifetime, and which at some point, needs to determine whether some condition currently holds. Regression involves transforming this condition back through those many actions, and then determining whether the transformed condition held initially. This is not an ideal way of staying up to date.

The alternative to regression is *progression* [65]. In this case, we look for a progression operator $\mathcal{P}$ that can transform an initial database $\mathcal{D}_{S_0}$ into the database that results after performing an action. More precisely, we want to have that

$$\mathcal{D} \models \phi[do(\mathbf{a}, S_0)] \quad \text{iff} \quad \mathcal{D}_{una} \cup \mathcal{D}'_0 \models \phi[S_0],$$

where $\mathcal{D}_{S_0}$ is the part of $\mathcal{D}$ that characterizes S_0, $\mathcal{D}_{una}$ is the set of unique name axioms for primitive actions, and $\mathcal{D}'_0 = \mathcal{P}(\mathcal{D}_{S_0}, \mathbf{a})$. The idea is that as actions are performed, an agent would change its database about the initial situation, so that to determine if ϕ held after doing actions $\mathbf{a}$, it would be sufficient to determine if ϕ held in the progressed situation (with no further actions), again a much simpler entailment. Moreover, unlike the case with regression, an agent can use its *mental idle time* (for example, while it is performing physical actions) to keep its database up to date. If it is unable to keep up, it is easy to imagine using regression until the database is fully progressed.

There are, however, drawbacks with progression as well. For one thing, it is geared to answering questions about the *current* situation only. In progressing a database forward, we effectively lose the historical information about what held in the past. It is, in other words, a form of *forgetting* [64, 47]. While questions about a current situation can reasonably be expected to be the most common, they are not the only meaningful ones.

A more serious concern with progression is that it is not always possible. As Lin and Reiter show [65], there are simple cases of basic action theories where there is no operator $\mathcal{P}$ with the properties we want. (More precisely, the desired $\mathcal{D}'_0$ would not be first-order definable.) To have a well-defined projection operator, it is necessary to impose further restrictions on the sorts of action theories we use, as we will see below.

Reasoning with Closed and Open World Knowledge Bases

So far, we have assumed like Reiter, that $\mathcal{D}_{S_0}$ is any collection of formulas uniform in S_0. Regression reduces the projection problem to that of calculating logical consequences of $\mathcal{D}_{S_0}$. In practice, however, we would like to reduce it to a much more tractable problem than ordinary first-order logical entailment. It it is quite common for applications to assume that $\mathcal{D}_{S_0}$ satisfies additional constraints: domain closure, unique names, and the closed-word assumption [80]. With these, for all practical purposes, $\mathcal{D}_{S_0}$ does behave like a database, and the entailment problem becomes one of database query evaluation. Furthermore, progression is well defined, and behaves like an ordinary database transaction.

Even without using (relational) database technology, the advantage of having a $\mathcal{D}_{S_0}$ constrained in this way is significant. For example, it allows us to use Prolog technology directly to perform projection. For example, to find out if $(\phi \vee \psi)$ holds, it is sufficient to determine if ϕ holds or if ψ holds; to find out if $\neg\phi$ holds, it is sufficient to determine if ϕ does not hold (using negation as failure), and so on. None of these are possible with an unconstrained $\mathcal{D}_{S_0}$.

This comes at a price, however. The unique name, domain closure and closed-world assumptions amount to assuming that we have *complete knowledge* about S_0: anytime we cannot infer that ϕ holds, it will be because we are inferring that $\neg\phi$ holds. We will never have the status of ϕ undecided. This is obviously a very strong assumption. Indeed we would expect that a typical agent might start with incomplete knowledge, and only acquire the information it needs by actively *sensing* its environment as necessary.

A proposal for modifying Reiter's proposal for the projection problem along these lines was made by De Giacomo and Levesque [27]. They show that a modified version of regression can be made to work with sensing information. They also consider how closed-world reasoning can be used in an open world using what they call *just-in-time queries*. In a nutshell, they require that queries be evaluated only in situations where enough sensing has taken place to give complete information about the query. Overall, the knowledge can be incomplete, but it will be locally complete, and allow us to use closed-world techniques.

Another independent proposal for dealing effectively with open-world reasoning is that of Liu and Levesque [106]. (Related proposals are made by Son and Baral [96] and by Amir and Russell [2].) They show that what they call *proper knowledge bases* represent open-world knowledge. They define a form of progression for these knowledge bases that provides an efficient solution to the projection problem that is always logically sound, and under certain circumstances, also logically complete. The restrictions involve the type of successor-state axioms that appear in the action theory $\mathcal{D}$: they require action theories that are *local-effect* (actions only change the properties of the objects that are parameters of the action) and *context-complete* (either the actions are context-free or there is complete knowledge about the context of the context-dependent ones). Vassos and Levesque [102] extended this approach to more general theories, while relying on the assumption that there is a finite domain and a restricted form of disjunctive knowledge in the initial database in order to remain first-order and tractable. In [103], they also show that an alternative definition of progression that is always first-order is nonetheless correct for reasoning about a large class of sentences. As well, in [101] they reconsider Lin and Reiter's progression (actually a slight variant that solves a few problems) and show that in case actions have only local effects, this form of progression is always first-order representable; moreover, for a restricted class of local-effect axioms they show how to construct a progressed database that remains finite.

The Offline Execution Semantics

Now let's return to the formal semantics of IndiGolog. This semantics is based on that of ConGolog, so we will go over the latter first. In [24], a single step structural operational (transition system) semantics in the style of [75] is defined for ConGolog programs. Two special predicates $Trans$ and $Final$ are introduced. $Trans(\delta, s, \delta', s')$ means that by executing program δ starting in situation s, one can get to situation s' in one elementary step with the program δ' remaining to be executed. $Final(\delta, s)$ means that program δ may successfully terminate in situation s.

Note that this semantics requires quantification over programs. To allow for this, [24] develops an encoding of programs as first-order terms in the logical language (observe that programs as such, cannot in general be first-order terms, since they mention formulas in tests, and the operator π in $\pi x.\delta$ is a sort of quantifier, hence an encoding is needed).[2] Encoding programs as first-order terms, although it requires some care (e.g. introducing constants denoting variables and defining substitution explicitly in the language), does not pose any major problem.[3] In the following we abstract from the details of the encoding as much as possible, and essentially use programs within formulas as if they were already first-order terms. The full encoding is given in [24]. (In [36], an approach to handling ConGolog programs that does not rely on any type of encoding is presented. There, high-level programs are compiled into standard situation calculus basic action theories such that the executable situations are exactly those that are permitted by the program.)

[2] In the original presentation of Golog [62], a simpler semantics was given where $Do(\delta, s, s')$ was only an abbreviation for a formula $\Phi(s, s')$ that did not mention the program δ (or any other programs), thus avoiding the need to reify programs. However, when dealing with concurrency, it is more convenient to use a transition semantics.

[3] Observe that we assume that formulas that occur in tests never mention programs, so it is impossible to build self-referential sentences.

The predicate *Trans* for programs without procedures is characterized by the following set of axioms $\mathcal{T}$ (here as in the rest of the chapter, free variables are assumed to be universally quantified):

1. Empty program:
$$Trans(nil, s, \delta', s') \equiv False.$$

2. Primitive actions:
$$Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s).$$

3. Test/wait actions:
$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s.$$

4. Sequence:
$$\begin{aligned}&Trans(\delta_1; \delta_2, s, \delta', s') \equiv \\ &\quad \exists\gamma.\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s').\end{aligned}$$

5. Nondeterministic branch:
$$Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s').$$

6. Nondeterministic choice of argument:
$$Trans(\pi v.\delta, s, \delta', s') \equiv \exists x. Trans(\delta^v_x, s, \delta', s').$$

7. Nondeterministic iteration:
$$Trans(\delta^*, s, \delta', s') \equiv \exists\gamma.(\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \gamma, s').$$

8. Synchronized conditional:
$$\begin{aligned}&Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s, \delta', s') \equiv \\ &\quad \phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s').\end{aligned}$$

9. Synchronized loop:
$$\begin{aligned}&Trans(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s, \delta', s') \equiv \\ &\quad \exists\gamma.(\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s').\end{aligned}$$

10. Concurrent execution:
$$\begin{aligned}&Trans(\delta_1 \parallel \delta_2, s, \delta', s') \equiv \\ &\quad \exists\gamma.\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \exists\gamma.\delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s').\end{aligned}$$

11. Prioritized concurrency:

$$
\begin{aligned}
&\mathit{Trans}(\delta_1 \rangle\!\rangle\, \delta_2, s, \delta', s') \;\; \equiv \\
&\qquad \exists\gamma.\delta' = (\gamma \rangle\!\rangle\, \delta_2) \wedge \mathit{Trans}(\delta_1, s, \gamma, s') \;\; \vee \\
&\qquad \exists\gamma.\delta' = (\delta_1 \rangle\!\rangle\, \gamma) \wedge \mathit{Trans}(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''. \mathit{Trans}(\delta_1, s, \zeta, s'').
\end{aligned}
$$

12. Concurrent iteration:

$$\mathit{Trans}(\delta^{\|}, s, \delta', s') \;\equiv\; \exists\gamma.\delta' = (\gamma \parallel \delta^{\|}) \wedge \mathit{Trans}(\delta, s, \gamma, s').$$

The assertions above characterize when a configuration (δ, s) can evolve (in a single step) to a configuration (δ', s'). Intuitively they can be read as follows:

1. (nil, s) cannot evolve to any configuration.
2. (a, s) evolves to $(\mathit{nil}, \mathit{do}(a[s], s))$, provided that $a[s]$ is possible in s. After having performed a, nothing remains to be performed and hence nil is returned. Note that in $\mathit{Trans}(a, s, \delta', s')$, a stands for the program term encoding the corresponding situation calculus action, while Poss and do take the latter as argument; we take the function $\cdot[\cdot]$ as mapping the program term a into the corresponding situation calculus action $a[s]$, as well as replacing now by the situation s. The details of how this function is defined are in [24].
3. $(\phi?, s)$ evolves to (nil, s), provided that $\phi[s]$ holds, otherwise it cannot proceed. Note that the situation remains unchanged. Analogously to the previous case, we take the function $\cdot[\cdot]$ as mapping the program term for condition ϕ into the corresponding situation calculus formulas $\phi[s]$, as well as replacing now by the situation s (see [24] for details).
4. $(\delta_1; \delta_2, s)$ can evolve to $(\delta_1'; \delta_2, s')$, provided that (δ_1, s) can evolve to (δ_1', s'). Moreover it can also evolve to (δ_2', s'), provided that (δ_1, s) is a final configuration and (δ_2, s) can evolve to (δ_2', s').
5. $(\delta_1 \mid \delta_2, s)$ can evolve to (δ', s'), provided that either (δ_1, s) or (δ_2, s) can do so.
6. $(\pi v.\delta, s)$ can evolve to (δ', s'), provided that there exists an x such that (δ^v_x, s) can evolve to (δ', s'). Here δ^v_x is the program resulting from δ by substituting v with the variable x.[4]
7. (δ^*, s) can evolve to $(\delta'; \delta^*, s')$ provided that (δ, s) can evolve to (δ', s'). Observe that (δ^*, s) can also not evolve at all, (δ^*, s) being final by definition (see below).
8. $(\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2\ \textbf{endIf}, s)$ can evolve to (δ', s'), if either $\phi[s]$ holds and (δ_1, s) can do so, or $\neg\phi[s]$ holds and (δ_2, s) can do so.
9. $(\textbf{while}\ \phi\ \textbf{do}\ \delta\ \textbf{endWhile}, s)$ can evolve to $(\delta'; \textbf{while}\ \phi\ \textbf{do}\ \delta\ \textbf{endWhile}, s')$, if $\phi[s]$ holds and (δ, s) can evolve to (δ', s').

