ElGolog: A High-Level Programming Language with Memory of the Execution History

Giuseppe De Giacomo
Sapienza Università di Roma
Roma, Italy
degiacomo@diag.uniroma1.it

Yves Lespérance
York University
Toronto, Canada
lesperan@eecs.yorku.ca

Eugenia Ternovska
Simon Fraser University
Burnaby, BC Canada
ter@sfu.ca

Abstract

Most programming languages only support tests that refer exclusively to the current state. This applies even to high-level programming languages based on the situation calculus such as Golog. The result is that additional variables/fluents/data structures must be introduced to track conditions that the program uses in tests to make decisions. In this paper, drawing inspiration from McCarthy's Elephant 2000, we propose an extended version of Golog, called ElGolog, that supports rich tests about the execution history, where tests are expressed in a first-order variant of two-way linear dynamic logic that uses ElGolog programs with converse. We show that in spite of rich tests, ElGolog shares key features with Golog, including a sematics based on macroexpansion into situation calculus formulas, upon which regression can still be applied. We also show that like Golog, our extended language can easily be implemented in Prolog.

Introduction

Suppose that we want to give the following instructions to a robot assistant:

> *You have already delivered coffee to some offices. Now, go back and deliver milk as well.*

If we want to write a high-level program to represent these instructions, we might write the following:

$$\textbf{while } \exists o.(DeliveredCoffee(o) \wedge \neg DeliveredMilk(o))$$
$$\textbf{do } \pi o.[(DeliveredCoffee(o) \wedge \neg DeliveredMilk(o))?;$$
$$goto(o)\ ;\ deliverMilk(o)]$$

i.e., while there is an office where coffee has been delivered but not milk, (nondeterministically) pick such an office and go there and deliver milk. This program is written in Golog (Levesque et al. 1997), but there is nothing essential about this; any language where one can iterate over items that satisfy a condition would do.

Notice that the program relies on fluents (essentially, boolean functions that change value over time) such as $DeliveredCoffee(o)$ and $DeliveredMilk(o)$ that memorize historical information. The model of the world that captures the dynamics of the domain (i.e., the action theory in Golog) may not support these fluents. All it is likely to say is that immediately after delivering an item to a location, the item will be at this location. To be able to write the above program, the programmer would first need to extend the action theory/world model with these additional task-specific fluents. Such an extension might be needed for every new task.

More generally, most programming languages only support queries about the current state of the system. This is the so-called Markov property: the executability of an action and its effects are entirely determined by the current state or situation. This assumption is commonly made in formalisms for reasoning about action, such as the situation calculus (McCarthy and Hayes 1969; Reiter 2001). It is also made in the high-level programming languages Golog (Levesque et al. 1997) and ConGolog (De Giacomo, Lespérance, and Levesque 2000) based on it (as well as in any standard programming language such as C, Java, etc.), where tests may only query the current situation/state. If one wants to query some historical information, one has to introduce some data structures or fluents to remember what happened in the past. In contrast, in natural language instructions, one simply refers to past states and events.

It should be clear that referring to the past to specify a task is useful in many applications, for instance, in robotics. We might want to order the robot to "go back to all rooms where you saw discarded pop cans and bring them to the recycling bin". Note that bringing milk to people to whom one has already delivered coffee is not necessarily just a "fix" to the earlier behavior, as the milk may only have been ordered later, or it might not have been possible to deliver both at the same time (perhaps the milk is stored in a fridge far from the coffee machine).

In this paper, we show that we can conveniently extend a programming language to support queries about the past. In particular, we define ElGolog, which extends Golog with the possibility of using (converse) programs in tests, as sometimes allowed in Dynamic Logic (Harel, Kozen, and Tiuryn 2000). However we interpret such tests on the current history, or *log*, which is provided by the current situation. So in a sense we are equipping Golog with a sort of two-way version of Linear Dynamic Logic operators (De Giacomo and Vardi 2013).

