Incremental Execution of Guarded Theories

GIUSEPPE DE GIACOMO
Università di Roma "La Sapienza"
and
HECTOR J. LEVESQUE and SEBASTIAN SARDIÑA
University of Toronto

When it comes to building controllers for robots or agents, high-level programming languages like *Golog* and *ConGolog* offer a useful compromise between planning-based approaches and low-level robot programming. However, two serious problems typically emerge in practical implementations of these languages: how to evaluate tests in a program efficiently enough in an open-world setting, and how to make appropriate nondeterministic choices while avoiding full lookahead. Recent proposals in the literature suggest that one could tackle the first problem by exploiting sensing information, and tackle the second by specifying the amount of lookahead allowed explicitly in the program. In this paper, we combine these two ideas and demonstrate their power by presenting an interpreter, written in Prolog, for a variant of *Golog* that is suitable for efficiently operating in open-world setting by exploiting sensing and bounded lookahead.

Categories and Subject Descriptors: I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Plan execution, formation, and generation*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Logic Programming*

General Terms: Theory, Languages

Additional Key Words and Phrases: Reasoning about actions, situation calculus, agent behavior

1. INTRODUCTION

When it comes to building controllers for robots or agents, high-level programming languages like *Golog* [Levesque et al. 1997] and *ConGolog* [De Giacomo et al. 2000] offer a useful compromise between planning-based approaches and low-level robot programming. By a high-level program, we mean one whose primitive instructions are domain-dependent actions of the robot, whose tests involve domain-dependent

We dedicate this work to Robert Kowalski who, as much as anyone, invented the idea and practice of logic programming. *Golog* and its descendants are part of the same family, though currently somewhat estranged.

Authors' address: G. De Giacomo, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy; email: degiacomo@dis.uniroma1.it; H. J. Levesque and S. Sardiña, Department of Computer Science, University of Toronto, Toronto, Canada M5S 3H5; email: {hector,ssardina}@cs.toronto.edu

fluents affected by these actions, and whose code may contain nondeterministic choice points.

Two serious problems typically emerge in practical implementations of these languages, however: (1) to evaluate tests in a program efficiently enough, a Prolog-style closed-world assumption (CWA) is required, and (2) to make appropriate nondeterministic choices, an execution trajectory of the entire program needs to be calculated offline.

In De Giacomo and Levesque [1999b], *guarded action theories*, which do not require a CWA but still permit efficient evaluation by using sensing information, are introduced; while in De Giacomo and Levesque [1999a], an incremental version of *Golog* is proposed which would allow a programmer to specify which parts of a program should be run offline (using an explicit search operator), and which should be run online (foregoing full lookahead for efficiency purposes).

In this paper, we study an amalgamation of these two ideas and a version of *Golog*, namely *IndiGolog* (incremental deterministic *Golog*), which has all the language features of *ConGolog*, but which also (i) works in an open-world setting with guarded action theories, (ii) allows for programmer control over online and offline execution, and in the online case, allows sensing information to affect subsequent computation. Based on such amalgamation, we propose a simple implementation in Prolog of an *IndiGolog* interpreter that incrementally execute programs w.r.t. to guarded theories. Such an implementation is provably correct under suitable conditions.

The structure of the rest of the paper is as follows. In Section 2, we briefly review the situation calculus and basic action theories. Section 3 is devoted to guarded action theories and Section 4 to *ConGolog* programs. Next, in Section 5, we discuss how those programs can be executed incrementally. In Section 6, we talk about "well-behaved" histories and programs with sufficient information to be executed. In Section 7, we describe our Prolog implementation of an *IndiGolog* interpreter, and we prove its correctness in Section 8. We draw conclusions and discuss future work in Section 9.

2. SITUATION CALCULUS AND BASIC ACTION THEORIES

The *situation calculus* [McCarthy and Hayes 1969] is a second-order language specifically designed for representing dynamically changing worlds in which all changes are the result of named *actions*. The language has a special constant S_0 used to denote the *initial situation* where 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 action a; relations whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate symbols taking a situation term as their last argument;[1] and there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s. For instance, the fluent $Floor(n, s)$ may indicate that the elevator is in the floor n in situation s, and $Poss(up, s)$ says whether it is possible to go one floor up in situation s.

Within this language, we can formulate action theories that describe how the world changes as the result of the available actions. In Reiter [1991] action theories

[1]We only deal with relational fluents in this paper.

of a special form, called *basic action theories*, were introduced. Basic action theories have the following form:

—$\mathcal{D}_0$ is the set of axioms describing the initial situation S_0.

—$\mathcal{D}_{Poss}$ contains one precondition axiom for each action A, characterizing the relation $Poss(A, s)$, of the following form:[2]

$$Poss(A, s) \equiv \psi(s)$$

where $Poss$ does not occur in $\psi(s)$.

—$\mathcal{D}_{SSA}$ is the set of all successor state axioms, one for each fluent F, of the following form:

$$F(\vec{x}, do(a, s)) \equiv \gamma(\vec{x}, a, s)$$

which state under what conditions $F(\vec{x}, do(a, s))$ holds as function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem [Reiter 1991].

—$\mathcal{D}_{una}$ is the set of unique names axioms for actions.

—$\mathcal{D}_{FUN}$ is the set of foundational, domain independent axioms.

Successor state axioms are the key to such theories given that they provide an axiomatization of the effects and noneffects of actions, i.e., they encapsulate the causal laws of the world.

EXAMPLE 2.1. *For example, the successor state axiom*

$$ControllerOn(do(a, s)) \equiv (a = turnOnController \wedge \neg ControllerBroken(s)) \vee (a \neq turnOffControler \wedge a \neq breakController \wedge ControllerOn(s))$$

states that a specific controller is functioning after doing action a if a is turning it on and the controller is not broken, or it was already functioning, and a is not the action of turning it off nor breaking it.

One of the most fundamental tasks concerned with reasoning about actions is the *projection task*: determining whether a fluent does or does not hold after performing a sequence of actions. Projection is clearly a prerequisite to both *planning* and the *high-level program execution task* [Levesque et al. 1997]. Now, observe that in basic action theories successor state axioms can be thought as defining the truth-value of a fluent in a *given situation* in terms of the truth-value of fluents in the *previous situation*. This characteristic has a notable impact on reasoning with basic action theories: it allows us to base projection on a special form of evaluation, *regression* [Reiter 1991], which is a central computational mechanism in AI [Waldinger 1977], plus inference about the initial situation. As a result we have a convenient way of reasoning when we have complete information about the initial situation.

3. GUARDED ACTION THEORIES

Recently, basic action theories have been extended to *guarded action theories* (GATs) in which causal laws may not be complete, and sensing is formalized explicitly in the

[2]From now on, free variables are assumed to be universally quantified from the outside.

theory [De Giacomo and Levesque 1999b]. We assume that a robot has a number of on-board sensors that provide sensing readings at any time. Formally, we introduce a finite number of *sensing functions*, which are unary functions whose only argument is a situation. Syntactically, sensing functions look like functional fluents; however, it is assumed that sensed values, i.e., the values returned by the sensing functions, belong to a *concrete* domain, i.e., a possibly infinite domain such that every element of the domain has a unique term that denotes it. Examples of sensing functions modeling available sensors in a robot might be $thermometer(s)$ returning the readings of an onboard thermometer, $sonar(s)$ returning values from the robot sonar, $depthGauge(s)$ returning the depth measured by an onboard gauge, etc. We observe that with this approach we change the way we look at sensing, moving from the well-known *sensing actions* (Golden et al. [1996], Baral and Son [1997], and Levesque [1996]) to online on-board *sensors* whose data may or may not be applicable.

We define a *sensor-fluent formula* to be a formula of the language that uses at most one situation term, which is a variable, and that this term only appears as the final argument of a fluent or sensor function. We write $\phi(\vec{x}, s)$ when ϕ is a sensor-fluent formula with free variables among the $\vec{x}$ and s, and $\phi(\vec{t}, t_s)$ for the formula that results after the substitution of $\vec{x}$ by the vector of terms $\vec{t}$ and s by the situation term t_s. A *fluent formula* is one that mentions no sensor functions. A *sensor formula* is a sensor-fluent formula that does not mention fluents. In the following we restrict our attention to sensor formulas that are easily evaluable given the values of the sensors.

A guarded action theory is like a basic action theory where the set of successor axioms $\mathcal{D}_{SSA}$ is replaced by two new sets of axioms $\mathcal{D}_{GSSA}$ and $\mathcal{D}_{GSFA}$:

—$\mathcal{D}_{GSSA}$ is the set of *guarded successor state axioms* (GSSAs), i.e., axioms of the following form:

$$\alpha(\vec{x}, a, s) \supset [F(\vec{x}, do(a, s)) \equiv \gamma(\vec{x}, a, s)]$$

where α is a fluent formula called the *guard* of the axiom, F is a relational fluent, and γ is a fluent formula.[3]

—$\mathcal{D}_{GSFA}$ is the set of *guarded sensed fluent axioms* (GSFAs), i.e., axioms of the following form:

$$\beta(\vec{x}, s) \supset [F(\vec{x}, s) \equiv \rho(\vec{x}, s)]$$

where β is a sensor-fluent formula called the *guard* of the axiom, F is a relational fluent, and ρ is a sensor formula.

In a guarded action theory, each fluent has any number of GSSAs and GSFAs. We denote with $GSSA(F)$ and $GSFA(F)$ the sets of GSSAs and GSFAs for fluent F.

Example 3.1. *In an application for an elevator controller, the following axioms are used to reason about the state of the lights at each floor:*

$$ControllerOn(s) \supset (Light(x, do(a, s)) \equiv (a = on(x) \vee Light(x, s) \wedge a \neq off(x)))$$
$$Floor(x, s) \supset (Light(x, s) \equiv sensor\_floor(s) > 50)$$

[3]Observe that α and γ are completely general, except for the fact that their situation argument s is the predecessor of the situation argument $do(a, s)$ of F.

The first axiom is a GSSA that gives a complete characterization of the fluent Light whenever the light controller is working. In such a case, a floor light is on after an action a if a is turning the floor light in question on, or it was already on and a is not the action of switching it off. The second axiom is a GSFA that gives the state of the light at the floor the elevator is on (given by the fluent $Floor(x, s)$) by interpreting the value of an on-board sensor modeled via $sensor\_floor(s)$. If the elevator is on floor x, it is possible to conclude whether the light of floor x is on by checking if the value of the on-board sensor related to the sensing function $sensor\_floor(s)$ is more than 50. Similarly, fluent $Floor(x, s)$ would have its own GSSA and GSFA as well. Notice that GSSA not only can solve the frame problem, but can also compile others restrictions such as $Floor(x, s)$ being functional.

Observe that the general form of guarded action theories allows for ill-formed theories. In this paper, we restrict our attention to guarded action theories that are *consistent* and, moreover, that are *acyclic*.

Definition 3.2 *(Acyclic Guarded Action Theories).* We say $F_1 \prec F_2$ (F_2 depends on F_1) if there exists a GSFA $\{\alpha(\vec{x}, s) \supset [F_2(\vec{x}, s) \equiv \rho(\vec{x}, s)]\}$ in $\mathcal{D}$ where F_1 is mentioned in the guard $\alpha(\vec{x}, s)$. A GAT $\mathcal{D}$ is said to be *acyclic* iff the relation $\prec$ is well-founded.

Informally, acyclic guarded action theories are those in which the guards of the GSFAs for a fluent F do not logically depend circularly on F itself. This allows us to safely consider the evaluation of guards as subprojection tasks to be solved again by generalized regression (see below).