[4] More formally, in the program term δ, v is substituted by a term of the form $\mathit{nameOf}(x)$, where nameOf is used to convert situation calculus objects/actions into program terms of the corresponding sort (see [24]).

10. You single step $(\delta_1 \parallel \delta_2)$ by single stepping either δ_1 or δ_2 and leaving the other process unchanged.
11. The $(\delta_1 \rangle\!\rangle \delta_2)$ construct is identical, except that you are only allowed to single step δ_2 if there is no legal step for δ_1. This ensures that δ_1 will execute as long as it is possible for it to do so.
12. Finally, you single step $\delta^{\parallel}$ by single stepping δ, and what is left is the remainder of δ as well as $\delta^{\parallel}$ itself. This allows an unbounded number of instances of δ to be running.

Observe that with $(\delta_1 \parallel \delta_2)$, if both δ_1 and δ_2 are always able to execute, the amount of interleaving between them is left completely open. It is legal to execute one of them completely before even starting the other, and it also legal to switch back and forth after each primitive or wait action.[5]

$Final(\delta, s)$ tells us whether a program δ can be considered to be already in a *final state* (legally terminated) in the situation s. Obviously we have $Final(nil, s)$, but also $Final(\delta^*, s)$ since δ^* requires 0 or more repetitions of δ and so it is possible to not execute δ at all, the program completing immediately.

The predicate *Final* for programs without procedures is characterized by the set of axioms $\mathcal{F}$:

1. Empty program:
$$Final(nil, s) \equiv True.$$
2. Primitive action:
$$Final(a, s) \equiv False.$$
3. Test/wait action:
$$Final(\phi?, s) \equiv False.$$
4. Sequence:
$$Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s).$$
5. Nondeterministic branch:
$$Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s).$$
6. Nondeterministic choice of argument:
$$Final(\pi v.\delta, s) \equiv \exists x. Final(\delta^v_x, s).$$
7. Nondeterministic iteration:
$$Final(\delta^*, s) \equiv True.$$
8. Synchronized conditional:

[5] It is not hard to define new concurrency constructs $\parallel_{min}$ and $\parallel_{max}$ that require the amount of interleaving to be minimized or maximized respectively.

$$\begin{aligned} \mathit{Final}(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endIf}, s) &\equiv \\ &\phi[s] \land \mathit{Final}(\delta_1, s) \lor \neg\phi[s] \land \mathit{Final}(\delta_2, s). \end{aligned}$$

9. Synchronized loop:

$$\mathit{Final}(\mathbf{while}\ \phi\ \mathbf{do}\ \delta\ \mathbf{endWhile}, s) \ \equiv\ \neg\phi[s] \lor \mathit{Final}(\delta, s).$$

10. Concurrent execution:

$$\mathit{Final}(\delta_1 \parallel \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \land \mathit{Final}(\delta_2, s).$$

11. Prioritized concurrency:

$$\mathit{Final}(\delta_1 \rangle\!\rangle\ \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \land \mathit{Final}(\delta_2, s).$$

12. Concurrent iteration:

$$\mathit{Final}(\delta^{\parallel}, s) \equiv \mathit{True}.$$

The assertions above can be read as follows:

1. (nil, s) is a final configuration.
2. (a, s) is not final, indeed the program consisting of the primitive action a cannot be considered completed until it has performed a.
3. $(\phi?, s)$ is not final, indeed the program consisting of the test action $\phi?$ cannot be considered completed until it has performed the test $\phi?$.
4. $(\delta_1; \delta_2, s)$ can be considered completed if both (δ_1, s) and (δ_2, s) are final.
5. $(\delta_1 \mid \delta_2, s)$ can be considered completed if either (δ_1, s) or (δ_2, s) is final.
6. $(\pi v.\delta, s)$ can be considered completed, provided that there exists an x such that (δ^v_x, s) is final, where δ^v_x is obtained from δ by substituting v with x.
7. (δ^*, s) is a final configuration, since δ^* is allowed to execute 0 times.
8. $(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endIf}, s)$ can be considered completed, if either $\phi[s]$ holds and (δ_1, s) is final, or if $\neg\phi[s]$ holds and (δ_2, s) is final.
9. $(\mathbf{while}\ \phi\ \mathbf{do}\ \delta\ \mathbf{endWhile}, s)$ can be considered completed if either $\neg\phi[s]$ holds or (δ, s) is final.
10. $(\delta_1 \parallel \delta_2)$ can be considered completed if both δ_1 and δ_2 are final.
11. $(\delta_1 \rangle\!\rangle\ \delta_2)$ is handled identically with the previous case.
12. $\delta^{\parallel}$ is a final configuration, since $\delta^{\parallel}$ is allowed to execute 0 instances of δ.

The ConGolog semantics handles procedure definitions and procedure calls in a standard way with call-by-value parameter passing and lexical scoping. We leave out the axioms that handle this; they can be found in [24].[6]

In the following we denote by $\mathcal{C}$ the set of axioms for *Trans* and *Final* plus those needed for the encoding of programs as first-order terms.

[6] Note that when the number of recursive calls is unbounded, this requires defining *Trans* and *Final* using a second order formula. In ConGolog a procedure call is not a transition (only primitive actions and tests are), so one must allow for an arbitrarily large but finite number of procedure calls in a transition; see [24].

Regarding interrupts, it turns out that these can be explained using other constructs of ConGolog:

$$
\begin{array}{l}
\langle\ \phi \rightarrow \delta\ \rangle \overset{\text{def}}{=} \textbf{while } \mathit{Interrupts\_running} \textbf{ do} \\
\qquad\qquad \textbf{if } \phi \textbf{ then } \delta \textbf{ else } \mathit{False}? \textbf{ endIf} \\
\qquad \textbf{endWhile}
\end{array}
$$

To see how this works, first assume that the special fluent $Interrupts\_running$ is identically *True*. When an interrupt $\langle\phi \rightarrow \delta\rangle$ gets control, it repeatedly executes δ until ϕ becomes false, at which point it blocks, releasing control to any other process in the program that is able to execute. Note that according to the above definition of *Trans*, no transition occurs between the test condition in a while-loop or an if-then-else and the body. In effect, if ϕ becomes false, the process blocks right at the beginning of the loop, until some other action makes ϕ true and resumes the loop. To actually terminate the loop, we use a special primitive action *stop_interrupts*, whose only effect is to make $Interrupts\_running$ false. Thus, we imagine that to execute a program δ containing interrupts, we would actually execute the program $\{start\_interrupts\,;\,(\delta\ \rangle\!\rangle\ stop\_interrupts)\}$ which has the effect of stopping all blocked interrupt loops in δ at the lowest priority, *i.e.* when there are no more actions in δ that can be executed.

Offline executions of programs, which are the kind of executions originally proposed for Golog [62] and ConGolog [24], are characterized using the $Do(\delta, s, s')$ predicate, which means that there is an execution of program δ that starts in situation s and terminates in situation s':

$$Do(\delta, s, s') \ \overset{\text{def}}{=}\ \exists\delta'.Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'),$$

where $Trans^*$ is the reflexive transitive closure of $Trans$.[7] Thus there is an execution of program δ that starts in situation s and terminates in situation s' if and only if we can perform 0 or more transitions from program δ in situation s to reach situation s' with program δ' remaining, at which point one may legally terminate.

An *offline execution* of δ from s is a sequence of actions $a_1, \ldots, a_n$ such that: $\mathcal{D} \cup \mathcal{C} \models Do(\delta, s, do(a_n, \ldots, do(a_1, s) \ldots))$, where $\mathcal{D}$ is an action theory as mentioned above, and $\mathcal{C}$ is a set of axioms defining the predicates $Trans$ and $Final$ and the encoding of programs as first-order terms [24].

[7] $Trans^*$ can be defined as the (second-order) situation calculus formula:

$$\mathit{Trans}^*(\delta, s, \delta', s') \ \overset{\text{def}}{=}\ \forall T.[\ldots \ \supset\ T(\delta, s, \delta', s')],$$

where ... stands for the conjunction of the universal closure of the following implications:

$$
\begin{array}{l}
\mathit{True} \ \supset\ T(\delta, s, \delta, s), \\
\mathit{Trans}(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \ \supset\ T(\delta, s, \delta', s').
\end{array}
$$

The Online Execution Semantics

The offline execution model of Golog and ConGolog requires the executor to search over the whole program to find a complete execution before performing any action. As mentioned earlier, this is problematic for agents that are long lived or need to sense their environment as they operate. The *online execution* model of IndiGolog [26, 87] addresses this. Imagine that we started with some program δ_0 in S_0, and that at some later point we have executed certain actions $a_1, \ldots a_k$ and have obtained sensing results $\mu_1, \ldots \mu_k$ from them, i.e. we are now in history $\sigma = (a_1, \mu_1) \cdot \ldots \cdot (a_k, \mu_k)$, with program δ remaining to be executed. The *online high-level program execution task* then is to find out what to do next, defined by:

- stop, if $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Final(\delta, end[\sigma])$;
- return the remaining program δ', if

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', end[\sigma]),$$

  and no action is required in this step;
- return action a and δ', if

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', do(a, end[\sigma])).$$

So the online version of program execution uses the sensing information that has been accumulated so far to decide if it should terminate, take a step of the program with no action required, or take a step with a single action required. In the case that an action is required, the agent can be instructed to perform the action, gather any sensing information this provides, and the online execution process iterates.

As part of this online execution cycle, one can also monitor for the occurrence of exogenous actions/events. When an exogenous action is detected it can be added to the history σ, possibly causing an update in the values of various fluents. The program can then monitor for this and execute a "reaction" when appropriate (e.g. using an interrupt).