With this addition, we can write programs that capture more directly the natural language instructions. For exam-

ple, we would write:

$$\begin{array}{l}\textbf{while } \exists o.\langle(deliverCoffee(o); any^*)^-\rangle \\ \qquad\qquad (\neg\langle any^*; deliverMilk(o)\rangle True) \\ \textbf{do } \pi o.[\langle(deliverCoffee(o); any^*)^-\rangle \\ \qquad (\neg\langle any^*; deliverMilk(o)\rangle True)?; \\ \qquad goto(o)\ ;\ deliverMilk(o)]\end{array}$$

The test in this example means that there is an office location o such that there is a state in the past where $deliverCoffee$ has occurred for that o, and, in the future of that state, there is no $deliverMilk$ for o. Here, any means any atomic action, which can be defined as $(\pi\vec{x}.A_1(\vec{x})) \mid \ldots \mid (\pi\vec{x}.A_k(\vec{x}))$, assuming that $A_1, \ldots, A_k$ are all the action types/functions.[1]

John McCarthy made a similar point to motivate his proposal for a programming language Elephant 2000, which, among other features, "does not forget" (McCarthy 1992; 2007). ElGolog is named after this. We go over other related work in the discussion section.

Preliminaries

Situation Calculus. The *situation calculus* is a well known predicate logic language for representing and reasoning about dynamically changing worlds (McCarthy and Hayes 1969; Reiter 2001). All changes to the world are the result of *actions*, which are terms in the language. A possible world history is represented by a term called a *situation*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol do, such that $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Predicates and functions whose value varies from situation to situation are called *fluents*, and are denoted by symbols taking a situation term as their last argument (e.g., $Holding(x, s)$). $s \sqsubset s'$ means that s is a predecessor situation to s', and $s \sqsubseteq s'$ stands for $s \sqsubset s' \vee s = s'$. The abbreviation $do([a_1, \ldots, a_n], s)$ stands for $do(a_n, do(a_{n-1}, \ldots do(a_1, s) \ldots))$.

Within the language, one can formulate action theories that describe how the world changes as a result of the available actions. A *basic action theory (BAT)* $\mathcal{D}$ (Pirri and Reiter 1999; Reiter 2001) is the union of the following disjoint sets: the foundational, domain independent, axioms of the situation calculus (Σ), which include a second order axiom characterizing the situation domain; precondition axioms stating when actions can be legally performed ($\mathcal{D}_{poss}$); successor state axioms describing how fluents change between situations ($\mathcal{D}_{ssa}$); unique name axioms for actions ($\mathcal{D}_{una}$); and axioms describing the initial configuration of the world ($\mathcal{D}_{S_0}$). A special predicate $Poss(a, s)$ is used to state that action a is executable in situation s; precondition axioms in $\mathcal{D}_{poss}$ characterize this predicate. $Executable(s)$ means that every action performed in reaching situation s is executable in the situation in which it occurs. In turn, successor state axioms encode the causal laws of the world being modeled; they take the place of the so-called effect axioms and provide a solution to the frame problem.

A key feature of BATs is the existence of a sound and complete *regression mechanism* for answering queries about situations resulting from performing a sequence of actions (Pirri and Reiter 1999; Reiter 2001). In a nutshell, the regression operator $\mathcal{R}^*$ reduces a formula ϕ about a particular future situation to an equivalent formula $\mathcal{R}^*[\phi]$ about the initial situation S_0, essentially by substituting fluent relations with the right-hand side formula of their successor state axioms. A formula ϕ is *regressable* if and only if *(i)* all situation terms in it are of the form $do([a_1, \ldots, a_n], S_0)$, *(ii)* in every atom of the form $Poss(a, \sigma)$, the action function is specified, i.e., a is of the form $A(t_1, \ldots, t_n)$, *(iii)* it does not quantify over situations, and *(iv)* it does not contain $\sqsubset$ or equality over situation terms. Thus in essence, a formula is regressable if it does not contain situation variables. Another key result about BATs is the relative satisfiability theorem (Pirri and Reiter 1999; Reiter 2001): $\mathcal{D}$ is *satisfiable* if and only if $\mathcal{D}_{S_0} \cup \mathcal{D}_{una}$ is satisfiable (the latter being a purely first-order theory). This implies that we can check if a regressable formula ϕ is entailed by $\mathcal{D}$, by checking if its regression $\mathcal{R}^*[\phi]$ is entailed by $\mathcal{D}_{S_0} \cup \mathcal{D}_{una}$ only.