In order to take into account the values actually sensed by the sensors (sensing functions), we make use of the so-called *histories* [De Giacomo and Levesque 1999b]. A *history* is a sequence $(\vec{\mu_0}) \cdot (A_1, \vec{\mu_1}) \cdot \cdots \cdot (A_n, \vec{\mu_n})$ where A_i is a ground action term[4] ($0 \leq i \leq n$) and $\vec{\mu_i} = (\mu_{i1}, \ldots, \mu_{im})$, for $0 \leq i \leq n$, is a vector of sensed values, with μ_{ij} understood as the reading of the jth sensor after the ith action. If σ is a history, we define a ground situation term $end[\sigma]$ as follows: $end[\vec{\mu_0}] = S_0$, and $end[\sigma \cdot (A, \vec{\mu})] = do(A, end[\sigma])$. We also define a ground sensor formula $Sensed[\sigma]$ as $\bigwedge_{i=0}^{n} \bigwedge_{j=1}^{m} h_j(end[\sigma_i]) = \mu_{ij}$, where σ_i is the subhistory up to action i, $(\vec{\mu_0}) \cdots (A_i, \vec{\mu_i})$, and h_j is the jth sensor function. So $end[\sigma]$ tells us all the actions performed in σ, and $Sensed[\sigma]$ tells us the sensing information.

In De Giacomo and Levesque [1999b], a form of *generalized regression* that roughly follows the one for basic action theories, but takes into account guards and sensed values, is presented for guarded action theories. Such a form of generalized regression is a sensible compromise between syntactic transformation and logical reasoning that allows us to solve (in certain cases) the projection task in an efficient way. Later on in this paper, we will present a Prolog evaluation procedure based on such a mechanism.

[4]By a history we mean no more than a situation term extended to encode sensing information. Thus, it should not be confused with similar notions in other narrative-based formalisms like Kowalski and Sergot [1986] and Baral et al. [1997]. In fact, the situation calculus itself is not a narrative-based approach, but a hypothetical reasoning one. For narratives in the situation calculus we refer to Miller and Shanahan [1994] and Reiter [2000].

In general, we cannot expect generalized regression to be a complete form of reasoning: a tautological sensor-fluent formula might be entailed even though nothing is entailed about the component fluents and hence regression cannot be applied. Nonetheless, in a practical setting, we can imagine never asking the robot to evaluate a formula unless the history is such that it knows enough about the component fluents, using the given GSSAs and GSFAs, and their component fluents. In general, we call a history *just-in-time* (JIT) for a formula, if the actions and sensing readings it contains are enough to guarantee that suitable formulas (including guards) can be evaluated at appropriate points to determine the truth value of all the fluents in the formula.

Definition 3.3 *(Just-in-time Histories for a formula).* A history σ is a just-in-time-history (JIT-history) for a (possibly open) sensor-fluent formula $\phi(\vec{x}, s)$ w.r.t. a background GAT $\mathcal{D}$ iff

—$\phi(\vec{x}, s)$ is a sensor formula (directly evaluable over the history σ);

—$\phi(\vec{x}, s) = \neg\phi_1(\vec{x}, s)$, or $\phi(\vec{x}, s) = \phi_1(\vec{x}, s) \wedge \phi_2(\vec{x}, s)$, and σ is a JIT-history for $\phi_1(\vec{x}, s)$ and $\phi_2(\vec{x}, s)$;

—$\phi(\vec{x}, s) = \exists y.\phi_1(\vec{x}, y, s)$ and σ is a JIT-history for $\phi_1(\vec{x}, y, s)$;

—$\phi(\vec{x}, s) = F(\vec{t}, s)$, where F is a fluent and σ is an initial history ($\vec{\mu_0}$), and either $\mathcal{D}_0 \models \forall F(\vec{t}, S_0)$ or $\mathcal{D}_0 \models \forall\neg F(\vec{t}, S_0)$;[5]

—$\phi(\vec{x}, s) = F(\vec{t}, s)$, where F is a fluent, and there is a GSFA $\beta(\vec{z}, s) \supset [F(\vec{z}, s) \equiv \rho(\vec{z}, s)]$ in $\mathcal{D}$ such that σ is a JIT-history for formula $\beta(\vec{t}, s)$, and such that $\mathcal{D} \cup Sensed[\sigma] \models \forall\beta(\vec{t}, end[\sigma])$;

—$\phi(\vec{x}, s) = F(\vec{t}, s)$, where F is a fluent, $\sigma = \sigma' \cdot (A, \vec{\mu})$, and there is GSSA $\alpha(\vec{z}, a, s) \supset [F(\vec{z}, do(a, s)) \equiv \gamma(\vec{z}, a, s)]$ in $\mathcal{D}$, such that σ' is a JIT-history for both $\alpha(\vec{t}, A, s)$ and $\gamma(\vec{t}, A, s)$, and $\mathcal{D} \cup Sensed[\sigma'] \models \forall\alpha(\vec{t}, A, end[\sigma])$.

Although guarded action theories are assumed to be open-world, a JIT-history provides a sort of *dynamic* closed-world assumption in that it ensures that the truth value of any fluent will be known whenever it is part of a formula whose truth value we need to determine. This allows us to evaluate complex formulas as we would if we had a normal closed-world assumption.

4. CONGOLOG: OFFLINE EXECUTION

So far, we have considered the kind of theory we intend to use to model the world and its dynamics. What we have not addressed so far is the strategy used to control the sequence of actions to perform. One possibility is to define suitable goals and to use planning to achieve them. However, Levesque [1997] argued that high-level program execution offered a practical alternative in complex domains to plan synthesis. Instead of looking for a legal sequence of action achieving some goal, the task is to find a legal execution of a high-level program that possibly includes some *nondeterministic steps*. High-level programs provide a blueprint (just a blueprint

[5]We use $\forall.\phi$ to denote the formula obtained from ϕ by quantifying universally all free variables occurring in it.

because of nondeterminism) of the solution where the user gets to control the search effort required.

Here we focus on the high-level language *ConGolog* [De Giacomo et al. 2000] which is an extension of *Golog* [Levesque et al. 1997] that allows for (interleaved) concurrency. *ConGolog* includes the following constructs (δ, possibly sub-scripted, ranges over *ConGolog* programs):

a,	primitive action
$\phi?$,	wait for a condition[6]
$(\delta_1; \delta_2)$,	sequence
$(\delta_1 \mid \delta_2)$,	nondeterministic choice between actions
$\pi v.\delta$,	nondeterministic choice of arguments
δ^*,	nondeterministic iteration
proc $P(\vec{v})$ δ ,	(recursive) procedures[7] **if** ϕ **then** δ_1 **else** δ_2,
synchronized conditional	
while ϕ **do** δ,	synchronized loop
$(\delta_1 \parallel \delta_2)$,	concurrent execution
$(\delta_1 \rangle\!\rangle \delta_2)$,	concurrency with different priorities
$\delta^{\parallel}$,	concurrent iteration
$<\phi \rightarrow \delta>$,	interrupt[8]

The primitive instructions of a *ConGolog*/*Golog* program are primitive actions which are domain dependent and whose effects, noneffects, and preconditions are modeled in a background action theory. Nondeterminism is achieved mainly with programs of the form $(\delta_1 \mid \delta_2)$ and $\pi x.\delta(x)$. Programs $(\delta_1 \mid \delta_2)$ are executed by nondeterministically choosing δ_i (with $i = 1$ or $i = 2$) and executing it, while programs $\pi x[\delta(x)]$ are executed by nondeterministically picking an individual x, and for that x, performing the program $\delta(x)$. The constructs **if** ϕ **then** δ_1 **else** δ_2 and **while** ϕ **do** δ are the synchronized versions of the usual if-then-else and while-loop. They are synchronized in the sense that 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.[9]

The construct $(\delta_1 \parallel \delta_2)$ denotes the (interleaved) concurrent execution of the actions δ_1 and δ_2. $(\delta_1 \rangle\!\rangle \delta_2)$ denotes the concurrent execution of the actions δ_1 and δ_2 with δ_1 having higher priority than δ_2. This restricts the possible interleavings of the two processes: δ_2 executes only when δ_1 is either done or blocked. Observe that this is a very general form of prioritized concurrency. For a full illustration of the various constructs, refer to De Giacomo et al. [2000].

[6]Because there are no exogenous actions or concurrent processes in *Golog*, waiting for ϕ amounts to testing that ϕ holds in the current state.

[7]For the sake of simplicity, we will not discuss recursive procedures in this paper.

[8]In fact, interrupts can be defined in terms of the other constructs in the language.

[9]In Levesque [1997], nonsynchronized versions of if-then-elses and while-loops are introduced. The synchronized versions of these constructs introduced here behave essentially as the nonsynchronized ones in absence of concurrency. Nevertheless, the difference is striking when concurrency is allowed.

EXAMPLE 4.1. *The following* ConGolog *(and* Golog *since it does not involve concurrency) program for an elevator serves all the floors whose lights are on:*

$$(\pi x.[Light(x)?; (goUp^*|goDown^*); serve(x)])^*; (\neg\exists f.Light(f))?$$

By using the nondeterministic choice of arguments operator π, the procedure nondeterministically picks up a floor x, tests whether the light of floor x is on, and if it is, goes to such floor (by going up or down some nondeterministic number of times) and serves it (maybe by opening the door, closing it, and switching off its light). Finally, the program uses the nondeterministic iteration operator to repeat the above behavior zero or more times until there is no floor requesting to be served.

Notice that the situation term is removed from all fluents in high-level programs, since they are assumed to be the "current" situation as it happens in any conventional programming language. Formally, *ConGolog* executions are captured by introducing two predicates $Trans$ and $Final$ that together define a single-step transition semantics [Nielson and Nielson 1992; Hennessy 1990; Plotkin 1981]:

—$Trans(\delta, s, \delta', s')$ is intended to say that program δ in situation s may legally execute one step, ending in situation s' with program δ' remaining.

—$Final(\delta, s)$ is intended to say that program δ may legally terminate in situation s.

Using these two predicates we can define what it means to execute a program. One possibility is to define a so-called *offline interpreter*, i.e., an interpreter that searches for a successful execution of the entire program before actually executing any action [Levesque et al. 1997; De Giacomo et al. 2000]. Observe that this is the way to execute the program required by the example above, since nondeterminism has to be resolved favorably, typically by backtracking, to get the program to termination.[10]

To define an offline interpreter we first define a predicate $Do(\delta, s, s')$ as follows:

$$Do(\delta, s, s') \equiv \exists\delta'.Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$$

where $Trans^*$ is the second-order definition of the transitive closure of $Trans$ (considering it as a binary relation between pairs of program-situation).

Thus, an offline interpreter looks for a situation S_f such that $Do(\delta, S_0, S_f)$ is logically implied by the action theory and the definitions of $Trans$ and $Final$. From the situation S_f which has the form $do(A_n, \ldots (do(A_1, S_0)) \ldots)$ the sequence of actions, $A_1, \ldots, A_n$, to actually execute is extracted.

We say that a program δ admits a successful *offline execution* from a situation s if there exists a ground situation term s_t such that $Do(\delta, s, s_t)$ is entailed by the action theory.

5. INDIGOLOG: ONLINE EXECUTION

The usual *ConGolog* offline interpreter described above cannot make use of actual sensed values exactly because it works offline. An alternative approach is defining a

[10]This form of nondeterminism is sometimes call *angelic nondeterminism* (e.g., Best [1996].)

so-called *online interpreter* that searches for a next legal step, executes it, and then continues [De Giacomo and Levesque 1999a; De Giacomo et al. 1998]. It is clearly possible in this schema to take into account sensed values. Hence this approach is much more suitable for our context. In particular, we can define online execution in our context as follows.