The IndiGolog semantics of [26] defines an *online execution* of a program δ starting from a history σ, as a sequence of *online configurations* $(\delta_0 = \delta, \sigma_0 = \sigma), \ldots, (\delta_n, \sigma_n)$ such that for $i = 0, \ldots, n-1$:

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma_i]\} \models Trans(\delta_i, end[\sigma_i], \delta_{i+1}, end[\sigma_{i+1}]),$$

$$\sigma_{i+1} = \begin{cases} \sigma_i & \text{if } end[\sigma_{i+1}] = end[\sigma_i], \\ \sigma_i \cdot (a, \mu) & \text{if } end[\sigma_{i+1}] = do(a, end[\sigma_i]) \text{ and } a \text{ returns } \mu. \end{cases}$$

An *online execution successfully terminates* if

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma_n]\} \models Final(\delta_n, end[\sigma_n]).$$

Note that this definition assumes that exogenous actions do not occur; one can easily generalize the definition to allow them.

The Search Operator

The online execution of a high-level program does not require a reasoner to determine a lengthy course of action, formed perhaps of millions of actions, before executing the first step in the world. It also gets to use the sensing information provided by the first n actions performed so far in deciding what the $(n+1)$'th action should be. On the other hand, once an action has been executed in the world, there may be no way of backtracking if it is later found out that a nondeterministic choice was resolved incorrectly. As a result, an online execution of a program may fail where an offline execution would succeed.

To cope with the fact that it may be impossible to backtrack on actions executed in the real world, IndiGolog incorporates a new programming construct, namely the *search operator*. The idea is that given any program δ the program $\Sigma(\delta)$ executes online just like δ does offline. In other words, before taking any action, it first ensures using offline reasoning that this step can be followed successfully by the rest of δ. More precisely, according to [26], the semantics of the search operator is that

$$\mathit{Trans}(\Sigma(\delta), s, \Sigma(\delta'), s') \;\equiv\; \mathit{Trans}(\delta, s, \delta', s') \wedge \exists s^*.\mathit{Do}(\delta', s', s^*).$$

If δ is the entire program under consideration, $\Sigma(\delta)$ emulates complete offline execution. But consider $[\delta_1 \,;\, \delta_2]$. The execution of $\Sigma([\delta_1 \,;\, \delta_2])$ would make any choice in δ_1 depend on the ability to successfully complete δ_2. But $[\Sigma(\delta_1) \,;\, \delta_2]$ would allow the execution of the two pieces to be done separately: it would be necessary to ensure the successful completion of δ_1 before taking any steps, but consideration of δ_2 is deferred. If we imagine, for example, that δ_2 is a large high-level program, with hundreds of pages of code, perhaps containing Σ operators of its own, this can make the difference between a scheme that is practical and one that is only of theoretical interest.

Being able to search still raises the question of how much offline reasoning should be performed in an online system. The more offline reasoning we do, the safer the execution will be, as we get to look further into the future in deciding what choices to make now. On the other hand, in spending time doing this reasoning, we are detached from the world and will not be as responsive. This issue is very clearly evident in time-critical applications such as robot soccer [34] where there is very little time between action choices to contemplate the future. Sardina has cast this problem as the choice between deliberation and reactivity [86], and see also [6].

Another issue that arises in this setting is the form of the offline reasoning. Since an online system allows for a robot to *acquire information during* *execution* (via sensing actions, or passive sensors, or exogenous events), how should the agent deal with this during offline deliberation [23]. The simplest possibility is to say that it ignores any such information in the plan for the future that it is constructing. This is essentially what the semantics for the search operator given above does. It ensures that there exist a complete execution of the program (in the absence of exogenous actions), but it does not ensure that the agent has enough information to know what to do. For example, consider the program

$$\Sigma((a \mid sense_P); \mathbf{if}\ P\ \mathbf{then}\ b\ \mathbf{else}\ c\ \mathbf{endIf})$$

and an agent that does not know initially whether P hold. The search semantics given above says that the agent may do action a, since it knows that afterwards there will be some way to successfully complete the execution. But in fact the agent will then get stuck not knowing which branch to take. A more adequate deliberation mechanism would require the execution of $sense_P$ as the first step, since it does guarantee the complete executability of the program, unlike action a.

An even more sophisticated deliberation approach would have the agent construct a plan that would prescribe different behaviors depending on the information acquired during execution. This is *conditional planning* (see, for example, [10, 73]). For the example above, this would produce a plan that requires the agent to first do $sense_P$, and then if it had sensed that P held, to do b and otherwise to do c; then the agent is guaranteed to always know what action to perform to complete the execution. One form of this has been incorporated in high-level execution by Lakemeyer [48] and Sardina [84]. In [87], a semantics for such a sophisticated search operator is axiomatized in a version of the situation calculus extended with a possible-world model of knowledge. [25] and [88] develop non-epistemic, metatheoretic accounts of this kind of deliberation, and discuss difficulties that arise with programs involving unbounded iteration.

Another possibility is to attempt to *simulate* what will happen external to the agent *including exogenous events*, and use this information during the deliberation [56]. This is a kind of contingency planning [77]. In [41], this idea is taken even further: at deliberation time a robot uses, for example, a model of its navigation system by computing, say, piece-wise linear approximations of its trajectory; at execution time, this model is then replaced by the real navigation system, which provides position updates as exogenous actions. [54] develops an account of deliberation where the agent's high level program must be executed against a dynamic environment also modeled as a nondeterministic program. Deliberation must produce a deterministic conditional plan that can be successfully executed against all possible executions of the environment program.

Another issue arises whenever an agent performs at least some amount of lookahead in deciding what to do. What should the agent do when the world (as determined by its sensors) does not conform to its predictions (as determined by its action theory)? First steps in logically formalizing this possibility were taken by De Giacomo et al. [30] in what they call *execution monitoring*. The deliberation model formalized in [87] incorporates execution monitoring and replanning. Lastly, a search operator that deliberates on a program relative to a set of goals is described in [91].

As we have seen, it is possible to define different search/deliberation constructs with varying degrees of sophistication. For many cases, however, the simple search operator defined above suffices, and implementations for it can easily be developed; these are provably correct under some plausible assumptions (for instance, when the truth value of all tests in a program will be known by the time they are evaluated, as in the "just-in-time histories" of [26, 28]).

We close the section by noting that Shapiro et. al [95, 94] have developed a verification environment, CASLve, for an extension of ConGolog that supports multiagent plans and modeling agents' knowledge and goals, based on the PVS theorem proving/verification system.[8] Some non-trivial programs have been formally verified.

2.3 Software Engineering Issues and Other Features of the Language

At this point, our language offers only limited support for building large software systems. It supports procedural abstraction, but not modules. Very complex agents can be decomposed into simpler agents that cooperate, and each can be implemented separately. One important feature that we do offer is that the agent's beliefs are automatically updated based on a declarative action theory, which supports the use of complex domain models, and helps avoid the errors that typically occur when such models are manually updated.

Our language/platform is implemented in SWI-Prolog, which provides flexible mechanisms for interfacing with other programming languages such as Java or C, and for socket communication. There are also libraries for interfacing with the JADE and OAA multiagent platforms; see the end of Section 3 for details.

Note that ConGolog has been used as a formal specification/modeling language for software requirements engineering [50, 104]. Such ConGolog specifications can be validated by simulated execution or verification. One may even use them for early prototyping. IndiGolog could be used in a requirements-driven approach to software development such as Tropos [19].

Our approach provides for a high degree of extensibility. The declarative language definition supports the easy addition of new programming con-

[8] `http://pvs.csl.sri.com/`

structs. The underlying situation calculus framework supports many extensions in the way change is modeled, e.g. continuous change, stochastic effects, simultaneous actions, etc. [82] Evidence for this extensibility is that the language has been extended numerous times. Note that there is currently no specific built-in support for mobile agents.

What we have seen so far, is the formal specification of our language. If one wants to actually run an agent programmed in IndiGolog in the real-world, one needs to address many practical issues that are not dealt with in the formal account. For instance, when an action transition step is performed in an online execution, the action ought to be carried out in the environment where it is supposed to occur, and its sensing outcome needs to be extracted as well. Similarly, a mechanism for recognizing and assimilating external exogenous events needs to be developed. All this requires a framework in which an online execution is realized in the context of a real (external) environment. In the next section, we describe a platform that does exactly this.

3 Platform

We now turn to describing what is probably the most advanced IndiGolog based platform currently available. This platform[9] was originally developed at the University of Toronto and is based on LeGolog [60], which is in turn based on a proof-of-concept simple implementation originally written by Hector Levesque. The platform is a *logic-programming* implementation of IndiGolog that allows the incremental execution of high-level Golog-like programs [85]. This is the only implementation of IndiGolog that is modular and easily extensible so as to deal with any external platform, as long as the suitable interfacing modules are programmed (see below). Among others, the system has been used to control the LEGO MINDSTORM [60] and the ER1 Evolution[10] robots, as well as other software agents [92], to coordinate mobile actors in pervasive scenarios [52], and to incorporate automated planning into cognitive agents [21, 20].

Although most of the code is written in vanilla Prolog, the overall architecture is written in the well-known open source SWI-Prolog[11] [105]. SWI-Prolog provides flexible mechanisms for interfacing with other programming languages such as Java or C, allows the development of multi-threaded applications, and provides support for socket communication and constraint solving.

Generally speaking, the IndiGolog implementation provides an incremental interpreter of high-level programs as well as a framework for dealing with the *real* execution of these programs on *concrete* platforms or devices. This

[9] Available at `http://sourceforge.net/projects/indigolog/`.