Golog. To represent and reason about complex actions or processes obtained by suitably executing atomic actions, various so-called *high-level programming languages* have been defined such as Golog (Levesque et al. 1997; Reiter 2001), which includes the usual procedural programming constructs and constructs for nondeterministic choices.[2]

The syntax of Golog programs is as follows:

$$\begin{array}{rcl}\delta & ::= & a \mid (\delta_1; \delta_2) \mid (\delta_1|\delta_2) \mid \pi z.\delta \mid \delta^* \mid \varphi? \\ \varphi & ::= & P(\vec{x}) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\varphi\end{array}$$

In the above, a is an action term, possibly with parameters, and φ is situation-suppressed formula, i.e., a formula with all situation arguments in fluents suppressed. Atomic program a executes the action a, while a test $\varphi?$ confirms that φ holds. The sequence of program δ_1 followed by program δ_2 is denoted by $\delta_1; \delta_2$. Program $\delta_1|\delta_2$ allows for the nondeterministic branch between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* nondeterministic choice of a binding for object variable x (observe that such a choice is, in general, unbounded). δ^* performs δ zero or more times. Conditionals and while loops can be defined as in Dynamic Logic (Harel, Kozen, and Tiuryn 2000). Note the presence of nondeterministic constructs, which allow a loose specification of behavior by leaving "*gaps*" that ought to be resolved by the executor (one can write anything from a generic planning program to a fully deterministic plan/program).

The semantics of Golog programs is specified by defining an abbreviation $Do(\delta, s, s')$, which means that program δ,

[1] Note that tests are used in Golog not just to branch (in conditionals and loops), but also to constrain the nondeterministic choice of arguments of actions/programs within the "pick" construct. The rich tests in ElGolog can be used to select program arguments based on complex historical conditions.

[2] Extensions of Golog include ConGolog (De Giacomo, Lespérance, and Levesque 2000), which supports concurrency, and IndiGolog (Sardiña et al. 2004), which provides means for interleaving sensing, planning, and execution.

when executed starting in situation s, has s' as a legal terminating situation. It is defined inductively as follows:

$$
\begin{array}{lcl}
Do(a, s, s') & \doteq & s' = do(a, s) \wedge Poss(a, s) \\
Do(\delta_1; \delta_2, s, s') & \doteq & \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s') \\
Do(\delta_1|\delta_2, s, s') & \doteq & Do(\delta_1, s, s') \vee Do(\delta_2, s, s') \\
Do(\pi x.\delta, s, s') & \doteq & \exists x.Do(\delta, s, s') \\
Do(\delta^*, s, s') & \doteq & \forall P. \\
 & & [\forall s_1.P(s_1, s_1)] \wedge \\
 & & [\forall s_1, s_2, s_3.Do(\delta, s_1, s_2) \wedge P(s_2, s_3) \supset P(s_1, s_3)] \\
 & & \supset P(s, s') \\
Do(\varphi?, s, s') & \doteq & s' = s \wedge \varphi[s]
\end{array}
$$

In the above definition, we use $\varphi[s]$ to denote the formula obtained from φ by restoring the situation argument s into all fluents in φ. More formally:

$$
\begin{array}{l}
P(\vec{x})[s] \doteq P(\vec{x}, s) \\
(\neg\varphi)[s] \doteq \neg(\varphi[s]) \\
(\varphi_1 \wedge \varphi_2)[s] \doteq \varphi_1[s] \wedge \varphi_2[s] \\
(\exists x.\varphi)[s] \doteq \exists x.\varphi[s]
\end{array}
$$