Definition 5.1 *(Online Execution of Programs w.r.t. GAT).* An online execution of a program δ_0 starting from a history σ_0 is a sequence $(\delta_0, \sigma_0), \ldots, (\delta_n, \sigma_n)$, such that for $i = 0, .., n-1$

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

$$\sigma_{i+1} = \begin{cases} \sigma_i, & \text{if } end[\sigma_{i+1}] = end[\sigma_i] \\ \sigma_i \cdot (a, \vec{\mu}), & \text{if } end[\sigma_{i+1}] = do(a, end[\sigma_i]) \\ & \text{and } \vec{\mu} \text{ is the sensor results after } a \end{cases}$$

The online execution is successful if

$$Axioms \cup Sensed[\sigma_n] \models Final(\delta_n, end[\sigma_n]).$$

The online execution is unsuccessful if

$$Axioms \cup Sensed[\sigma_n] \not\models Final(\delta_n, end[\sigma_n])$$

and there is no program δ_{n+1} and situation s such that

$$Axioms \cup Sensed[\sigma_n] \models Trans(\delta_n, end[\sigma_n], \delta_{n+1}, s)$$

where *Axioms* denotes the background theory $\mathcal{D}$ plus a set of axioms defining the predicates $Trans$, $Final$ (see De Giacomo et al. [2000]).

It can be shown that, in the absence of sensing, whenever there is a successful (i.e., legally terminating) online execution, there is also a successful offline one. However, the converse is not true, as the following example shows.

EXAMPLE 5.2. *Consider the simple program $\delta = (\phi?; a) \mid (\neg\phi?; a)$, such that $Axioms \not\models \phi[S_0]$, $Axioms \not\models \neg\phi[S_0]$, but $Axioms \models Poss(a, S_0)$. Then, δ admits a successful offline execution: indeed $Axioms \models Do(\delta, S_0, do(a, S_0))$, in other words, the offline interpreter may return a as the sequence to actually execute. However, there is no successful online execution of δ in S_0, since there is no single transition logically implied by Axioms: indeed, we have that neither $Axioms \models \exists\delta'.Trans((\phi?; a), S_0, \delta', S_0)$ nor $Axioms \models \exists\delta'.Trans((\neg\phi?; a), S_0, \delta', S_0)$.*

Although we are usually interested in finding successful executions, it is also interesting to know whether there is an unsuccessful one, namely a sequence of (bad) transitions that derives to a dead-end. Even when there is a successful online execution of a program, without some form of lookahead, or any provision for backtracking after a nondeterministic choice, we cannot guarantee that we will be able to follow it, or in other words, we cannot guarantee that we will avoid reaching an *unsuccessful* execution as shown by the next example.

EXAMPLE 5.3. *Consider the program $\delta = ((a; \phi?) \mid a)$ and assume that $Axioms \models Poss(a, S_0) \wedge \neg\phi[do(a, S_0)]$. Without any form of lookahead, nothing prevents an online interpreter from choosing to execute the first program $(a; \phi?)$, execute the*

action a, and then get blocked due to the fact that ϕ does not hold in the resulting situation. Observe that this may happen is spite of the fact that the second program a does admit a successful online execution.

Now observe that backtracking is out of the question in online executions, since actions are performed in the real world. Hence we have to concentrate on some form of lookahead. One possibility is to perform a full lookahead on the entire program as in De Giacomo et al. [1998]. However in general it is appropriate to put the amount of lookahead under the control of the programmer. In this way, the programmer may ask to explore a search space that is significantly smaller than all possible nondeterministic executions of the entire program.

Work on interleaving planning with execution of actions in the world and sensing [Jonsson and Backstrom 1995; Shanahan 1999; Kowalski 1995; Kowalski and Sadri 1999; Baral et al. 1997] suggests that a combination between online and offline execution of programs arises as a practical and powerful approach to high-level programs. Following De Giacomo and Levesque [1999a], we add to *ConGolog* a local lookahead construct that is based on looking for a successful offline execution of parts of the program specified by the programmer. The construct is called the *search operator* (Σ), and $Trans$ and $Final$ for it are defined as follows:

$$\begin{array}{l} Trans(\Sigma\delta, s, \delta', s') \equiv \\ \quad \exists\gamma, \gamma', s''.\delta' = \Sigma\gamma \wedge Trans(\delta, s, \gamma, s') \wedge Trans^*(\gamma, s', \gamma', s'') \wedge Final(\gamma', s'') \\ Final(\Sigma\delta, s) \equiv Final(\delta, s) \end{array}$$

Roughly speaking, $\Sigma\delta$ selects from all possible transitions of δ in the situation s those for which there exists a sequence of further transitions leading to a final configuration (and hence a successful completion). Moreover, the Σ operator is propagated to the remaining program so that this restriction is enforced throughout the execution of δ.

EXAMPLE 5.4. *Let us consider the program $\delta = [goUp^*; (Light(floor) \wedge \neg\exists x.x > floor \wedge Light(x))?]$ which says to go up for a nondeterministically chosen number of times and then check whether the highest floor with the light on has been reached. And let us assume that we have enough information in Axioms that the value of the test is correctly determined in each reachable situation. Obviously there are several unsuccessful online executions of δ, all those that result in getting to some floor that does not satisfy the test. Instead executing $\Sigma\delta$ online would result in keeping going up until the test is satisfied, i.e., would result in a successful online execution.*

Observe that in principle we can nest search operators. It can be shown however that nesting search operators is equivalent to applying the search operator only once. Hence in the following we will assume that no nested search operators are present in our programs.

We call *IndiGolog* (incremental deterministic *Golog*) the high-level programming language resulting from extending *ConGolog* with a search operator, and with the proviso that programs are to be designed to be executed in an online style. Below we study such language w.r.t. background theories that are guarded action theories.

6. HISTORIES AND INDIGOLOG

What kinds of histories make sense for our purposes? First of all, a sequence of actions should be such that each particular action in the sequence can indeed be executed.[11]

Definition 6.1. A history σ is *executable* w.r.t. a GAT $\mathcal{D}$ iff either $\sigma = \vec{\mu_0}$, or $\sigma = \sigma' \cdot (A, \vec{\mu})$, σ' is an executable history, and $\mathcal{D} \cup Sensed[\sigma'] \models Poss(A, end[\sigma'])$.

Furthermore, not only should a history be executable, but it should also be consistent with the conclusions drawn. Since we may have multiple axioms per fluent, we have to make sure that those axioms are all consistent with each other. In particular, sensors must provide information reflecting the real state of the world: if a robot knows the door is closed after closing it, then the sensors should not tell it about the door being open after the closing action!

Definition 6.2 *(Coherent Histories).* A history σ is *coherent* w.r.t. a GAT $\mathcal{D}$ iff it is an executable history and $\mathcal{D} \cup Sensed[\sigma]$ is satisfiable.

Coherent histories have the property that they do not rule out all possible sensor readings after performing an action whose preconditions are satisfied. After some legal sequence of action, there must be at least one reading of the sensors not contradicting the state of the world after the action.

PROPOSITION 6.3. *If σ is a coherent history w.r.t. a GAT $\mathcal{D}$, then for every action A such that $\mathcal{D} \cup Sensed[\sigma] \models Poss(A, end[\sigma])$ there exists a sensor-reading vector $\vec{\mu}$ such that the history $\sigma' = \sigma \cdot (A, \vec{\mu})$ is coherent w.r.t. $\mathcal{D}$.*

Indeed, given a model M of $\mathcal{D} \cup Sensed[\sigma]$, $M \models Poss(A, end[\sigma])$, and moreover M gives an interpretation to all sensors in situation $do(A, end[\sigma])$. Now considering that $\mathcal{D} \cup Sensed[\sigma] \models Poss(A, end[\sigma])$, and that sensed values are interpreted over concrete domains, we can use that interpretation to build a sensor-reading vector $\vec{\mu}$ such that $\sigma' = \sigma \cdot (A, \vec{\mu})$ is coherent w.r.t. $\mathcal{D}$. Observe that as a consequence of this proposition we have that if a guarded action theory $\mathcal{D}$ is consistent then there exists an initial coherent history.

Next, we formalize what is meant by a guarded action theory in which the reasoning about sensors is restricted up to some situation. After that situation the sensors are not taken into account.

Definition 6.4. Given a GAT $\mathcal{D}$ and a situation s', the GAT $\mathcal{D}^{s'}$ is obtained by replacing each GSFA $\{\alpha(\vec{x}, s) \supset F(\vec{x}, s) \equiv \rho(\vec{x}, s)\}$ in $\mathcal{D}$ by the new GSFA $\{\alpha(\vec{x}, s) \wedge s \sqsubseteq s' \supset F(\vec{x}, s) \equiv \rho(\vec{x}, s)\}$.

Here, the relation $s \sqsubseteq s'$ means that s is a prefix of s', maybe s' itself [Reiter 1991]. Observe that the kind of GSFAs $\mathcal{D}^{s'}$ are not allowed in guarded action theories because of the presence of the relation $\sqsubseteq$. In fact, this special form of GSFAs is used below only to suitably restrict reasoning on sensors when needed.

Using such definition we can introduce the notion of JIT-histories w.r.t. $\mathcal{D}^{s_t}$ as a revised version of Definition 3.3.

[11] Note that we say that an action is executable if *Poss* for that action holds. That is, we are not considering the possibility for the action to fail when executed in the real world. See De Giacomo et al. [1998] for ways to deal with such failures.

Definition 6.5 *(Just-in-time Histories for a Formula (Revised)).* A history σ is a just-in-time-history (JIT-history) for a sensor-fluent formula $\phi(\vec{x}, s)$ w.r.t. $\mathcal{D}^{s_t}$ iff

—$\phi(\vec{x}, s)$ is a sensor formula with the proviso that if $s_t \sqsubseteq end[\sigma]$ it does not mention any sensing function;

—$\phi(\vec{x}, s) = \neg\phi_1(\vec{x}, s)$, or $\phi(\vec{x}, s) = \phi_1(\vec{x}, s) \wedge \phi_2(\vec{x}, s)$, and σ is a JIT-history for $\phi_1(\vec{x}, s)$ and $\phi_2(\vec{x}, s)$ w.r.t. $\mathcal{D}^{s_t}$;

—$\phi(\vec{x}, s) = \exists y.\phi_1(\vec{x}, y, s)$ and σ is a JIT-history for $\phi_1(\vec{x}, y, s)$ w.r.t. $\mathcal{D}^{s_t}$;

—$\phi(\vec{x}, s) = F(\vec{t}, s)$, where F is a fluent, and σ is an initial history ($\vec{\mu_0}$), and either $\mathcal{D}_0 \models \forall F(\vec{t}, S_0)$ or $\mathcal{D}_0 \models \forall\neg F(\vec{t}, S_0)$;

—$\phi(\vec{x}, s) = F(\vec{t}, s)$, where F is a fluent, and there is a GSFA $\beta(\vec{z}, s) \supset [F(\vec{z}, s) \equiv \rho(\vec{z}, s)]$ in $\mathcal{D}^{s_t}$ such that σ is a JIT-history for formula $\beta(\vec{t}, s)$ w.r.t. $\mathcal{D}^{s_t}$, and such that $\mathcal{D}^{s_t} \cup Sensed[\sigma] \models \forall\beta(\vec{t}, end[\sigma])$.