[10] `http://www.evolution.com/er1/`

[11] `http://www.swi-prolog.org/`

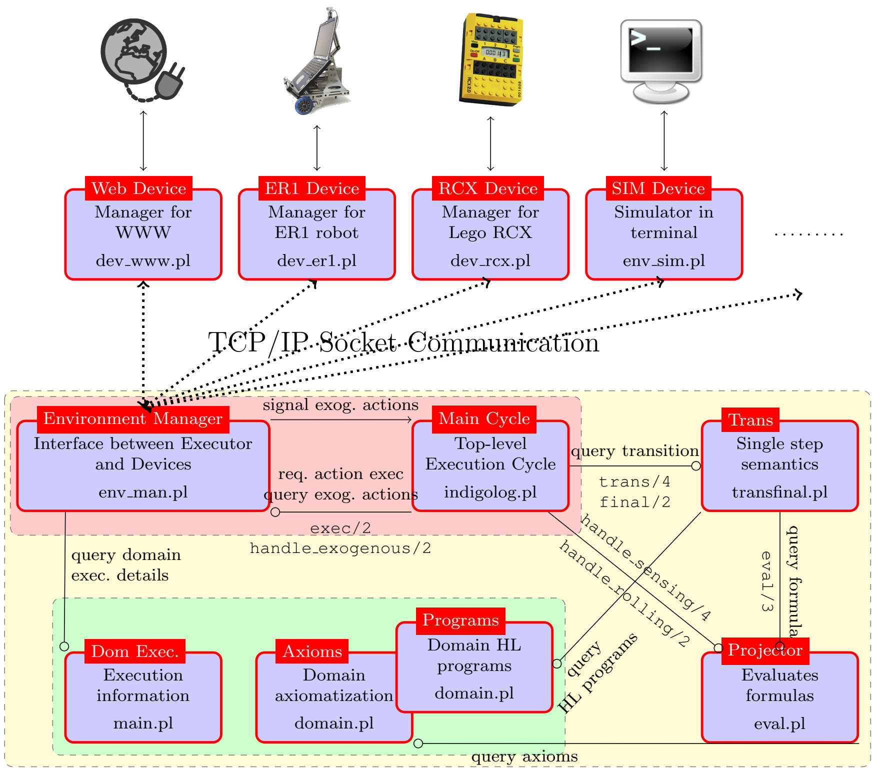


Fig. 1 The IndiGolog implementation architecture. Links with a circle ending represent goal posted to the circled module.

amounts to handling the real execution of actions on concrete devices (e.g., a real robot platform), the collection of sensing outcome information (e.g., retrieving some sensor's output), and the detection of exogenous events happening in the world. To that end, the architecture is modularly divided into six parts, namely, *(i)* the top-level main cycle; *(ii)* the language semantics; *(iii)* the temporal projector; *(vi)* the environment manager; *(v)* the set of device managers; and finally *(vi)* the domain application. The first four modules are completely domain independent, whereas the last two are designed for specific domain(s). The architecture is depicted in Figure 1.

The Top-Level Main Cycle and Language Semantics

The top-level main cycle implements the IndiGolog online execution account explained in Section 2.2. It realizes the *sense-think-act* loop well-known in the agent community [46].

The main predicate of the main cycle is `indigo/2`; a goal of the form `indigo(E,H)` states that the high-level program `E` is to be executed online at history `H`. As in the definition of online executions, the main cycle strongly relies on the meaning of the language constructs. Hence, clauses for relations $Trans$ and $Final$ are needed for each of the constructs. These two relations are modeled with Prolog predicates `trans/4` and `final/2` and are defined in the language semantics module (see below).

The following is a simplified version of the top-level main cycle:

```
indigo(E,H):- handle_exogenous(H,H2), !, indigo(E,H2).
indigo(E,H):- handle_rolling(H,H2), !, indigo(E,H2).
indigo(E,H):- catch(final(E,H), exog, indigo(E,H)).
indigo(E,H):- catch(trans(E,H,E1,H1), exog, indigo(E,H)),
  (var(H1) -> true ;
   H1=H -> indigo(E1,H) ;
   H1=[A|H] -> exec(A,S), handle_sensing(H,A,S,H2), indigo(E1,H2)).
```

The first thing the main cycle does is to assimilate all exogenous events that have occurred since the last execution step. To that end, predicate `handle_exogenous/2`, provided by the environment manager (see below), is used to transform the current history `H` into the new history `H2` containing the new exogenous events—if no exogenous actions occurred during the cycle, then `handle_exogenous/2` just fails.

In the second clause, predicate `handle_rolling/2` may "roll forward" the current history `H`, for example, if its length has exceeded some threshold, yielding then the new (shorter) history `H2`. This amounts to doing *progression* of the current history [65, 101]. Since progressing the current history is a task related to the background action theory being used to execute the program, the predicate `handle_rolling/2` is implemented by the temporal projector (see below).

After all exogenous actions have been assimilated and the history progressed as needed, the main cycle goes on to actually executing the high-level program `E`. First, if the current program to be executed is terminating in the current history, then the top-level goal `indigo/2` simply succeeds (third clause). Otherwise, the interpreter checks whether the program can evolve a single step (fourth clause) by relying on predicate `trans/4` (explained below). If the program evolves without executing any action, then the history remains unchanged and we continue to execute the remaining program from the same history. If, however, the step involves performing an action, then this action is executed and incorporated into the current history, together with its sensing result (if any), before continuing the execution of the remaining program. The actual execution of the action is implemented via predicate `exec/2`, provided by the environment manager (described below), which returns the sensing outcome of the action. Finally, `handle_sensing/4` returns the new history obtained by incorporating the executed action and its sensing outcome into the current history (this predicate is provided by the temporal projector, to allow for alternative implementations, e.g. through progression).

Note that, in the third and fourth clauses above, goals `final/2` and `trans/4` are posted within a `catch/3` extra-logical predicate.[12] The point is that proving `final/2` or `trans/4` could be time consuming, as there may be substantial reasoning involved. If, during such reasoning, an exogenous event happens, such reasoning is not guaranteed to be adequate anymore, as the history of events has changed. In that case, the interpreter simply *aborts* the single-step reasoning (i.e., goal `final/2` or `trans/4`) and re-starts the cycle, which in turn will first assimilate the just occurred events.

As mentioned above, the top-level loop relies on two central predicates, namely, `final/2` and `trans/4`. These predicates implement relations $Trans$ and $Final$, giving the single step semantics for each of the constructs in the language. It is convenient, however, to use an implementation of these predicates defined over histories instead of situations. So, for example, these are the corresponding clauses for sequence (represented as a list), nondeterministic choice of programs, tests, and primitive actions:

```
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(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).

trans(?(P),H,[],H) :- eval(P,H,true).
trans(E,H,[],[E|H]) :- action(E), poss(E,P), eval(P,H,true).
/* Obs: no final/2 clauses for action and test programs */
```

As is easy to observe, these Prolog clauses are almost directly "lifted" from the corresponding axioms for $Trans$ and $Final$. Predicates `action/1` and `poss/2` specify the actions of the domain and their corresponding precondition axioms; both are defined in the domain axiomatization (see below). More importantly, `eval/3` is used to check the truth of a condition at a certain history, and is provided by the temporal projector, described next.

A naive implementation of the search operator would deliberate from scratch at every point of its incremental execution. It is clear, however, that one can do better than that, and cache the successful plan obtained and avoid replanning in most cases:

```
final(search(E),H) :- final(E,H).
trans(search(E),H,path(E1,L),H1) :-
      trans(E,H,E1,H1), findpath(E1,H1,L).
```

[12] `catch(:Goal, +Catcher, :Recover)` behaves as `call/1`, except that if an exception is raised while `Goal` executes, and the `Catcher` unifies with the exception's name, then `Goal` is aborted and `Recover` is called.

```
/* findpath(E,H,L): solve (E,H) and store the path in list L  */
/* L = list of configurations (Ei,Hi) expected along the path */
findpath(E,H,[(E,H)]) :- final(E,H).
findpath(E,H,[(E,H)|L]) :- trans(E,H,E1,H1), findpath(E1,H1,L).

/* When we have a path(E,L), try to advance using list L */
final(path(E,[(E,H)]),H) :- !. /* last step */
final(path(E,_),H) :- final(E,H). /* off path; re-check */
trans(path(E,[(E,H),(E1,H1)|L]),H,path(E1,[(E1,H1)|L]),H1) :- !.
trans(path(E,_),H,E1,H1) :-
      trans(search(E),H,E1,H1).  /* redo search */
```

So, when a search block is first solved, the whole solution path found is stored as the sequence of configurations that are expected. If the actual configurations reached match, then steps are performed without any reasoning (first `final/2` and `trans/4` clauses for program `path(E,L)`). If, on the other hand, the actual configuration does not match the one expected next, for example, because an exogenous action occurred and the history thus changed, replanning is performed to look for an alternative path (second `final/2` and `trans/4` clauses for program `path(E,L)`). Other variants of the search operator are provided, such as a `searchc(E)` construct in the spirit of [48, 84] that constructs a conditional plan that solves `E`.

Finally, we point out that by decoupling `trans/4` and `final/2` from the main cycle and the temporal projector, one can change the actual high-level programming language used. In that way, one could use the architecture to execute any agent language with a single-step operational semantics. For example, one could use the architecture to execute AgentSpeak agents [78], by suitably recasting the derivation rules of such BDI languages into `trans/4` and `final/2` clauses—in this case, the program `E` would stand for the agent's current active intentions and `H` for the history of executed actions and external events.

The Temporal Projector

The temporal projector is in charge of maintaining the agent's beliefs about the world and evaluating a formula relative to a history. It could be realized with standard database technology (see [29]) or with an evaluation procedure for some reasoning about action formalism. In the context of the situation calculus, for instance, one could use temporal projectors for basic action theories [74], guarded theories [27], or even fluent calculus theories of action [98]. The only requirement for the projector module is to provide an implementation of predicate `eval/3`: goal `eval(+F,+H,?B)` states that formula `F` has truth value `B`, usually `true` or `false`, at history `H`.

Within the architecture, the projector is used in two places. First, predicate `eval/3` is used to define `trans/4` and `final/2`, as the legal evolutions of high-level programs may often depend on what things are believed true or false. Furthermore, as seen above, the temporal projector provides a number of auxiliary tools used by the top-level loop for bookkeeping tasks. For instance, the top-level cycle is agnostic on how sensing results are incorporated into the belief structure of the agent; this is handled by the `handle_sensing/4` predicate defined in the projector. Similarly, the projector may provide progression facilities by implementing the predicate `handle_rolling/2`.

We illustrate the projector module by briefly describing the one used for modeling the Wumpus World domain [92]. This projector is an extension of the classical formula evaluator used for Golog in [62, 24], so as to handle some limited forms of incomplete knowledge. To that end, the evaluator deals with the so-called *possible values* that (functional) fluents may take at a certain history. We say that a fluent is *known* at h only when it has exactly one possible value at h. For a detailed description and semantics of this type of knowledge-based theories we refer to [100, 59].

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 a occurs and 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 this happens and 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 of 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), \+(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.

Finally, the interface of the module is defined as follows:

```
eval(F,H,true) :- kTrue(F,H).
eval(F,H,false) :- kTrue(neg(F),H).
```

The Environment Manager and the Device Managers

Because the architecture is meant to be used with concrete agent/robotic platforms, as well as with software/simulation environments, the online execution of IndiGolog programs must be linked with the external world. To that end, the *environment manager* (EM) provides a complete interface with all the external devices, platforms, and real-world environments that the application needs to interact with.

In turn, each external device or platform that is expected to interact with the application (e.g., a robot, a software module, or even a user interface) is assumed to have a corresponding *device manager*, a piece of software that is able to talk to the actual device, instruct it to execute actions, as well as gather information and events from it. The device manager understands the "hardware" of the corresponding device and provides a high-level interface to the EM. For example, the device manager for the Wumpus World application is the code responsible for "simulating" an actual Wumpus World environment. It provides an interface for the execution of actions (e.g., `moveFwd`, `smell`, etc.), the retrieval of sensing outcomes for action `smell`, and the detection of occurrences of exogenous events (e.g., `scream`). In our case, the device is also in charge of depicting the world configuration in a Java applet.

Because actual devices are independent of the IndiGolog application and may be in remote locations, device managers are meant to run in different processes and, possibly, on different machines; they communicate then with the EM via TCP/IP sockets. The EM, in contrast, is part of the IndiGolog agent architecture and is tightly coupled with the main cycle. Still, since the EM needs to be open to the external world regardless of any computation happening in the main cycle, the EM and the main cycle run in different (but interacting) threads, though in the same process and Prolog run-time engine.[13]

So, in a nutshell, the EM is responsible of executing actions in the real world and gathering information from it in the form of sensing outcome and exogenous events by communicating with the different device managers. More concretely, given a domain high-level action (e.g., $moveFwd(2m)$), the EM is in charge of: *(i)* deciding which actual "device" should execute the action; *(ii)* ordering its execution by the device via its corresponding device manager; and finally *(iii)* collecting the corresponding sensing outcome. To realize the execution of actions, the EM provides an implementation of `exec/2` to the top-level main cycle: `exec(+A,-S)` orders the execution of action `A`, returning `S` as its sensing outcome.

Besides the execution of actions, the EM continuously listens to the external devices, that is to their managers, for the occurrence of exogenous events. When a device manager reports the occurrence of one or more exogenous actions in its device (e.g., the robot bumped into an object), the EM stores these events in the Prolog database so that the main cycle can later assimilate them all. Moreover, if the main cycle is currently reasoning about a possible program transition (i.e., it is trying to prove a `trans/4` or `final/2` goal), the EM raises an exception named "`exog`" in the main cycle thread. As already mentioned, this will cause the main cycle to abort its reasoning efforts, re-start its loop, and assimilate the pending events.

The Domain Application

From the user perspective, probably the most relevant aspect of the architecture is the specification of the domain application. Any domain application must provide:

1. An *axiomatization of the dynamics of the world.* The exact form of such an axiomatization would depend on the temporal projector used.
2. One or more *high-level agent programs* that specify the different agent behaviors available. In general, these will be IndiGolog programs, but they could be other types of programs under different implementations of `trans/4` and `final/2`.
3. All the necessary *execution information* to *run* the application in the external world. This amounts to specifying which external devices the application relies on (e.g., the device manager for the ER1 robot), and how high-level actions are actually executed on these devices (that is, by which device each high-level action is to be executed). Information on how to translate high-level symbolic actions and sensing results into the device managers' low-level representations, and vice-versa, could also be provided.

[13] SWI-Prolog provides a clean and efficient way of programming multi-threaded Prolog applications.

We illustrate the modeling of an application domain using our running example for the Wumpus domain (we only give a partial specification for the sake of brevity):

```
fluent(locAgent).
fluent(isGold(L)) :- loc(L).
init(locAgent,cell(1,1)).
init(hasArrow,true).
init(locWumpus,L):- loc(L), not L=cell(1,1).

action(pickGold).
poss(pickGold, isGold(locAgent)=true).

causes(moveFwd, locAgent, Y, and(dirAgent=up, up(locAgent,Y))).
causes(moveFwd, locWumpus, Y, or(Y=locWumpus, adj(locWumpus,Y)).

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

The first block defines two (functional) fluents: `locAgent` stands for the current location of the agent; `isGold(L)` states whether location `L` is known to have gold. Initially, the agent is in location `cell(1,1)` and is holding an arrow. More interestingly, the Wumpus is believed to be somewhere in the grid but not in `cell(1,1)`. The second block defines the action of picking up gold, which possible only if the agent believes that there is gold its current location The two clauses shown for `causes/4` state possible ways fluents `locAgent` and `locWumpus` may change when the agent moves forward. First, if the agent is aiming north, then the new location of the agent is updated accordingly. Second, whenever the agent moves, the Wumpus will either stay still or move to an adjacent cell. Observe that even if at some point the agent knows exactly where the Wumpus is located (that is, there is only one possible value for fluent `locWumpus`), after moving forward the agent considers several possible values for the location of the Wumpus. The remaining clauses specify how sensing actions affect the possible values of the relevant fluents. Fluent `locWumpus` is sensed by the `smell` action: if there is no stench (i.e., the sensing result is 0) then each of the agent's adjacent locations is not a possible value for fluent `locWumpus`, otherwise the opposite holds. Fluent `isGoldL` is sensed by the `senseGold` action which settles the value of the fluent depending on the sensing result.

Available Tools and Documentation

The platform distribution includes documentation and examples that, though simple, have allowed new users to learn how to effectively develop new applications. Currently, there are no tools developed specifically for the platform.

For debugging, tracing facilities are provided; Prolog facilities can also be used. This is an area where more work is needed.

Standards Compliance, Interoperability and Portability

There has been some work on interfacing IndiGolog with commonly used multiagent platforms and supporting the use of standard agent communication languages. This can support the development of multiagent systems that incorporate planning and reasoning agents implemented in IndiGolog. The IG-OAAlib library [49] supports the inclusion of IndiGolog agents in systems running under SRI's Open-Agent Architecture (OAA) [67]. Another library, IG-JADE-PKSlib [69, 68] supports the inclusion of IndiGolog agents in systems running under JADE [8], which is FIPA-compliant and more scalable. This library allows IndiGolog agents to use the FIPA agent communication language and run standard agent interaction protocols (e.g. contract net).

Other Features of the Platform

Our platform is an advanced stable prototype and is currently hosted as an open source project at SourceForge (`http://sourceforge.net/projects/indigolog/`). It is designed in a modular way and is easily extensible, though this requires expertise in Prolog.

No detailed analysis regarding the number of agents that could be run efficiently or the number of messages that could be handled has been performed so far. For use in robotic architectures or workflow management, performance has not been a problem.

4 Applications Supported by the Language and/or the Platform

Among some of the applications built using the "high level program execution approach", we can mention an automated banking agent that involved a 40-page Golog program [55, 83]. This is an example of high-level specification that would have been completely infeasible formulated as a planning problem.

A number of cognitive robotic systems have been implemented on a variety of robotic platforms, using Golog-family languages. For a sampling of these systems, see [57, 34, 18, 35]. Perhaps the most impressive demonstration to date was that of the museum tour-guide robot reported in [16]. Borzenko et al. [13] have used IndiGolog to develop a high-level controller for a vision-based pose estimation system. They have also developed an IndiGolog library for knowledge-based control of vision systems called INVICON [12].

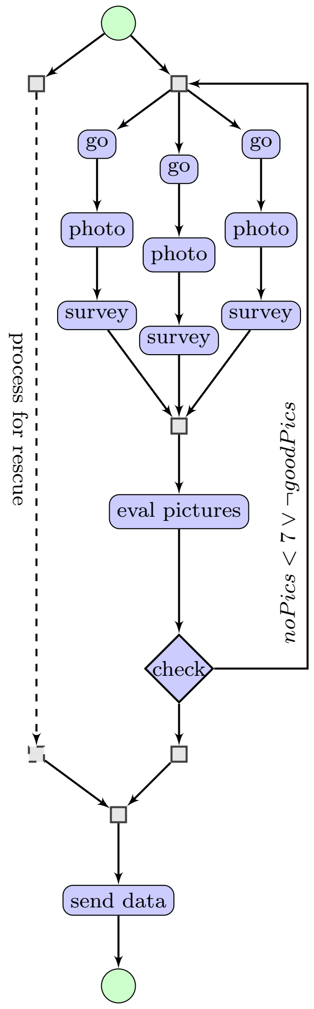
McIlraith and Son [72] have adapted ConGolog to obtain a framework for performing web service composition and customization, an approach that has been very influential. Martinez and Lespérance [69, 68, 70] have developed a library and toolkit that combines IndiGolog, the JADE multiagent platform [8], and the PKS planner [73] for performing web service composition. As well, [38] used a version of ConGolog to support the modeling and analysis of trust in social networks. To get a better idea of how IndiGolog can be used in applications, let us briefly discuss some work using IndiGolog in the area of pervasive computing.

4.1 Using IndiGolog to Coordinate Mobile Actors in Pervasive Computing Scenarios

In [51, 52], de Leoni *et al.* use the IndiGolog platform described in Section 3 to build a process management system (PMS) that coordinates mobile actors in pervasive computing scenarios. PMSs ([63, 1]) are widely used for the management of business processes that have a clear and well-defined structure. In de Leoni et al.'s work, the authors argue that PMSs can also be used in mobile and highly dynamic situations to coordinate, for instance, operators, devices, robots, or sensors. To that end, they show how to realize PMSs in IndiGolog, and how to operationalize the framework proposed in [53] for automatically adapting a process when a gap is sensed between the internal world representation (i.e., the virtual reality) and the actual external reality (i.e., the physical reality).

As an example, consider one of the scenarios investigated in [51, 52, 53]. This scenario concerns an emergency response operation involving various activities that may need to be adapted on-the-fly to react to unexpected exogenous events that could arise during the operation. Figure 2 depicts an Activity Diagram of a process consisting of two concurrent branches; the final task is *send data* and can only be executed after the branches have successfully completed. The left branch, abstracted out from the diagram, is built from several concurrent processes involving *rescue*, *evacuation*, and *census* tasks. The right branch begins with the concurrent execution of three sequences of tasks: *go*, *photo*, and *survey*. When all survey tasks have been completed, the task *evaluate pictures* is executed. Then, a condition is evaluated on the resulting state at a decision point (i.e., whether the pictures taken are of sufficient quality). If the condition holds, the right branch is considered finished; otherwise, the whole branch should be repeated.

When using IndiGolog for process management, *tasks* are taken to be predefined sequences of actions and *processes* to be IndiGolog programs. The objective of the PMS is to carry out the specified processes by assigning tasks to actors, monitoring the progress of the overall process, and adapting its execution when required. Thus, after each action, the PMS may need to



```
proc(pms,
 prioritized_interrupts(
  [interrupt(exogEvent, monitor),
   interrupt(true, process),
   interrupt(neg(finished), wait)]
)).

proc(process,
[conc(processRescue,
  while(or(noPhotos<7,neg(goodPics)),
   [conc(
        [mTasks(
            [witem(go,id19,loc(6,6)),
             witem(photo,id20,loc(6,6)),
             witem(survey,id21,loc(6,6))]),
         mTasks(
            [witem(go,id19,loc(6,6)),
             witem(photo,id20,loc(6,6)),
             witem(survey,id21,loc(6,6))]),
         mTasks(
            [witem(go,id19,loc(6,6)),
             witem(photo,id20,loc(6,6)),
             witem(survey,id21,loc(6,6))]),
        ]
    ),
    mTasks([witem(evalPics,id28,input)])
   ]) % end of while
 ), % end concurrent subprocesses
 mTasks([witem(sendData,id29,input)])
]).