—$\phi(\vec{x}, s) = F(\vec{t}, s)$, where F is a fluent, $\sigma = \sigma' \cdot (A, \vec{\mu})$, and there is GSSA $\alpha(\vec{z}, a, s) \supset [F(\vec{z}, do(a, s)) \equiv \gamma(\vec{z}, a, s)]$ in $\mathcal{D}^{s_t}$, such that σ' is a JIT-history for both $\alpha(\vec{t}, A, s)$ and $\gamma(\vec{t}, A, s)$ w.r.t. $\mathcal{D}^{s_t}$, and $\mathcal{D}^{s_t} \cup Sensed[\sigma'] \models \forall\alpha(\vec{t}, A, end[\sigma])$.

This definition differs from Definition 3.3 in that it does not assume complete sensing information, but only up to a certain (past) situation. Indeed, the agent may have already committed only to some prefix of σ, and after that the agent is reasoning hypothetically and without any feedback from its on-board sensors. Finally, we extend the notion of JIT-histories to whole *IndiGolog* programs.

Definition 6.6 *(Just-in-time Histories for a Program).* A history σ is a said to be a JIT-history for a program δ w.r.t. to a GAT $\mathcal{D}$ iff

—$\delta = a$, where a is an action, with precondition axiom $Poss(a, s) \equiv \phi(s)$, and σ is a JIT-history for formula $\phi(s)$ w.r.t. $\mathcal{D}$;

—$\delta = \phi?$ and σ is a JIT-history for formula $\phi(s)$ w.r.t. $\mathcal{D}$;

—$\delta = \delta_1; \delta_2$, σ is a JIT-history for δ_1 w.r.t. $\mathcal{D}$, and σ is also a JIT-history for δ_2 w.r.t. $\mathcal{D}$ only if it the case that $\mathcal{D} \cup \mathcal{C} \cup Sensed[\sigma] \models Final(\delta_1, end[\sigma])$;

—$\delta = (\delta_1 \mid \delta_2)$, or $\delta = (\delta_1 \| \delta_2)$, or $\delta = (\delta_1 \rangle\rangle \delta_2)$, or $\delta_1^{\|}$, or $\delta = \pi x.\delta_1$, and σ is a JIT-history for programs δ_1 and δ_2 w.r.t. $\mathcal{D}$;

—$\delta =$ if ϕ then δ_1 else δ_2 | while ϕ do δ_1 $|< \phi \rightarrow \delta_1 >$, σ is a JIT-history for condition $\phi(s)$ and programs δ_1 and δ_2 w.r.t. $\mathcal{D}$;

—$\delta = \Sigma\delta_1$, σ is a JIT-history for program δ_1 w.r.t. $\mathcal{D}$, and for every pair (δ', s') such that $\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models Trans^*(\delta_1, end[\sigma], \delta', s')$, and for every σ' that is an extension of σ such that $end[\sigma'] = s'$, σ' is a JIT-history for δ' w.r.t. the GAT $\mathcal{D}^{end[\sigma]}$.

In other words, a history is just-in-time for a program if it is a JIT-history for every formula that needs to be evaluated at the next step. For the search operator, we impose a more sophisticated requirement that takes into account that the search typically involves several steps and that it is done offline and hence sensed values are not available. This guarantees that we can determine the truth-value of each test in the program, as needed using regression and avoiding general theorem-proving.

In particular, when performing a lookahead we require all the information needed to do the search on the program. In other words, every possible configuration that is reachable from the beginning of the search should be "just in time," even if it belongs to a (bad) branch of execution not reaching any final configuration.

7. AN INCREMENTAL INTERPRETER

We have developed a simple incremental interpreter in Prolog that implements the online execution of high-level programs based on guarded theories of action including sensing, and an approximation of the search operator. Observe that the task is fundamentally a theorem proving one: does a certain $Trans$ or $Final$ formula follow logically from the axioms of the action theory and the sensing information obtained so far? We exploit the sort of dynamic closed-world assumption provided by JIT-histories to simplify such a theorem-proving task to *evaluation*. The interpreter can be divided into three parts: the evaluation of test conditions, the implementation of $Trans$ and $Final$, and the main loop.

7.1 Evaluating Test Conditions

As already said, an evaluation procedure to compute the projection task is needed. In previous implementations [Levesque et al. 1997; De Giacomo et al. 2000; De Giacomo and Levesque 1999a], a procedure `holds(`c, s`)` was used to test whether c held in situation s. Those implementations relied on a CWA over the formulas being evaluated. Here, we do not want to rely on a strict CWA, but still we want to have correct execution of programs. In De Giacomo and Levesque [1999b], an evaluation procedure `eval/3` for guarded theories was given. Here we give an alternative one for which we prove soundness and completeness in the next section.

We assume the user provides the following set of clauses D corresponding to the domain description, i.e., the background guarded action theory:

—`fluent(`F`)`, for each relational fluent F.

—`sensor(`h_i`)`, for each sensor function h_i.

—`init(`F`)/init(`$neg(F)$`)`, for each fluent F with $\mathcal{D}_0 \models F(S_0)$ or $\mathcal{D}_0 \models \neg F(S_0)$.

—`closed(`F`)`, whenever $\mathcal{D}_0 \models F(S_0)$ or $\mathcal{D}_0 \models \neg F(S_0)$.

—`gsfa(`$F(\vec{x})$`,`$\beta(\vec{x})$`,`$\rho(\vec{x})$`)`, for each GSFA $\beta(\vec{x}, s) \supset [F(\vec{x}, s) \equiv \rho(\vec{x}, s)]$. We assume each sensor formula $\rho(\vec{x}, s)$ is formed from atomic formulas mentioning at least one sensing function.

—`gssa(`A`,`$F(\vec{x})$`,`$\alpha(\vec{x}, a)$`,`$\gamma(\vec{x}, a)$`)`, for each GSSA $\alpha(\vec{x}, a, s) \supset [F(\vec{x}, do(a, s)) \equiv \gamma(\vec{x}, a, s)]$.

Observe that the predicates `gsfa/3` and `gssa/4` represent respectively GSFAs and GSSAs with the situation argument suppressed.

To deal with sensing, we assume we have a database of all sensor results up to history σ. In Prolog, this database consists of clauses `has_sensor(`h_i, v_i, s`)` where v_i is the value of the sensor function h_i in history σ ($s = end[\sigma]$), i.e., $\mathcal{D} \cup Sensed[\sigma] \models h_i(s) = v_i$. In practice, we would use an efficient data structure

storing only changes to sensor values.[12] Finally, we represent the sequence of actions in the current history as Prolog lists.

With these predicates in place, the code for the evaluation procedure is as follows (we deal with negation, conjunction, and existential quantification only, and consider the other logical operators as abbreviations):

```
eval(neg(P),H,true)      :- eval(P,H,false).
eval(neg(P),H,false)     :- eval(P,H,true).
eval(and(P1,P2),H,true)  :- eval(P1,H,true)  , eval(P1,H,true).
eval(and(P1,P2),H,false) :- eval(P1,H,false) ; eval(P1,H,false).
eval(some(V,P),H,Bool_P) :- varsort(V,D), eval(some(V,D,P),H,Bool_P).
eval(some(V,D,P),H,true) :- domain(D,O), subv(V,O,P,P1), eval(P1,H,true).
eval(some(V,D,P),H,false):- not alltrue(P,V,D,H).
alltrue(P,V,D,H):- domain(D,O), subv(V,O,P,P1), not evalfalse(P1,H).
evalfalse(P,H)  :- eval(P,H,false).

/* P is fluent */
eval(P,H,Bool_P) :- fluent(P), bool(P,Bool_P,H).

/* P is a relational operator (like =, <, >, etc.) */
eval(P,H,Bool_P) :-
   rel(P,OP,S,V), subf(S,S1,H), subf(V,V1,H), has_value(OP,S1,V1,Bool_P).

/* bool(F,V,H): Relational fluent F has value V in H */
bool(F,true,[]) :- init(F).
bool(F,false,[]):- (closed(F), not init(F)) ; init(neg(F)).
bool(F,V,H)     :- gsfa(F,G,P), eval(G,H,true), eval(P,H,V).
bool(F,V,[A|H]) :- gssa(A,F,G,P), eval(G,H,true), eval(P,H,V).

/* P2 is P1 with sensors replaced by their values in H*/
subf(P1,P2,H):- sensor(P1), has_sensor(P1,P2,H).
subf(P1,P1,H):- not sensor(P1).
```

In this program, the predicate `varsort(`*var*`,`*sort*`)` is provided by the user and asserts that variable *var* ranges over the set of elements *sort*. Similarly, the predicate `domain(`*sort*`,`*obj*`)` is also provided by the user and denotes the objects *obj* that are of sort *sort*. The predicate `subv(`v`,`x`,`p`,`p'`)` implements the substitution for both program and formulas where p' is obtained by substituting x for v in p. The predicate `rel(`p`,`*op*`,`s`,`v`)` asserts that the term p denotes a binary predefined relational operator *op* applied to the operands s and v. The predicate `has_value(`*op*`,`s`,`v`,`b`)` asserts that applying the relational operator *op* to the operands s and v one gets the truth value b. A possible implementation of these last two predicates is the following:

```
rel(P,OP,S,V) :- P =..[OP,S,V].

has_value(OP,S,V) :- P =..[OP,S,V], call(P).
```

[12] In addition, both the sensor database and the initial database should be progressed after some number of actions.

The predicate `eval/3` determines the truth value of a formula, by first recursively reducing `and`, `neg`, and `some` formulas to the atomic cases: either fluents or a relational operator possibly involving sensors. Notice how existential is handled by retrieving the possible elements for the free variable. For fluents, `eval/3` uses `bool/3` to determine the value of a fluent in any history: for the initial situation, `bool/3` tries to use the user-provided value; otherwise, it attempts to find a GSFA or GSSA whose guard recursively evaluates to true. For relational operators, the last clause of `eval/3` first replaces sensing functions appearing as operands with their values in the history. Note that this code only deals with relational fluents, but that it is straightforward to accommodate functional fluents in a modular way.

7.2 Implementation of $Trans$ and $Final$

Clauses for $Trans$ and $Final$ are needed for each of the program constructs. Since many language constructs involve conditions, we use an evaluation procedure for guarded action theories shown above. However, in the presence of prioritized concurrency, we cannot use it, since unsound transition may arise. To see why, consider the program $(\phi?\rangle\rangle\, \delta_2)$ where $Axioms \not\models \phi(S_0)$ and $Axioms \not\models \neg\phi(S_0)$. It is easy to see that there should be no execution of this program in S_0. Indeed, the wait action $\phi?$ cannot succeed, because ϕ is not known to hold, and moreover, δ_2 cannot be used, since $\neg\phi$ is not know to hold either. Recall, that in order to execute the lower-priority program, all higher-priority process should be *completely* blocked. Nevertheless, if the evaluation of ϕ merely fails, then so would the transition for $\phi?$, and we would be free to execute δ_2, allowing unsound transitions. To tackle this problem we use a "secure evaluation" procedure `evalsec/3` that aborts its execution to the top level whenever it cannot determine the truth value of the formula being evaluated.

```
evalsec(P,H,B):- eval(P,H,B1)-> B1=B ; (write('Unknown:'),write(P), abort).
```