proc(mTasks(LTasks),
  pi(actr,
    [?(and(Idle(actr),Capable(actr,LTasks))),
     manageListTasks(LTasks,actr)]
)).

proc(manageSingleTask(T,D,I,actr),
  [assign(T,D,actr), start(T,D,I,actr),
    stop(T,D,actr), release(T,D,actr)]
).
```

Fig. 2 An example of process management with IndiGolog.

align the internal world representation with the actual external reality. In Figure 2, parts of the IndiGolog program implementing the PMS for the emergency response example are shown ([52]). The main procedure, called pms, involves three interrupts running at different priorities. The first highest priority interrupt fires when an exogenous event has happened (i.e., condition exogEvent is true). In such a case, the monitor procedure is executed, evaluating whether or not adaptation is required (see below).

If no exogenous event has occurred (or the ones that occurred were expected), then the second interrupt triggers and execution of the actual emergency response process is attempted. Procedure process, also shown in the figure, encodes the Activity Diagram of the example process. It relies, in turn, on procedure mTasks(LTasks), where LTasks is a sequence of elements witem(T,I,D), each one representing a task T, with identifier I, and input data D that needs to be performed. This procedure is meant to carry out all tasks in the list by assigning them to a *single* actor that can perform all of them.

Of course, to assign tasks to an actor, the PMS needs to reason about the available actors, their current state (e.g., their location), and their capabilities, as not every actor is capable of performing a task. In fact, before assigning the first task in any task list, a *pick* operation is done to choose an actor actr that is idle (i.e., fluent Idle(actr) holds), and able to execute the whole task list (we leave out the definition of Capable(actr,LTasks)).

Once a suitable actor has been chosen, procedure manageSingleTask(T, I,D) will be called with each task T in the list (with identifier I and input data D). This procedure will first execute assign(T,D,actr), which, among other things, makes fluent Idle(actr) false. The actor is then instructed to start working on the task when the PMS executes the action start(T,D,I,actr), which also provides the required information to the actor. When an actor finishes executing an assigned task, it alerts the PMS via exogenous action finishedTask(T,actr); the PMS notes the completion of the task by performing stop(T,D,actr) and releases the actor by executing the action release(T,D,actr), after which fluent Idle(actr) becomes true.

It is worth mentioning that, if the process being carried out cannot execute further, for instance, because it is waiting for actors to complete their current tasks, the lowest priority interrupt fires and the PMS just waits.

The execution of the process being carried out by the PMS can be interrupted by the monitor module when a misalignment between the expected reality and the actual reality is discovered. In this case, the monitor *adapts* the (current) process to deal with the discrepancy. To do this, the monitor procedure uses the IndiGolog lookahead operator Σ to search for a plan that would bring the actual reality back into alignment with the expected reality. To that end, the PMS keeps a "copy" of the expected value of each relevant fluent so that when an exogenous action is sensed, it can check whether the action has altered the value of some relevant fluent. If so, the monitor looks for a plan that would bring all fluents to their expected values using a program along the lines of $\Sigma([(\pi a.a)^*; \mathit{ExpectedState}?])$. It is easily noted that this kind of adaptation amounts to solving a classical planning problem, and hence, that a state-of-the-art automated planner could be used to perform the required search. In many cases though, a domain expert would be able to provide information on how the *adaptation* should be performed, thus reducing the complexity of the planning task. For instance, when the mismatch involves a team of mobile actors becoming *disconnected* (e.g., because some actor moved too far away), then the whole process can be adapted by running a search program along the lines of $\Sigma([\pi\, actr, loc.Idle(actr)?; moveTo(actr, loc)]^*; TeamConnected?)$, which tries to relocate idle actors so that the whole team is re-connected (the actual program used would in fact implement a better search strategy). IndiGolog is well suited for realizing this kind of domain-specific planning and execution monitoring.

5 Final Remarks

IndiGolog is a rich programming language for developing autonomous agents. Agents programmed in IndiGolog have a situation calculus action theory that they use to model their domain and its dynamics. The theory is used to automatically update their beliefs after actions are performed or events occur. This supports the use of complex domain models and helps avoid the errors that typically occur when such models are manually updated. Moreover it can be used for performing planning/lookahead. The language supports "high level program execution", where the programmer provides a sketchy nondeterministic program and the system searches for a way to execute it. This is usually much less computationally demanding than planning, as the sketchy program constrains the search. As well, programs are executed online and the agent can acquire information at execution time by performing sensing actions or by observing exogenous actions. The language supports concurrent programming, and reactive behaviors can easily be programmed. The language has a classical predicate logic semantics specified through a transition system account defined on top of the situation calculus. One can make statements about offline executions of programs within the logical language and reason about properties of programs in the logic. Online executions of programs are formalized metatheoretically in terms of entailment in the situation calculus theory.

Compared to the mainstream BDI agent programming languages, IndiGolog seems to have several advantages: support for planning/lookahead, automatic belief update, built-in reasoning capabilities, and clean logical semantics. The downside is that these reasoning capabilities can slow the agent's response to events. But with suitable programming of control knowledge, adequate responsiveness can usually be achieved.

Perhaps one weakness of IndiGolog in comparison to BDI agent programming languages is that in the former, plans/procedures are not associated with goals/events; there is no "goal directed invocation". This can make it harder to organize the agent's plan library and to find alternative plans to achieve a goal when a selected plan fails.

There has only been limited work on relating "Golog-like" high-level programming languages and BDI agent programming languages. Hindriks et al. [43, 42] show that ConGolog can be bisimulated by the agent language 3APL under some conditions, which include the agent having complete knowledge; ConGolog's lookahead search mechanism is also ignored as are sensing actions and exogenous events. Also related is the work of Gabaldon [37] on encoding Hierarchical Task Network (HTN) libraries in ConGolog. Much work remains to be done in order to better understand how our approach relates to the BDI approach and others. It would be very interesting to develop an agent programming framework that combines the best features of the IndiGolog and BDI approaches.

More work is necessary to improve the effectiveness of the IndiGolog platform as a programming tool. The platform currently provides little built-in support for programming multiagent systems, interfacing with other agent platforms, or using standard communication languages. But this can be circumvented by using a library like IG-JADE-PKSlib [69, 68], which supports the inclusion of IndiGolog agents in systems running under JADE [8] and allows IndiGolog agents to use the FIPA agent communication language and run standard agent interaction protocols. An integrated development environment with good monitoring and debugging facilities would also be highly desirable. Work is also required on facilities to support large-scale agent programming, e.g. the use of modules.

Another limitation of our platform is that it uses a simple, relatively inefficient planning/lookahead mechanism implemented in Prolog. But it should be possible to address this by doing planning with Golog-style task specifications using state-of-the-art planners. Some authors have addressed this problem. [5] develops an approach for compiling Golog-like task specifications together with the associated domain definition into a PDDL 2.1 planning problem that can be solved by any PDDL 2.1 compliant planner. [4] describes techniques for compiling Golog programs that include sensing actions into domain descriptions that can be handled by operator-based planners. [36] shows how a ConGolog task specification involving concurrent processes together with the associated domain definition can be compiled into an ordinary situation calculus basic action theory; moreover it show how the specification can be complied into PDDL under some assumptions. Classen et al. [21, 20] have also used the IndiGolog architecture to integrate automated planning systems into cognitive agents and tested the performance of the integrated system in typical planning domain benchmarks [40].

Acknowledgements The late Ray Reiter was a major contributor to Golog and to our approach to agent programming. Gerhard Lakemeyer helped us with the sections on reasoning about action and robotics applications. Stavros Vassos helped develop the Wumpus World application. Massimiliano de Leoni helped us with the section on applications involving mobile actors in pervasive computing scenarios. We thank everyone who contributed to developing the approach and platform over the years.

Appendix (Language Summary)

Here we discuss how our language/platform addresses the comparison criteria identified by the book editors.

1(a) The language supports agents with complex beliefs about their environment and its dynamics, specified as a situation calculus action theory. The beliefs are automatically updated based on the model when actions are performed or events occur. The agent can perform sensing actions to acquire additional knowledge. It can perform means-ends reasoning to generate a plan that will achieve a goal or find an execution of a "sketchy" nondeterministic program. Specifying reactive behaviors is also supported. However, there is no built-in support for declarative goals, or for reasoning about other agents and their mental states.

1(b) The language does not provide built-in support for speech act based communication. However, communication in FIPA ACL and FIPA coordination protocols (e.g. contract net), as well as interfacing with the JADE [8] multiagent platform are supported by the IG-JADE-PKSlib library [69, 68]. The framework been extended to incorporate a rich model of multiagent beliefs and goals and speech acts based communication in [95, 94]; but the resulting formalism is no longer a programming language but a specification language that supports verification of properties.

1(c) No specific built-in support for mobile agents is available so far.

1(d) The language is easy to understand and learn, as it combines a classical Algol-like imperative language for specifying behavior with a well known action description language for specifying the application domain dynamics. The whole language has a classical logic semantics.

1(e) The language has a very solid formal foundation. The semantics of programs is specified through a transition system account defined on top of the situation calculus (the latter is used to specify the application domain, primitive actions and state-dependent predicates). Thus the language is fully formalized in classical predicate logic. One can make statements about offline executions of programs within the logical language, and one can reason about properties of programs in the logic. Online executions of programs are formalized metatheoretically in terms of entailment in the situation calculus theory.

1(f) The language is very rich and expressive. Complex domain models can be specified declaratively and the agent's beliefs are automatically updated. Complex tests about the state of the world can be evaluated. Behavior can be fully scripted, or synthesized through planning, with the program constraining the search. A rich set of procedural constructs is provided, including concurrent programming facilities. Reactivity and online sensing are also supported.

1(g) The declarative language definition supports the easy addition of new programming constructs. The underlying situation calculus framework supports many extensions in the way change is modeled, e.g. continuous change, stochastic effects, etc. The language has been extended numerous times.

1(h) Given its strong formal foundations, the language is highly suited for formal verification. The CASLve verification environment [95, 94], which is based on the PVS theorem proving/verification system, has been developed to support verification of programs in an extended version of ConGolog.

1(i) The language supports procedural abstraction, but not modules. However, very complex agents can be decomposed into simpler agents that cooperate. The agent's beliefs are automatically updated based on a declarative action theory, which supports the use of complex domain models, and helps avoid the errors that typically occur when such models are manually updated.

1(j) Our platform is implemented in SWI-Prolog, which provides flexible mechanisms for interfacing with other programming languages such as Java or C, and for socket communication.

2(a).i The platform provides documentation and examples that, though simple, have allowed new users to learn how to effectively develop new applications.

2(a).ii The current implementation of the platform requires SWI-Prolog, a sophisticated Prolog implementation which is actively supported and available free for many architectures and operating systems (including MS-Windows, Linux and MacOS X).

2(b) The basic language and its current platform do not per se adhere or conform to any standards. However, a library, IG-JADE-PKSlib [69, 68], has been developed to support communication in FIPA ACL and FIPA coordination protocols (e.g. contract net), as well as interfacing with the JADE [8] multiagent platform.

2(c) The platform is designed in a modular way and is easily extensible, though this requires expertise in Prolog. It is currently hosted as an open source project at SourceForge (`http://sourceforge.net/projects/indigolog/`).

2(d) Currently, there are no CASE tools developed specifically for the platform. For debugging, tracing facilities are provided; Prolog facilities can also be used.

2(e) The platform is integrated with Prolog (more specifically SWI-Prolog) and all the facilities it provides can be used (e.g. socket communication, calling C or Java procedures). The IG-OAAlib library [49] supports the inclusion of IndiGolog agents in systems running under SRI's Open-Agent Architecture (OAA) [67]. As mentioned earlier, another library, IG-JADE-PKSlib

[69, 68] supports the inclusion of IndiGolog agents in systems running under JADE [8].

2(f) The platform is implemented in Prolog and requires SWI-Prolog (`http://www.swi-prolog.org/`).

2(g).i No detailed analysis regarding the number of agents that could be run efficiently or the number of messages that could be handled has been performed so far. For use in robotic architectures or workflow management, performance has not been a problem.

2(g).ii The current state of the implementation is as an advanced stable prototype that is available through open source distribution.

2(h).i The language itself does not provide specific facilities for multi-agent programming (though it and the underlying theory are expressive enough to allow the design of multi-agent systems). It is intended primarily for the implementation of individual autonomous agents. Multi-agent programming (including open systems) is accommodated through the interfaces with the JADE and OAA platforms.

2(h).ii The language provides a centralized control architecture.

2(h).iii As already mentioned, the IG-JADE-PKSlib library [69, 68] allows IndiGolog agents to be integrated in systems running under the JADE [8] multi-agent platform; it supports the development of IndiGolog agents that use FIPA ACL communication and coordination protocols. Another library [49] supports including IndiGolog agents in systems running under the OAA platform [67].

3(a) So far, the language and platform have been used to program high-level controllers for several real robotic platforms (as part of a larger control architecture). Moreover, the language (or variants), and the platform, have been used as part of larger systems to develop advanced applications, for instance the museum guide robot of [16], the process/workflow management system for pervasive computing applications of [52], the automated web service composition/customization systems of [70, 72], etc.

3(b) The language is not targeted at any particular application domain. However, it is primarily intended for developing complex autonomous agents that do reasoning and planning. It provides good support for interfacing with robotic control architectures/platforms.

References

1. van der Aalst, W., van Hee, K.: Workflow Management. Models, Methods, and Systems. MIT Press (2004)
2. Amir, E., Russell, S.: Logical filtering. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 75–82. Acapulco, Mexico (2003)
3. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence **22**, 5–27 (1998)

4. Baier, J., McIlraith, S.: On planning with programs that sense. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR), pp. 492–502. Lake District, UK (2006)
5. Baier, J.A., Fritz, C., McIlraith, S.A.: Exploiting procedural domain control knowledge in state-of-the-art planners. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), pp. 26–33. Providence, Rhode Island (2007)
6. Baral, C., Son, T.C.: Relating theories of actions and reactive control. Electronic Transactions of Artificial Intelligence **2**(3-4), 211–271 (1998)
7. Belecheanu, R.A., Munroe, S., Luck, M., Payne, T., Miller, T., McBurney, P., Pechoucek, M.: Commercial applications of agents: Lessons, experiences and challenges. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 1549–1555. ACM Press (2006)
8. Bellifemine, F., Claire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley (2007)
9. Benfield, S.S., Hendrickson, J., Galanti, D.: Making a strong business case for multiagent technology. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 10–15. ACM Press, New York, NY, USA (2006)
10. Bertoli, P., Cimatti, A., Roveri, M., Traverso, P.: Planning in nondeterministic domains under partial observability via symbolic model checking. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 473–478 (2001)
11. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason. Wiley Series in Agent Technology. Wiley (2007). Series in Agent Technology
12. Borzenko, O., Lespérance, Y., Jenkin., M.: INVICON: a toolkit for knowledge-based control of vision systems. In: Proc. of the 4th Canadian Conference on Computer and Robot Vision (CRV'07), pp. 387–394. Montréal, QC, Canada (2007)
13. Borzenko, O., Xu, W., Obsniuk, M., Chopra, A., Jasiobedzki, P., Jenkin, M., Lespérance, Y.: Lights and camera: Intelligently controlled multi-channel pose estimation system. In: Proc. of the IEEE International Conference on Vision Systems (ICVS'06). New York, NY, USA (2006). Paper 42 (8 pages)
14. Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S.: Decision-theoretic, high-level agent programming in the situation calculus. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 355–362. AAAI Press, Menlo Park, CA (2000)
15. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press (1987)
16. Burgard, W., Cremers, A., Fox, D., Hähnel, D., Lakemeyer, G., Schulz, D., Steiner, W., Thrun, S.: Experiences with an interactive museum tour-guide robot. Artificial Intelligence **114**(1–2), 3–55 (1999)
17. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK intelligent agents: Components for intelligent agents in Java. AgentLink Newsletter **2** (1999). Agent Oriented Software Pty. Ltd.
18. Carbone, A., Finzi, A., Orlandini, A., Pirri, F., Ugazio, G.: Augmenting situation awareness via model-based control in rescue robots. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3699–3705. Edmonton, AB, Canada (2005)
19. Castro, J., Kolp, M., Mylopoulos, J.: Towards requirements-driven information systems engineering: The tropos project. Information Systems **27**(6), 365–389 (2002)
20. Classen, J., Engelmann, V., Lakemeyer, G., Röger, G.: Integrating Golog and planning: An empirical evaluation. In: Non-Monotonic Reasoning Workshop. Sydney, Australia (2008)

21. Classen, J., Eyerich, P., Lakemeyer, G., Nebel, B.: Towards an integration of planning and Golog. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1846–1851. Hyderabad, India (2007)
22. Cohen, P.R., Levesque, H.J.: Intention is choice with commitment. Artificial Intelligence Journal **42**, 213–261 (1990)
23. Dastani, M., de Boer, F.S., Dignum, F., van der Hoek, W., Kroese, M., Meyer, J.J.: Programming the deliberation cycle of cognitive robots. In: Proceedings of the International Cognitive Robotics Workshop (COGROBO). Edmonton, Canada (2002)
24. De Giacomo, G., Lespérance, Y., Levesque, H.J.: ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence Journal **121**(1–2), 109–169 (2000)
25. De Giacomo, G., Lespérance, Y., Levesque, H.J., Sardina, S.: On deliberation under incomplete information and the inadequacy of entailment and consistency-based formalizations. In: Proceedings of the Programming Multiagent Systems Languages, Frameworks, Techniques and Tools workshop (PROMAS). Melbourne, Australia (2003)
26. De Giacomo, G., Levesque, H.J.: An incremental interpreter for high-level programs with sensing. In: H.J. Levesque, F. Pirri (eds.) Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter, pp. 86–102. Springer, Berlin (1999)
27. De Giacomo, G., Levesque, H.J.: Projection using regression and sensors. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 160–165. Stockholm, Sweden (1999)
28. De Giacomo, G., Levesque, H.J., Sardina, S.: Incremental execution of guarded theories. ACM Transactions on Computational Logic (TOCL) **2**(4), 495–525 (2001)
29. De Giacomo, G., Mancini, T.: Scaling up reasoning about actions using relational database technology. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 245–256 (2004)
30. De Giacomo, G., Reiter, R., Soutchanski, M.: Execution monitoring of high-level robot programs. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR), pp. 453–465 (1998)
31. Dennett, D.: The Intentional Stance. The MIT Press (1987)
32. Doherty, P.: Advanced research with autonomous unmanned aerial vehicles. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR) (2004). Extended abstract for Plenary Talk
33. Erol, K., Hendler, J.A., Nau, D.S.: HTN planning: Complexity and expressivity. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 1123–1228 (1994)
34. Ferrein, A., Fritz, C., Lakemeyer, G.: On-line decision-theoretic Golog for unpredictable domains. In: Proc. of 27th German Conference on Artificial Intelligence, pp. 322–336. Ulm, Germany, UK (2004)
35. Finzi, A., Pirri, F., Pirrone, M., Romano, M.: Autonomous mobile manipulators managing perception and failures. In: Proceedings of the Annual Conference on Autonomous Agents (AGENTS), pp. 196–201. Montréal, QC, Canada (2001)
36. Fritz, C., Baier, J.A., McIlraith, S.A.: ConGolog, Sin Trans: Compiling ConGolog into basic action theories for planning and beyond. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR), pp. 600–610. Sydney, Australia (2008)
37. Gabaldon, A.: Programming hierarchical task networks in the situation calculus. In: AIPS'02 Workshop on On-line Planning and Scheduling. Toulouse, France (2002)
38. Gans, G., Jarke, M., Kethers, S., Lakemeyer, G., Ellrich, L., Funken, C., Meister, M.: Requirements modeling for organization networks: A (dis-)trust-based

approach. In: Proc. of IEEE Int. Requirements Engineering Conf., pp. 154–163 (2001)

39. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 677–682. Seattle, USA (1987)
40. Gerevini, A., Bonet, B., Givan, B. (eds.): Booklet of 4th International Planning Competition. Lake District, UK (2006). URL `http://www.ldc.usb.ve/~bonet/ipc5/`