Let us now consider the code for the *Golog* constructs: sequence is represented as a Prolog list; `ndet(`δ_1`,`δ_2`)` stands for the program $(\delta_1 \mid \delta_2)$; `?(`ϕ`)` for $\phi?$, `star(`δ`)` for δ^*, and `pi(`v`,`δ`)` stands for $\pi v.\delta(v)$. Finally, `if(`ϕ`,`δ_1`,`δ_2`)` stands for the **if-then-else** construct, and `while(`ϕ`,`δ`)` for the **while** one.

```
final([],_).
final(star(E),_).
final([E|L],H)       :- final(E,H), final(L,H).
final(ndet(E1,E2),H) :- final(E1,H) ; final(E2,H).
final(if(P,E1,E2),H) :- evalsec(P,H,Bool_P),
        ( (Bool_P=true,  final(E1,H)) ; (Bool_P=false, final(E2,H)) ).
final(while(P,E),H) :- evalsec(P,H,Bool_P), ((Bool_P=false) ; (final(E,H))).
final(pi(V,E),H) :- varsort(V,D), domain(D,O), subv(V,O,E,E2), final(E2,H).

trans([E|L],H,E1,H1)      :- not L=[], final(E,H), trans(L,H,E1,H1).
trans([E|L],H,[E1|L],H1)  :- trans(E,H,E1,H1).
trans(?(P),H,[],H)        :- evalsec(P,H,true).
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(pi(V,E),H,E1,H1)    :- varsort(V,D), domain(D,O),
                             subv(V,O,E,E2), trans(E2,H,E1,H1).
```

```
trans(if(P,E1,E2),H,E,H1) :- evalsec(P,H,Bool_P),
      ( (Bool_P=true, trans(E1,H,E,H1)) ; (Bool_P=false, trans(E2,H,E,H1)) ).
trans(star(E),H,[E1,star(E)],H1)       :- trans(E,H,E1,H1).
trans(while(P,E),H,[E1,while(P,E)],H1) :- evalsec(P,H,true),
                                          trans(E,H,E1,H1).
trans(E,H,[],[E1|H]) :- prim_action(E1), poss(E1,P), evalsec(P,H,true).
```

We restrict our attention to programs that are *bounded* in the sense that they never evaluate open formulas. To do so, we restrict the variables of the nondeterministic choice construct $\pi x.\delta$ and the existential quantification $\exists x.\alpha$ to range over a finite domain or sort. Observe how `varsort/2` and `domain/2` are used for the nondeterministic choice of argument to select each possible element for a variable. In addition to `varsort/2` and `domain/2`, the user needs to give the clauses for `prim_action/1` and `poss/2` to describe the named actions and their corresponding precondition axioms.

Next, we list the code for the *ConGolog* constructs. `conc(`δ_1`,`δ_2`)` stands for the program $(\delta_1 \| \delta_2)$; `pconc(`δ_1`,`δ_2`)` stands for $(\delta_1 \rangle\rangle \delta_2)$; and finally, `iconc(`$\delta$`)` stands for $\delta^{\|}$.

```
final(conc(E1,E2),H)  :- final(E1,H), final(E2,H).
final(pconc(E1,E2),H) :- final(E1,H), final(E2,H).
final(iconc(E),H).

trans(conc(E1,E2),H,conc(EE,E2),H1)   :- trans(E1,H,EE,H1).
trans(conc(E1,E2),H,conc(E1,EE),H1)   :- trans(E2,H,EE,H1).
trans(pconc(E1,E2),H,pconc(EE,E2),H1) :- trans(E1,H,EE,H1).
trans(pconc(E1,E2),H,pconc(E1,EE),H1) :- not existtrans(E1,H),
                                         trans(E2,H,EE,H1).
existtrans(E1,H):- trans(E1,H,_,_).
trans(iconc(E),H,conc(E1,iconc(E)),H1):- trans(E,H,E1,H1).
```

We do not show here the code for interrupts, but they can be easily defined in terms of the other constructs. The only tricky part of *Trans* and *Final* has to do with the search operator, because we have to avoid the potential presence of sensing information inside a local lookahead. Nevertheless, we can obtain a simple, but quite effective version of the search operator. The Prolog term `search(`δ`)` stands for the corresponding program $\Sigma\delta$.

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

/* Look for a good path */
path(E,H):- final(E,H).
path(E,H):- trans(E,H,E1,H1), path(E1,H1).
```

Briefly, our implementation of the search operator tries to find a finite sequence of transitions[13] leading to a successful execution. However, since this search is done offline (i.e, no actual action is executed in the real world), no new sensing

[13]The path it finds can be cached and reused so long as no external actions intervene as it is being executed.

results are acquired during it. Among other things, this means that the implementation is not able to determine whether some currently unknown fluent will be known when needed. For example, suppose that fluent f is unknown, but that after performing action a, a sensor will provide its correct value. Then the program `search([a,if(f,p1,p2)])` will fail, given that a is being considered offline. However, the alternative program `[a,search(if(f,p1,p2))]` would work fine because the sensor is consulted before the search. It is precisely to deal with this issue in a practical way that we use online execution. Relying on the fact that there is, usually, a limited amount of sensing inside an offline search, we think that conditional plans could also be calculated efficiently, as suggested in Lakemeyer [1999].

7.3 Main Loop

The top part of the interpreter deals with the execution of actions in the world. It makes use of *Trans* and *Final* to determine the next action to perform and to end the execution.

```
indigo(E,H):- trans(E,H,E1,H), !, indigo(E1,H1).
indigo(E,H):- trans(E,H,E1,do(A,H)), execute(A,H), !, indigo(E1,do(A,H)).
indigo(E,H):- final(E,H).
```

At each step, either we find a legal transition, commit to it, and continue with the new history, or we terminate successfully. Committing means to execute a new action in the world, if the transition requires it,[14] and executing a new action in the world implies getting new sensing results.

As mentioned we store these values by adding facts to the predicate `has_sensor/3`. In a sample implementation we can just ask for the new sensing results to the user. For example, we may define a predicate `ask_sensor(`s`,`h`)` that asks for the value of a sensor s in the last situation of the history h and asserts it as a new `has_sensor/3` fact, as in the following code:

```
execute(A,H) :- exec_act(A), ask_all_sensors(do(A,H)).

exec_act(A) :- write('Executed action '), write(A), nl.

ask_all_sensors(H) :- sensor(S), ask_sensor(S,H), fail.
ask_all_sensors(_).

ask_sensor(S,H):- write('Enter value for sensor: '), write(S), nl,
                  write('At: '), write(H), nl,
                  write('Value: '), read(V), assert(has_sensor(S,V,H)).
```

Observe, that instead of retrieving all sensor values after an action is executed in the world, it may be sensible to ask for them when required the first time (i.e., on demand) during the execution of the *IndiGolog* program. Indeed most of these values are typically not useful.

We remark that it is easy to modify the interpreter to deal with exogenous events, i.e., actions that may occur without being part of the program as, for instance,

[14] Recall that a wait condition step involves no new action.

actions performed by other agents or even by nature itself. In particular, one may add to the definition of `indigo/2` the following clause:

```
indigo(E,H):- exog_occurs(Act,H), !, indigo(E,[Act|H]).
```

In fact, to fully deal with exogenous events one should define a search operator that anticipates exogenous events, guaranteeing executability whatever the exogenous events turn out to be. However, defining and implementing a search dealing with exogenous events in this way still remains an open problem.

8. CORRECTNESS OF THE INTERPRETER

To finish, we show soundness and completeness results for the interpreter. Our proofs rely on the following assumptions:

—We assume consistent guarded action theories, coherent histories, and (bounded) programs without nested search operators.
—The domain theory $\mathcal{D}$ enforces the unique name assumption (UNA) on both actions and objects.
—The predicate `subv/4` correctly implements the substitution for both programs and formulas.

We start by centering our attention on soundness and completeness results for the evaluation procedure. From now on, D is the set of user-provided clauses describing the domain; P_D is the program D union the set of clauses implementing the evaluation procedure (Section 7.1); P'_D is P_D plus the only clause for `evalsec/3`; and T_D is P'_D union all the rules implementing predicates $Trans$ and $Final$, i.e., all the rules given in Section 7.2.

THEOREM 8.1 (SOUNDNESS OF `eval/3`). *Let $\phi(s)$ be a sensor-fluent formula with no free variable except the situation argument s, and let σ be a history. Assume sensors are available up to some subhistory σ' of σ (i.e., $end[\sigma'] \sqsubseteq end[\sigma]$).*[15] *Whenever the goal `eval(`$\phi, end[\sigma], B$`)` succeeds w.r.t. program P_D with computed answer $B = b$:*

(i) if $b = true$, then $\mathcal{D}^{end[\sigma']} \cup Sensed[\sigma] \models \phi(end[\sigma])$;
(ii) if $b = false$, then $\mathcal{D}^{end[\sigma']} \cup Sensed[\sigma] \models \neg\phi(end[\sigma])$.

Given that P'_D includes the only clause for `evalsec/3` we have

COROLLARY 8.2 (SOUNDNESS OF `evalsec/3`). *Under the same assumptions as Theorem 8.1, the goal `eval(`$\phi, end[\sigma], B$`)` is sound w.r.t. program P'_D.*

Now, we state a completeness result for the evaluation procedure. For that, we recall the notion of JIT-histories for formulas introduced in Section 6.

THEOREM 8.3 (COMPLETENESS OF `eval/3`). *Let $\mathcal{D}$ be an acyclic GAT; let $\phi(s)$ be a sensor-fluent formula with no free variable except the situation argument s; and let σ be a JIT-history for $\phi(s)$ w.r.t. $\mathcal{D}^{end[\sigma']}$ where sensors are available up to the sub-history σ' of σ (i.e., $end[\sigma'] \sqsubseteq end[\sigma]$). Then, the goal `eval(`$\phi, end[\sigma], B$`)` succeeds w.r.t. program P_D and with answer $B = true$ or $B = false$.*

[15] Sensing outcomes from σ' to σ are meaningless, so we can assume any value for such outcomes.

COROLLARY 8.4 (COMPLETENESS OF `evalsec/3`). *Let $G1$ and $G2$ be the goals* `evalsec(`$\phi, end[\sigma], B$`)` *and* `evalsec(`$\phi, end[\sigma], b$`)` *(where b stands for $true$ or $false$) respectively w.r.t. program P'_D. Then, the goal $G1$ does not finitely fail (i.e., $G1$ may succeed or abort to the top level). What is more, $G1$ succeeds with answer $B = true$ or $B = false$, and goal $G2$ either succeeds or finitely fails, provided the assumptions in Theorem 8.3 apply.*

It is important to notice that both results are relative to the slightly modified theories of Definition 6.4, which are determined by the availability of the sensor outcomes. It is not hard to propagate the soundness of the evaluation procedure to `trans/4` and `final/2` for programs not containing search.

THEOREM 8.5 (SOUNDNESS OF $Trans$ AND $Final$ WITHOUT SEARCH). *Let δ be a bounded program without free variables and not mentioning search, and $\mathcal{D}$ the underlying GAT. Let σ be a history, and let P and H be Prolog variables. Assuming that sensors are available up to some subhistory σ' of σ (i.e., $end[\sigma'] \sqsubseteq end[\sigma]$) and the program used is T_D we have*

(1) If `trans(`$\delta, end[\sigma], P, H$`)` *succeeds with $P = \delta', H = s'$, then*

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

Furthermore, δ' and s' do not contain any free variable.

(2) If `trans(`$\delta, end[\sigma], P, H$`)` *finitely fails, then*

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

(3) If `final(`$\delta, end[\sigma]$`)` *succeeds, then*

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

(4) If `final(`$\delta, end[\sigma]$`)` *finitely fails, then*

$$\mathcal{D}^{end[\sigma']} \cup \mathcal{C} \cup Sensed[\sigma] \models \neg Final(\delta, end[\sigma]).$$

In general, because programs are mostly executed online, we expect σ' to be exactly σ. The exception arises when a search is started, since we do not have future sensing outcomes, nor do we reason by cases about them. Thus, the next step is to understand how search really behaves. As already stated, its implementation is just an approximation of its definition, and therefore it is important to capture in which sense it is sound.

LEMMA 8.6 (SOUNDNESS OF SEARCH). *Let $\Sigma\delta$ be a bounded program with no free variables, and let $\mathcal{D}$ be the underlying GAT.*

Whenever the goal `trans(`$\Sigma\delta, end[\sigma], P, S$`)` *succeeds on program T_D with answer $P = \Sigma\delta', S = s'$ and with sensors available up to σ itself, the following holds:*

$$\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models Trans(\delta, end[\sigma], \delta', s') \text{ and}$$
$$\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models \exists s''. Do(\delta', s', s'').$$

Furthermore, δ' and s' do not contain any free variables. What is more, if goal `final(`$\Sigma\delta, end[\sigma]$`)` *succeeds then*

$$\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models Final(\Sigma\delta, end[\sigma]).$$

However, if goal `final(`$\Sigma\delta, end[\sigma]$`)` *finitely fails then*

$$\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models \neg Final(\Sigma\delta, end[\sigma]).$$

The following Corollary generalizes Theorem 8.5 for *IndiGolog* programs.

COROLLARY 8.7 (SOUNDNESS OF $Trans$ AND $Final$). *Let δ be a bounded program without free variables, and let $\mathcal{D}$ be the underlying GAT. Let σ be a history, and let P and H be Prolog variables. Assume sensors are available up to history σ and the program used is T_D. Then, the goals* `trans(`$\delta, end[\sigma], P, H$`)` *and* `final(`$\delta, end[\sigma]$`)` *on T_D are sound w.r.t. $\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma]$. Moreover, if* `trans/4` *succeeds both P and H are bound to ground terms.*

Finally, we want to study sufficient conditions under which we can successfully evaluate tests in *IndiGolog* programs. To do that, we must recall the concept of JIT-histories for programs (Definition 6.6).

THEOREM 8.8 (WEAK COMPLETENESS OF $Trans$ AND $Final$). *Let σ be a JIT-history for a bounded program δ; let $\mathcal{D}$ be the underlying acyclic GAT; and assume that sensors outcomes are available up to σ and the program used is T_D.*

(1) *The goal* `trans(`$\delta, end[\sigma], P, S$`)` *succeeds with $P = \delta'$ and $S = s'$ or does not terminate*[16] *whenever*

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

(2) *The goal* `final(`$\delta, end[\sigma]$`)` *succeeds whenever*

$$\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models Final(\delta, end[\sigma]).$$

So for bounded programs we have a sound and a weakly complete implementation of $Trans$ and $Final$. We observe that nontermination can only arise due to a search. Otherwise, termination of `trans/4` and `final/3` can be guaranteed.

9. CONCLUSIONS

In the present work, we have presented a new high-level programming language of the *Golog* family with two main characteristics: (i) the execution framework is more realistic than previous languages for large programs and complex scenarios where sensing information can be suitably exploited; (ii) the background theory allows for more open-world theories where causal laws need not be complete. We consider *IndiGolog* as a new step toward a practical programming language for agents.

The *IndiGolog* interpreter described here is provably correct over guarded action theories in open-world settings, but, as seen, relies on a number of assumptions. One serious limitation concerns the use of bounded programs, which would be inconvenient when dealing with large sets of elements that are added and deleted dynamically and cannot be enumerated. For instance, we may have a fluent $Student(x, s)$ denoting that x is a student in situation s and such that students enroll and withdraw dynamically. In such cases, we clearly need an evaluation procedure that

[16] Nontermination may arise because of an infinite sequence of legal transitions.

works with free variables. However, this is not sufficient to guarantee soundness of $Trans$ and $Final$. To do so, either we need to drop the prioritized concurrency construct, or use an evaluation that is complete over free variables. One way might be to add *constraints* as a new argument to `eval`, `trans`, and `final`. These procedures would say things like "everything but John, Mark, and Peter is not a student" or "there is a transition whenever X is not Mark." With this, we would have a sound and sometimes complete implementation of $Trans$ and $Final$, while keeping the ability to express open formulas.

Last, but not least, we observe that the search operator defined here essentially returns the next action to execute that is guaranteed to be part of a successful execution. Alternatively one may define a search operator that actually returns a new program ready to be executed online and containing no search. Such a program should resemble a conditional plan and constitute a reasoned course of action leading to a successful execution as sensing information is gathered. The work on sGolog [Lakemeyer 1999] as well as that on general forms of planning [Levesque 1996] may give some insight on how to tackle this issue. The point is that in one way or another in order to exploit nondeterminism we also need a form of lookahead like that provided by the search operator.

A. PROOFS

A.1 Occur-Check and Floundering Free Proofs

We will need the following terminology: A *mode* for an n-ary predicate symbol p is a function $m_p : \{1, ..., n\} \rightarrow \{+, -\}$. Positions mapped to $'+'$ are called *input* positions of p, and positions mapped to $'-'$ are called *output* positions of p. Intuitively, queries formed by predicate p will be expected to have input positions occupied by ground terms. We write m_p in the form $p(m_p(1), ..., m_p(n))$. A family of terms is *linear* if every variable occurs at most once on it. A clause is (input) output linear if the family of terms occurring in all (input) output positions of its body is linear.

An *input-output specification* for a program P is a set of modes, one for each predicate symbol in P. A clause (goal) is *well-moded* if every variable occurring in an input position of a body goal occurs either in an input position of the head, or in an output position of an earlier body goal; and every variable occurring in an output position of the head occurs in an input position of the head, or in an output position of a body goal. A goal can be viewed as a clause with no head and we will be interested only in goals with one atom, i.e. $G = \leftarrow A$. A program is called *well-moded* w.r.t. its input-output specification if all its clauses are. The definition of well-moded program constrains "the flow of data" through the clauses of the program. Lastly, a clause (goal) is *strictly moded* if it is well-moded and output linear, and a program is *strictly moded* if every rule of it is.

It was proved in Apt and Pellegrini [1994, Corollary 4.5] that well-moded and output linear programs (for some input-output specification) are occur-check free w.r.t. well-moded goals. It was also proven there (Corollary 6.5) that a program P is occur-check free w.r.t. a goal G if both P and G are strictly moded.

LEMMA A.1. *Program P_D for any domain description D is occur-check free w.r.t. queries of the form $G =$`eval(`ϕ, h, B`)` where ϕ and h are ground terms*

and B is a variable.

PROOF. We are to prove that both P_D and G are *strictly moded*. To do that we select the following mode, and we assume that `has_value/4`, `sub_v/4`, and `rel/4` are all correctly implemented.

```
      eval(+,+,-), alltrue(+,+,+,+), varsort(+,-), domain(+,-)
 rel(+,-,-,-), has_value(+,+,+,-), evalfalse(+,+), has_sensor(+,-,+)
               subv(+,+,+,-), subf(+,-,+), bool(+,-,+)
fluent(+), sensor(+), init(+), closed(+), gsfa(+,-,-), gssa(+,+,-,-)
```

In this mode, P_D is both well moded and strictly moded. Moreover, the goal G is both well moded and strictly moded (ϕ and h are ground terms). Hence, by a generalization of Corollary 6.5 in Apt and Pellegrini [1994] for general programs, P_D is occur-check free for goal G. □

LEMMA A.2. *A query $G =$`eval`(ϕ, h, B) to P_D does not flounder, i.e., the LDNF-tree of G does not have nodes marked by flounder.*

PROOF. Here we appeal to Theorem 8.5 in Apt and Pellegrini [1994]: if P_D and G are well moded and all predicate symbols occurring under *not* in P_D and G are moded completely input, then $P_D \cup \{G\}$ does not flounder. The only predicate symbols occurring under *not* in P_D are `init/1`, `sensor/1`, `evalfalse/2`, and `alltrue/4` which are all moded completely input. □

LEMMA A.3. *Program T_D, for any domain description D, is occur-check free w.r.t. queries of the form $G1 =$`trans`(δ, h, E, H) and $G2 =$`final`(δ, h) where ϕ and h are ground terms and E, H are variables.*

PROOF. We extend the input-output specification given in the proof of Lemma A.1 as follows:

```
        trans(+,+,-,-), final(+,+), existtrans(+,+), =(+,-)
  evalsec(+,+,-), path(+,+), write(+), poss(+,-), prim_action(+)
```

By inspecting each rule, we can verify that T_D is well moded in this extended mode. Also, every clause of T_D is output linear, and goals $G1$ and $G2$ are, clearly, well moded. Hence, all the conditions for Apt-Pellegrini's Corollary 4.5 are satisfied, and T_D is occur-check free w.r.t. goals $G1$ and $G2$. □

LEMMA A.4. *The queries `trans`(δ, h, E, H) and `final`(δ, h) on program T_D, where δ and h are ground terms and E, H are variables, do not flounder.*

PROOF. The only predicates occurring under *not* are `existtrans/2` and `=/2`. First, `existtrans/2` is moded completely input. Second, whenever `=/2` is used under *not*, its output position is grounded. □

A.2 Soundness and Completeness Proofs

For this section, we need the following terminology. A mapping from ground atoms of a general logic program P into the set of natural numbers is called a *level mapping* of P. We use *ground*(P), A, and L_i $(i > 0)$ to refer to the set of all variable-free instances of the clauses in P, an atom, and an atom or an atom preceded by *not* respectively.

A program P is *acceptable* if there exists a level mapping $|\cdot|$ and a model I of P, such that for every clause $A \leftarrow L_1, ..., L_n \in ground(P)$ and any $0 < i < n$ it is the case that $I \not\models L_1, ..., L_i$ or $|A| > |L_{i+1}|$.[17] A program P is *acyclic* if there is a level mapping $|\cdot|$ such that for every clause $A \leftarrow L_1, ..., L_n \in ground(P)$ it is the case that $|A| > |L_{i+1}|$. It can be proved that every acyclic programs is acceptable. Finally, a goal $G = \leftarrow A$ is *bounded* w.r.t. a level mapping $|\cdot|$ if there exists a number l such that $l \geq |[A]|$ where $|[L]|$ denotes the maximum that $|\cdot|$ takes on the set $[A]$ of variable-free instances of A. If $|[A]|$ is constant, G is called *rigid*, and G is bounded.

Apt and Pedreschi [1991, Theorem 4.1] proved that every SLDNF-tree (including, of course, all LDNF-trees) for a bounded goal G on an acyclic program P is finite. Later, Apt and Pedreschi [1993, Corollary 4.11] showed that if P is an acceptable program and G is a bounded goal, then all LDNF-derivations for goal G w.r.t. program P are finite, and therefore the Prolog interpreter terminates on G.

For the following proofs, we rely on the Lemmas A.1, A.2, A.3, and A.4 to guarantee that all programs used are *occur-check free and floundering free* for the corresponding goals.

Proof of Theorem 8.1. Assume that `eval(`$\phi, end[\sigma], B1$`)` succeeds with $B1 = b$. We prove that `eval/3` is sound w.r.t. $\mathcal{D}^{end[\sigma']}$ by induction on the number of calls to `eval/3` in the finite LDNF-tree. We allow σ' to be a subhistory of σ or the converse.

Base Case (only one call to `eval/3`, the goal call). $\phi(s)$ can only be a ground atomic sensor formula, such us $sonar(s) > 20$, or a relational fluent at the initial history.

In the former case, we have to distinguish two different situations: (i) when the sensor formula mentions no sensing function; (ii) when the sensor formula mentions at least one sensing function. An example of (i) is when two action terms are compared (such as $openDoor(3) = walk$), in which case the soundness of `eval/3` is easily justified by virtue of Clark's Theorem [Clark 1978]. In the second case, the evaluation of the sensor formula succeeds only if $end[\sigma] \leq end[\sigma']$ due to the fact that `has_sensor/3` finitely fails for any history extending σ'. Therefore, `eval/3` replaces the sensing functions in $\phi(s)$ by their values at σ–which are all available– and soundness follows as well.

Finally, when the formula is a fluent and the history is the initial one, soundness is justified because of the clauses `init/1` implementing $\mathcal{D}_0$.

Induction Step (more than one call to `eval/3`). Here, a new induction on the structure of the formula $\phi(s)$ is performed. The base case can only arise when $\phi(s)$ is a fluent and the history is not the initial one. In such a case, `eval/3` reduces to `bool/3`. This means that either the third or fourth rule of `bool/3` has to succeed. In the latter case, we apply the (first) induction hypothesis on the evaluation of the guard and the condition of the GSSA selected to get soundness

[17] More precisely, the model I has to satisfy some other conditions to deal properly with negated atoms. However, since we do not take advantage of such restrictions through our proofs, we do not make them explicit.

for the evaluation of the fluent. On the other hand, when the third clause of `bool/3` applies, the induction hypothesis over the formulas of the selected GSFA is used. In such a case, it is worth pointing out that if the evaluation of the sensor formula ρ corresponding to the GSFA in question succeeds, then the evaluation is being done on a history not further than σ', i.e., $end[\sigma] \leq end[\sigma']$.

Recall that sensor outcomes are only available up to history σ' and that sensor formulas in GSFAs are built from atomic formulas mentioning at least one sensing function each. To end, the induction step when the formula is not an atomic one is performed using the first seven rules for `eval/3` in which a complex formula is decomposed up to its atomic components. Clearly, at each step the number of calls to `eval/3` is reduced, and the (first) induction hypothesis can be applied. □

Proof of Corollary 8.2. Direct from Theorem 8.1 and the only clause for `evalsec/3`. □

Proof of Theorem 8.3. We prove this theorem in two parts:

(1) First, we prove that whenever $\mathcal{D}$ is acyclic Prolog always terminates[18] on goal $G' =$`eval(`ϕ, h, B`)`, i.e., the LDNF-tree for G' is finite. Although the program is mainly acyclic in the sense of Apt and Bezem [1991], we are forced to use the more subtle termination condition for *acceptable* programs [Apt and Pedreschi 1993]. We now define the level mapping $|\cdot|$ for the program P_D, starting with the mapping for formulas and sequence of actions:

$|$`and(p1,p2)`$| = |p1| + |p2| + 1$
$|$`neg(p)`$| = |p| + 1$
$|$`some(v,p)`$| = |$`some(v,d,p)`$| = |p| + 3$
$|f| = level(f) + 3$ for each ground fluent f
$|a| = |[]| = 1; |[a|r]| = |r| + 1$ for each ground action term a
$|t| = 0$ for all other ground atomic formulas t (including sensing functions)

where $level(f)$ stands for the highest level mapping among the guards and sensor formulas mentioned in $GSFA(f)$. Formally,

$$level(f) = max \left\{ \bigcup_{i=1}^{|GSFA(f)|} \{|\beta_i^f|, |\rho_i^f|\} \right\}$$

where β_i^f is the guard of any ground instance of the ith GSFA for fluent f (with the situation variable suppressed), and ρ_i^f is its corresponding sensor formula. *It is very important to remark that the above level mapping for formulas is well defined because the theory of action $\mathcal{D}$ is acyclic, and therefore, the dependency relation $\prec$ between fluents is well founded.* Next, we define the level mapping to the remaining atoms. For that, we first define a constant $\mathcal{M}$ for the GAT $\mathcal{D}$ denoting the highest level mapping among all the formulas involved in GSFAs and GSSAs in the theory.

[18] With termination we mean either succeeding or finitely failing. In other words, we do not consider floundering or aborting as (legal) termination.

Formally,

$$\mathcal{M} = max\left\{\bigcup_{j=1}^{\mathcal{F}}\left[\bigcup_{i=1}^{|GSFA(f_j)|}\{|\beta_i^{f_j}|,|\rho_i^{f_j}|\}\cup\bigcup_{i=1}^{|GSSA(f_j)|}\{|\alpha_i^{f_j}|,|\gamma_i^{f_j}|\}\right]\right\}$$

where $\mathcal{F}$ denotes the number of fluent names in the theory, $\beta_i^{f_j}$ ($\alpha_i^{f_j}$) is the guard of any ground instance of the ith GSFA (GSSA) (with the situation variable suppressed) of the jth fluent, and $\rho_i^{f_j}$ ($\gamma_i^{f_j}$) is the corresponding sensor-formula (fluent-formula) for such axiom. We complete the level mapping for P_D as follows:

$|\texttt{eval(p,h,b)}| = |p| + |h|(\mathcal{M}+3) + 2$
$|\texttt{alltrue(p,v,d,h)}| = (|p|+2) + |h|(\mathcal{M}+3) + 2$
$|\texttt{evalfalse(p,h)}| = (|p|+1) + |h|(\mathcal{M}+3) + 2$
$|\texttt{subf(p,q,h)}| = |p| + |h|(\mathcal{M}+3) + 1$
$|\texttt{bool(p,v,h)}| = |p| + |h|(\mathcal{M}+3)$
$|\texttt{gsfa(f,g,c)}| = |\texttt{gssa(a,f,g,c)}| = |\texttt{subv(v,o,p,q)}| = 1$
$|t| = 0$ for all other atoms

It is possible to show that program P_D is acceptable w.r.t. the above level mapping and some model I. What is more, the (stronger) acyclic condition [Apt and Bezem 1991] is satisfied for all the clauses except for the last two of `bool/3`. For instance, for the third and the last two rules for `eval/3` we have

$$\begin{aligned}
|\texttt{eval(and(p1,p2),h,true)}| &= |and(p1,p2)| + |h|(\mathcal{M}+3) + 2\\
&> |p1| + |h|(\mathcal{M}+3) + 2 = |\texttt{eval(p1,h,true)}|\\
&> |p2| + |h|(\mathcal{M}+3) + 2 = |\texttt{eval(p2,h,true)}|\\
|\texttt{eval(p,h,b)}| &= |p| + |h|(\mathcal{M}+3) + 2\\
&> 0 = |\texttt{fluent(p,rel)}|\\
&> |p| + |h|(\mathcal{M}+3) = |\texttt{bool(p,b,h)}|\\
|\texttt{eval(p,h,b)}| &= |p| + |h|(\mathcal{M}+3) + 2\\
&> 0 = |\texttt{rel(p,op,s,v)}| = |\texttt{has_value(op,s1,v1,bool)}|\\
&> |s| + |h|(\mathcal{M}+3) + 1 = |\texttt{subf(s,s1,h)}| \quad (|s| \leq |p|)\\
&> |v| + |h|(\mathcal{M}+3) + 1 = |\texttt{subf(v,v1,h)}| \quad (|v| \leq |p|)
\end{aligned}$$

Finally, the most interesting cases arise with the last two rules for `bool/3`. There, we need to use the condition distinguishing acyclic programs from acceptable ones. Consider now any ground instance $A : -B_1, B_2, B_3.$ of of the last two clauses for `bool/3`. In what follows, we prove two facts which imply that the rule satisfies the appropriate requirements. We do not show here the existence of a model for P_D, say I, but just point out that $I \models gsfa(f,g,p)$ and $I \models gssa(a,f,g,p)$ iff they correspond to some GSFA and GSSA respectively in the domain theory $\mathcal{D}$. Notice that both `gssa/4` and `gsfa/3` are simply two database clauses.

Fact 1: $|A| > |B_1|$. Because $|[]| = 1$ it follows trivially that

$$\begin{aligned}
|\texttt{bool(f,v,h)}| &= |f| + |h|(\mathcal{M}+3) > 1 = |\texttt{gsfa(f,g,p)}|\\
|\texttt{bool(f,v,[a|h])}| &= |f| + |[a|h]|(\mathcal{M}+3) > 1 = |\texttt{gssa(a,f,g,p)}|
\end{aligned}$$

Fact 2: Suppose $I \models B_1$. Then, $|A| > |B_2|$ and $|A| > |B_3|$. For the fourth `bool/3` rule we have the following inequalities:

$$\begin{aligned} |\texttt{bool(f,v,[a|h])}| &= |f| + |h|(\mathcal{M}+3) + (\mathcal{M}+3) \\ &> |g| + |h|(\mathcal{M}+3) + 2 = |\texttt{eval(g,h,true)}| = B_2 \\ |\texttt{bool(f,v,[a|h])}| &= |f| + |h|(\mathcal{M}+3) + (\mathcal{M}+3) \\ &> |p| + |h|(\mathcal{M}+3) + 2 = |\texttt{eval(p,h,v)}| = B_3 \end{aligned}$$

Notice, that given $I \models B_1 = gssa(a, f, g, p)$, g and p are formulas from a GSSA in $\mathcal{D}$, and therefore, $\mathcal{M} \geq |g|$ and $\mathcal{M} \geq |p|$ hold.
Similarly, we have the following inequalities for the last rule of `bool/3`:

$$\begin{aligned} |\texttt{bool(f,v,h)}| &= |f| + |h|(\mathcal{M}+3) = level(f) + 3 + |h|(\mathcal{M}+3) \\ &\geq |g| + 3 + |h|(\mathcal{M}+3) \\ &> |g| + |h|(\mathcal{M}+3) + 2 = |\texttt{eval(g,h,true)}| = B_2 \\ |\texttt{bool(f,v,h)}| &= |f| + |h|(\mathcal{M}+3) = level(f) + 3 + |h|(\mathcal{M}+3) \\ &\geq |p| + 3 + |h|(\mathcal{M}+3) \\ &> |p| + |h|(\mathcal{M}+3) + 2 = |\texttt{eval(p,h,v)}| = B_3 \end{aligned}$$

Again, given that $I \models B_1 = gsfa(f, g, p)$, g and p are formulas corresponding to some GSFA in $\mathcal{D}$. From the definition of $level(f)$ we know that $level(f) \geq |g|$ and $level(f) \geq |p|$. Intuitively, the key point is that due to the fact that $\mathcal{D}$ is acyclic (Definition 3.2), it is possible to build a well-defined level mapping where the level of a fluent is always higher than the level of the guards and sensor formulas mentioned in its GSFAs.

We have shown that program P_D is acceptable w.r.t. the level mapping $|\cdot|$ and a model I. Also, it is easy too check that goal $G' =$`eval(`ϕ, h, B`)` is rigid and hence bounded. Using Corollary 4.11 in Apt and Pedreschi [1993] we argue that the Prolog interpreter terminates on goal G'. Once again, we are relying on the fact that the program is occur-check and floundering free (Lemmas A.1 and A.2).

(2) Next, let us prove, that given that σ is a JIT-history for the formula $\phi(s)$ w.r.t. $\mathcal{D}^{end[\sigma']}$, the call to `eval/3` *does not finitely fail*. This is proved by induction on the number of calls to `eval/4`.

Base Case: Easy given that either the formula being evaluated is an atomic sensor formula or a relational fluent at the initial history.

Induction Step: Assume the theorem holds whenever the tree has n or less calls. There are two cases: (i) the formula is a compound one; (ii) the formula is a relational fluent. The first case is straightforward because the formula is decomposed into simpler formulas and such that the history remains JIT for each subdivision.

The interesting case is when the formula is a relational fluent $F(\vec{t}, s)$ and σ is JIT for for it. Thus, there has to be an applicable GSFA or GSSA in the sense of the JIT-history definition (Definition 6.5). When a GSFA is applicable, the induction hypothesis can be applied to the third rule of `bool/3`. Similarly, when a GSSA is applicable, the induction hypothesis can be applied to the fourth rule of `bool/3`.

In either case, `bool/3` will succeed for the relational fluent $F(\vec{t}, s)$ at history σ, and so will the call to `eval/3`.

By (1) and (2), goal `eval(`$\phi, end[\sigma], B$`)` succeeds with computed answer $B = true$ or $B = false$. □

PROOF OF COROLLARY 8.4. Direct from Theorem 8.3 and the only clause for `evalsec/3`. □

PROOF OF THEOREM 8.5. By induction on the number of calls to `trans/4` and `final/2`.

Base Case: For `trans/4`, the program is either a primitive action or a check condition; for `final/2` the program is an empty program or a nondeterministic iteration of a program. Take the first case for `trans/4` for which the following rule of T_D is used:

```
trans(E,H,[],[E1,H]):- prim_action(E1), poss(E1,P),
                        evalsec(P,H,true)
```

The first two subgoals refer to database relations to retrieve the precondition axiom for the primitive action. The success of the third subgoal implies, by Corollary 8.2, that the action is executable. Therefore `trans/4` is sound, and both [] and $[E1|H]$ are ground given that $E1$ and H are bound to ground terms. On the other hand, if `trans/4` happens to finitely fail for a primitive action, then its precondition must be false; for, the third subgoal must have finitely failed (Corollary 8.4), and from that, it is easy to see that `evalsec(P,H,false)` would succeed.

Induction Step: Here the LDNF-tree has more than one call, and the program is a complex one. For all constructs except prioritized concurrency, the proof is simple and relies on Theorem 8.1 whenever a formula is evaluated via `evalsec/3`.

The interesting cases happen when `trans/4` and `final/2` finitely fail and the case of prioritized concurrency. For the latter, if `trans/4` succeeds on a program $\delta_1 \rangle\rangle\, \delta_2$, it has to be the case that either: (i) `trans/4` succeeds on δ_1; or (ii) `existtrans/2` finitely fails on δ_1, but `trans/4` succeeds on δ_2. For (i), the induction can be applied easily. For (ii), the finite failure of `existtrans/2` allows us to state that there *exists no transition* for δ_1. In addition, the success of `trans/4` on δ_2 together with the induction hypothesis gives us the soundness for the whole program $\delta_1 \rangle\rangle\, \delta_2$.

Finally, let us see a case where `trans/4` finitely fails. Suppose then that `trans/4` finitely fails on program $?(\phi)$ at history σ. By inspecting the only eligible clause we know that `evalsec(`$\phi, end[\sigma], true$`)` must have finitely failed. By the same argument as above, it must be the case that $\phi(s)$ is false at σ, and the soundness of `trans/4` follows. Notice that if $\phi(s)$ had been unknown at σ, the call to `evalsec` would have aborted to the top level instead of finitely failing. □

PROOF OF LEMMA 8.6. Assume that goal `trans(`$search(\delta), end[\sigma], P, S$`)` succeeds with $P = search(\delta')$ and $S = s'$. This means that `trans(`$\delta, end[\sigma], P1, S$`)` succeeds with $P1 = \delta'$ and $S = s'$, and it follows from Theorem 8.5 that $\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models Trans(\delta, end[\sigma], \delta', s')$ where δ' and s' do not contain any free variables.

In addition, the goal `path(`δ', s'`)` must succeed with sensors available up to σ,

which implies a sequence of zero or more successful calls to `trans/4` ended by a successful call to `final/2`.

—If the length of the sequence is zero, then `final(`δ', s'`)` succeeds with sensors available up to σ. By Theorem 8.5 (δ' has no search), $\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models Final(\delta', end[\sigma])$, and hence $\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models \exists s''.Do(\delta', end[\sigma], s'')$.

—Assume the length of the sequence is $n+1$. Then, `trans(`$\delta', end[\sigma], E1, H1$`)` succeeds with $E1 = \delta'', H1 = s''$ with sensors available up to history σ, and by Theorem 8.5, $\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models Trans(\delta', end[\sigma], \delta'', s'')$. Moreover, `path(`$\delta'', s''$`)` succeeds in n steps with sensors available up to σ. By induction, $\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models \exists s'''.Do(\delta'', s'', s''')$ and $\mathcal{D}^{end[\sigma]} \cup \mathcal{C} \cup Sensed[\sigma] \models \exists s''.Do(\delta', end[\sigma], s'')$ follows.

The soundness of `final/2` w.r.t. *Final* is obtained directly from Theorem 8.5. □

Proof of Corollary 8.7. It follows directly from the soundness of programs not containing search (Theorem 8.5) and the soundness of search itself (Lemma 8.6). □

Proof of Theorem 8.8. We divide this proof into two parts:

(A) We start by proving that whenever σ is a JIT-history for program δ not containing search w.r.t. $\mathcal{D}^{end[\sigma']}$ ($end[\sigma'] \sqsubseteq end[\sigma]$), the Prolog interpreter always terminates on goals $G1 =$`trans(`$\delta, end[\sigma], P, S$`)` and $G2 =$`final(`$\delta, end[\sigma]$`)`. Let us first concentrate on program $T' = T_D - P_D - S$ where S is the set of the four rules for search (one for `trans/4`, one for `final/2`, and two for `path/2`).

Fortunately, we can appeal here to the acyclicity condition from Apt and Pedreschi [1993]. Given a (ground) program term p, $noactions(p)$ ($nocons(p)$) is the number of primitive (complex) actions mentioned in p. We define the following level mapping $|\cdot|$ for the program T' as follows:

$|\mathtt{p}| = noactions(p) + nocons(p)$
$|\mathtt{trans}(\mathtt{p},\mathtt{h},\mathtt{p}',\mathtt{h}')| = |p| + 2$
$|\mathtt{final}(\mathtt{p},\mathtt{h})| = |p|$
$|\mathtt{existtrans}(\mathtt{p},\mathtt{h})| = |p| + 1$

All the other atoms are mapped to zero. It can be verified that program T' is acyclic and that goals of the form $G1$ and $G2$ are rigid, and hence bounded, w.r.t. the above level mapping. We can generalize this result to the complete program T_D by giving two simple arguments: (i) we can safely include the set of rules S, i.e., all the rules implementing search, given that δ has no search, and hence, no clause in S will be used; and (ii) we can safely include the whole program P_D, i.e., the code for the evaluation procedure plus the domain description, because whenever a formula $\phi(s)$ needs to be evaluated via `evalsec/3`, Corollary 8.4 applies in other words, `evalsec/3` would eventually terminate legally. Again, Lemmas A.3 and A.4 assure our programs and goals are occur-check and floundering free.

(B) Next, a weak-completeness result is proved for search itself. Assume a program of the form $\Sigma\delta$ where δ mentions no search, and a history σ such that sensor outcomes are available up to it and such that σ is a JIT-history for program $\Sigma\delta$.

The hard part is to realize that if `trans(`$\delta, end[\sigma], P1, S1$`)` succeeds with $P1 = \delta'$ and $S1 = s'$, the goal $G =$`path(`δ', s'`)` either terminates or loops forever. Suppose

that `path/2` has already performed a sequence of n consecutive transitions and is querying for the goal `path(`δ''`,`s''`)` once again for some δ'' and s''. From soundness of `trans/4` (Theorem 8.5),

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

for any extension σ'' of σ such that $end[\sigma''] = s''$. What is more, σ'' is a JIT-history for program δ'' w.r.t. $\mathcal{D}^{end[\sigma]}$. Under this circumstance and the fact that δ'' has no search, we know, by part A, that both goals `final(`δ''`,`s''`)` and `trans(`δ''`,`s''`,`P`,`S`)` terminate legally (either by finitely failing or succeeding). It follows then that the goal `path(`δ'`,`s'`)` either terminates legally (by finitely failing or succeeding) or loops forever, and as a result, so does the original goal `trans(`$\Sigma\delta, end[\sigma], P, S$`)`.

Putting parts A and B together, and assuming no nested searches, the weak-completeness result for general *IndiGolog* programs follows. □

REFERENCES

Apt, K. R. and Bezem, M. 1991. Acyclic programs. *New Generation Computing 9*, 335–363.

Apt, K. R. and Pedreschi, D. 1993. Reasoning about termination of pure Prolog programs. *Information and Computation 106,* 1, 109–157.

Apt, K. R. and Pellegrini, A. 1994. On the occur-check free Prolog program. *ACM Toplas 16,* 3, 687–726.

Baral, C. and Son, T. C. 1997. Approximate reasoning about actions in presence of sensing and incomplete information. In *International Logic Programming Symposium (ILSP' 97)*, J. Maluszynski, Ed. MIT Press, Port Je erson, NY, 387–401.

Baral, C., Gelfond, M., and Provetti, A. 1997. Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming 31*, 210–244.

Best, E. 1996. *Semantics of Sequential and Parallel Programs.* Prentice Hall International Series in Computer Science.

Clark, K. 1978. Negation as failure. In *Logic and Data Base*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 292–322.

De Giacomo, G. and Levesque, H. 1999a. An incremental interpreter for high-level programs with sensing. In *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, H. J. Levesque and F. Pirri, Eds. Springer, Berlin, 86–102.

De Giacomo, G. and Levesque, H. 1999b. Projection using regression and sensors. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI).* Stockholm, Sweden, 160–165.

De Giacomo, G., Lespérance, Y., and Levesque, H. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence 121,* 1–2, 109–169.

De Giacomo, G., Reiter, R., and Soutchanski, M. 1998. Execution monitoring of high-level robot programs. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98).* 453–465.

Golden, K., Etzioni, O., and Weld, D. 1996. Planning with execution and incomplete information. Tech. Rep. UW-CSE-96-01-09, Department of Computer Science and Engineering. University of Washington. April.

Hennessy, M. 1990. *The Semantics of Programming Languages.* John Wiley & Sons, Chichester, England.

Jonsson, P. and Backstrom, C. 1995. Incremental planning. In *New Directions in AI Planning: EWSP'95—3rd European Workshop on Planning*, M. Ghallab and A. Milani, Eds. IOS Press, Assisi, Italy, 79–80.

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

Kowalski, R. A. and Sadri, F. 1999. From logic programming to multi-agent systems. *Annals of Mathematics and Artificial Intelligence 25,* 3–4, 391–419.

Kowalski, R. A. and Sergot, M. J. 1986. A logic-based calculus of events. *New Generation Computing 4*, 67–95.

Lakemeyer, G. 1999. On sensing and off-line interpreting in Golog. In *Logical Foundations for Cognitive Agents, Contributions in Honor of Ray Reiter*. Springer, Berlin, 173–187.

Levesque, H. 1996. What is planning in the presence of sensing? In *The Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI-96*. American Association for Artificial Intelligence, Portland, Oregon, 1139–1146.

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

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

Miller, R. and Shanahan, M. 1994. Narratives in the situation calculus. *The Journal of Logic and Computation 4,* 5, 513–530.

Nielson, H. R. and Nielson, F. 1992. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Chichester, England.

Plotkin, G. 1981. A structural approach to operational semantics. Tech. Rep. Technical Report DAIMI-FN-19, Computer Science Department Aarhus University Denmark.

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed. Academic Press, San Diego, CA, 359–380.

Reiter, R. 2000. Narratives as programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)*. Morgan Kaufmann, San Francisco, CA.

Shanahan, M. 1999. What sort of computation mediates best between perception and action? In *Logical Fundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, H. Levesque and F. Pirri, Eds. Springer-Verlag, 352–368.

Waldinger, R. 1977. Achieving several goals simultaneously. *Machine Intelligence 8*, 94–136.

Received August 2000; revised February 2001; accepted March 2001