41. Grosskreutz, H., Lakemeyer, G.: ccGolog: an action language with continuous change. Logic Journal of the IGPL (2003)
42. Hindriks, K., Lespérance, Y., Levesque, H.: An embedding of ConGolog in 3APL. Tech. Rep. UU-CS-2000-13, Department of Computer Science, University Utrecht (2000)
43. Hindriks, K., Lespérance, Y., Levesque, H.J.: A formal embedding of ConGolog in 3APL. In: Proceedings of the European Conference in Artificial Intelligence (ECAI), pp. 558–562. Berlin, Germany (2000)
44. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.C.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems **2**, 357–401 (1999)
45. Huber, M.J.: JAM: A BDI-theoretic mobile agent architecture. In: Proceedings of the Annual Conference on Autonomous Agents (AGENTS), pp. 236–243. ACM Press, New York, NY, USA (1999)
46. Kowalski, R.A.: Using meta-logic to reconcile reactive with rational agents. In: K.R. Apt, F. Turini (eds.) Meta-Logics and Logic Programming, pp. 227–242. The MIT Press (1995)
47. Lakemeyer, G.: Relevance from an epistemic perspective. Artificial Intelligence **97**(1–2), 137–167 (1997)
48. Lakemeyer, G.: On sensing and off-line interpreting in Golog. In: H. Levesque, F. Pirri (eds.) Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter, pp. 173–187. Springer, Berlin (1999)
49. Lapouchnian, A., Lespérance, Y.: Interfacing IndiGolog and OAA — a toolkit for advanced multiagent applications. Applied Artificial Intelligence **16**(9-10), 813–829 (2002)
50. Lapouchnian, A., Lespérance, Y.: Modeling mental states in agent-oriented requirements engineering. In: Proc. of the 18th Conference on Advanced Information Systems Engineering (CAiSE'06), pp. 480–494. Luxembourg (2006)
51. de Leoni, M.: Adaptive Process Management in Pervasive and Highly Dynamic Scenarios. Ph.D. thesis, SAPIENZA - University of Rome (2009)
52. de Leoni, M., Marrella, A., Mecella, M., Valentini, S., Sardina, S.: Coordinating mobile actors in pervasive and mobile scenarios: An AI-based approach. In: Proceedings of the 17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE); CoMA sub-workshop. IEEE Computer Society, Rome, Italy (2008)
53. de Leoni, M., Mecella, M., De Giacomo, G.: Highly dynamic adaptation in process management systems through execution monitoring. In: Proceedings of the Fifth International Conference on Business Process Management (BPM'07), *Lecture Notes in Computer Science*, vol. 4714, pp. 182–197. Springer, Brisbane, Australia (2007)
54. Lespérance, Y., De Giacomo, G., Ozgovde, A.N.: A model of contingent planning for agent programming languages. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 477–484. Estoril, Portugal (2008)
55. Lespérance, Y., Levesque, H.J., Ruman, S.J.: An experiment in using Golog to build a personal banking ass istant. In: L. Cavedon, A. Rao, W. Wobcke (eds.) Intelligent Agent Systems: Theoretical and Practical Issues (Based on a

Workshop Held at PRICAI '96 Cairns, Austral ia, August 1996),, *LNAI*, vol. 1209, pp. 27–43. Springer-Verlag (1997)

56. Lespérance, Y., Ng, H.K.: Integrating planning into reactive high-level robot programs. In: Proceedings of the International Cognitive Robotics Workshop (COGROBO), pp. 49–54. Berlin, Germany (2000)
57. Lespérance, Y., Tam, K., Jenkin, M.: Reactivity in a logic-based robot programming framework. In: N. Jennings, Y. Lespérance (eds.) Intelligent Agents VI — Agent Theories, Architectures, and Languages, 6th International Workshop, ATAL'99, Proceedings, *LNAI*, vol. 1757, pp. 173–187. Springer-Verlag, Berlin (2000)
58. Levesque, H.J.: What is planning in the presence of sensing? In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 1139–1146. American Association for Artificial Intelligence, Portland, Oregon (1996)
59. Levesque, H.J.: Planning with loops. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 509–515 (2005)
60. Levesque, H.J., Pagnucco, M.: LeGolog: Inexpensive experiments in cognitive robotics. In: Proceedings of the International Cognitive Robotics Workshop (COGROBO), pp. 104–109. Berlin, Germany (2000)
61. Levesque, H.J., Reiter, R.: High-level robotic control: Beyond planning. A position paper. In: AAAI 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap (1998)
62. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31**, 59–84 (1997)
63. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR (1999)
64. Lin, F., Reiter, R.: Forget it! In: Proceedings of AAAI Fall Symposium on Relevance. New Orleans, USA (1994)
65. Lin, F., Reiter, R.: How to progress a database. Artificial Intelligence Journal **92**, 131–167 (1997)
66. Ljungberg, M., Lucas, A.: The OASIS air-traffic management system. In: Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI). Seoul, Korea (1992)
67. Martin, D., Cheyer A, J., Moran, D.: The open agent architecture: A framework for building distributed software systems. Applied Artificial Intelligence **13**, 91–128 (1999)
68. Martinez, E.: Web service composition as a planning task: An agent oriented framework. Master's thesis, Department of Computer Science, York University, Toronto, ON, Canada (2005)
69. Martinez, E., Lespérance, Y.: IG-JADE-PKSlib: an agent-based framework for advanced web service composition and provisioning. In: Proc. of the AAMAS 2004 Workshop on Web-services and Agent-based Engineering, pp. 2–10. New York, NY, USA (2004)
70. Martinez, E., Lespérance, Y.: Web service composition as a planning task: Experiments using knowledge-based planning. In: Proc. of the ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services, pp. 62–69. Whistler, BC, Canada (2004)
71. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence **4**, 463–502 (1969)
72. McIlraith, S., Son, T.C.: Adapting Golog for programming the semantic web. In: Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002), pp. 482–493. Toulouse, France (2002)
73. Petrick, R., Bacchus, F.: A knowledge-based approach to planning with incomplete information and sensing. In: Proceedings of the International Conference on AI Planning & Scheduling (AIPS), pp. 212–221 (2002)

74. Pirri, F., Reiter, R.: Some contributions to the metatheory of the situation calculus. Journal of the ACM **46**(3), 261–325 (1999)
75. Plotkin, G.D.: A structural approach to operational semantics. Tech. Rep. DAIMI-FN-19, Computer Science Department, Aarhus University, Denmark (1981)
76. Pollack, M.E.: The uses of plans. Artificial Intelligence Journal **57**(1), 43–68 (1992)
77. Pryor, L., Collins, G.: Planning for contingencies: A decision-based approach. J. of Artificial Intelligence Research **4**, 287–339 (1996)
78. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: W.V. Velde, J.W. Perram (eds.) Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World. (Agents Breaking Away), *Lecture Notes in Computer Science (LNCS)*, vol. 1038, pp. 42–55. Springer-Verlag (1996)
79. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR), pp. 473–484 (1991)
80. Reiter, R.: On closed world data bases. In: Logic and Data Bases, pp. 55–76 (1977)
81. Reiter, R.: The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In: V. Lifschitz (ed.) Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, pp. 359–380. Academic Press, San Diego, CA (1991)
82. Reiter, R.: Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems. The MIT Press (2001)
83. Ruman, S.J.: GOLOG as an agent-programming language: Experiments in developing banking applications. Master's thesis, Department of Computer Science, University of Toronto (1996)
84. Sardina, S.: Local conditional high-level robot programs. In: R. Nieuwenhuis, A. Voronkov (eds.) Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *Lecture Notes in Computer Science (LNCS)*, vol. 2250, pp. 110–124. Springer, La Habana, Cuba (2001)
85. Sardina, S.: IndiGolog: An Integrated Agent Arquitecture: Programmer and User Manual. University of Toronto (2004). URL `http://sourceforge.net/projects/indigolog/`
86. Sardina, S.: Deliberation in agent programming languages. Ph.D. thesis, Department of Computer Science (2005)
87. Sardina, S., De Giacomo, G., Lespérance, Y., Levesque, H.J.: On the semantics of deliberation in IndiGolog – from theory to implementation. Annals of Mathematics and Artificial Intelligence **41**(2–4), 259–299 (2004)
88. Sardina, S., De Giacomo, G., Lespérance, Y., Levesque, H.J.: On the limits of planning over belief states. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR), pp. 463–471. Lake District, UK (2005)
89. Sardina, S., de Silva, L.P., Padgham, L.: Hierarchical planning in BDI agent programming languages: A formal approach. In: H. Nakashima, M.P. Wellman, G. Weiss, P. Stone (eds.) Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 1001–1008. ACM Press, Hakodate, Japan (2006)
90. Sardina, S., Padgham, L.: Goals in the context of BDI plan failure and planning. In: E.H. Durfee, M. Yokoo, M.N. Huhns, O. Shehory (eds.) Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS). ACM Press, Hawaii, USA (2007)
91. Sardina, S., Shapiro, S.: Rational action in agent programs with prioritized goals. In: J.S. Rosenschein, M. Wooldridge, T. Sandholm, M. Yokoo (eds.) Proceedings

of Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 417–424. ACM Press, Melbourne, Australia (2003)
92. Sardina, S., Vassos, S.: The Wumpus World in IndiGolog: A preliminary report. In: L. Morgenstern, M. Pagnucco (eds.) Proceedings of the Workshop on Non-monotonic Reasoning, Action and Change at IJCAI (NRAC-05), pp. 90–95 (2005)
93. Scherl, R.B., Levesque, H.J.: Knowledge, action, and the frame problem. Artificial Intelligence Journal **144**(1–2), 1–39 (2003)
94. Shapiro, S.: Specifying and verifying multiagent systems using the cognitive agents specification language (CASL). Ph.D. thesis, Department of Computer Science, University of Toronto (2005)
95. Shapiro, S., Lespérance, Y., Levesque, H.J.: The cognitive agents specification language and verification environment for multiagent systems. In: C. Castelfranchi, W.L. Johnson (eds.) Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 19–26. ACM Press (2002)
96. Son, T.C., Baral, C.: Formalizing sensing actions — A transition function based approach. Artificial Intelligence **125**(1–2), 19–91 (2001)
97. Soutchanski, M.: An on-line decision-theoretic Golog interpreter. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 19–26. Seattle, WA, USA (2001)
98. Thielscher, M.: The fluent calculus. Tech. Rep. CL-2000-01, Computational Logic Group, Artificial Intelligence Institute, Department of Computer Science, Dresden University of Technology (2000)
99. Thielscher, M.: FLUX: A logic programming method for reasoning agents. Theory and Practice of Logic Programming **5**(4–5), 533–565 (2005). Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules
100. Vassos, S.: A feasible approach to disjunctive knowledge in situation calculus. Master's thesis, Department of Computer Science (2005)
101. Vassos, S., Lakemeyer, G., Levesque, H.: First-order strong progression for local-effect basic action theories. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR), pp. 662–672. Sydney, Australia (2008)
102. Vassos, S., Levesque, H.: Progression of situation calculus action theories with incomplete information. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 2024–2029. Hyderabad, India (2007)
103. Vassos, S., Levesque, H.: On the progression of situation calculus basic action theories: Resolving a 10-year-old conjecture. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 1004–1009. Chicago, Illinois, USA (2008)
104. Wang, X., Lespérance, Y.: Agent-oriented requirements engineering using congolog and i*. In: G. Wagner, K. Karlapalem, Y. Lespérance, E. Yu (eds.) Agent-Oriented Information Systems 2001, Proceedings of the 3rd International Bi-Conference Workshop AOIS-2001, pp. 59–78. iCue Publishing, Berlin, Germany (2001)
105. Wielemaker, J.: An overview of the SWI-Prolog programming environment. In: F. Mesnard, A. Serebenik (eds.) Proceedings of the 13th International Workshop on Logic Programming Environments, pp. 1–16. Katholieke Universiteit Leuven, Heverlee, Belgium (2003). CW 371
106. Yongmei Liu, H.J.L.: Tractable reasoning with incomplete first-)rder knowledge in dynamic systems with context-dependent actions. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 522–527. Edinburgh, UK (2005)

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.