Thus, for any Golog program δ, $Do(\delta, s, s')$ expands into a situation calculus formula that characterizes the pairs of situations $\langle s, s'\rangle$ over which a complete execution of δ occurs. Note also that for any Golog program δ *without nondeterministic iteration*, $Do(\delta, s, s')$ expands to a first-order regressable formula. Thus one can find executions of such a program though first-order theorem proving. (Levesque et al. 1997; Reiter 2001) presents an interpreter for Golog implemented in Prolog for the case where the initial situation description satisfies the Prolog closed world assumption, which is shown to be sound/correct with respect to the semantics under some assumptions. But note that there is no guarantee of termination in general (e.g., one might have a program of the form "while (true) do ...."). Regarding the π nondeterministic choice of argument construct, if the domain is finite and the regressed queries evaluated by the interpreter succeed or finitely fail, backtracking will exhaust all possible bindings of a π variable; but backtracking on infinitely many instances of a π variable is another case of non-termination. The interpreter searches for a way to resolve the nondeterministic choices in the program that leads to a complete execution. In doing this, it actually interleaves expanding the *Do* definition, regression, and reasoning about the initial situation, to try to detect failures early.

The ElGolog Language

The syntax of ElGolog is same as that of Golog except for tests:

$$
\begin{array}{lcl}
\delta & ::= & a \mid \delta^- \mid (\delta_1; \delta_2) \mid (\delta_1|\delta_2) \mid \pi z.\delta \mid \delta^* \mid \varphi? \\
\varphi & ::= & P(\vec{x}) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\varphi \mid \langle\delta\rangle\varphi
\end{array}
$$

where the *converse* program construct δ^-, which is borrowed from Dynamic Logic (Harel, Kozen, and Tiuryn 2000), *can only be used in tests,* i.e., in the $\langle\delta'\rangle\varphi$ construct.

As we will see, tests are interpreted over the past history up to the current situation, and $\langle\delta^-\rangle\varphi$ holds in the current situation s_n if there exists a situation in the past, say s_t, such that there is an execution of δ from s_t to s_n (i.e., a "backward" execution of δ from s_n to s_t) and where φ held in s_t. Note that the current situation s_n acts as a *log* and $\langle\delta\rangle\varphi$ *queries the log*.

We also define

$$[\delta]\varphi \doteq \neg\langle\delta\rangle\neg\varphi$$

meaning that for all situations reached by executing δ, φ holds.

For all the Golog constructs, *Do* is defined exactly as before, except for tests, where we now have a slightly different definition:

$$Do(\varphi?, s, s') \doteq s' = s \wedge \varphi[s, s]$$

A rich test $\varphi[s_t, s_n]$ is evaluated over a pair of situations, where s_n is the current situation ("now"), i.e., the situation at the end of the log, and s_t is the situation in the past ("then"), where the condition φ is being evaluated. In the definition of $Do(\varphi?, s, s')$ for tests above, both s_n and s_t get initialized to the current situation s, but as we will see the $\langle\delta'\rangle\varphi'$ operator may bind s_t to a different past situation to evaluate φ'.

The semantics of rich tests in general is defined follows:

$$
\begin{array}{l}
P(\vec{x})[s_t, s_n] \doteq P(\vec{x}, s_t) \\
(\neg\varphi)[s_t, s_n] \doteq \neg(\varphi[s_t, s_n]) \\
(\varphi_1 \wedge \varphi_2)[s_t, s_n] \doteq \varphi_1[s_t, s_n] \wedge \varphi_2[s_t, s_n] \\
(\exists x.\varphi)[s_t, s_n] \doteq \exists x.\varphi[s_t, s_n] \\
(\langle\delta\rangle\varphi)[s_t, s_n] \doteq \exists s.Do_{log}(\delta, s_t, s, s_n) \wedge \varphi[s, s_n]
\end{array}
$$

All cases other than $\langle\delta\rangle\varphi$ just recursively obtain the truth value of the formula in terms of that of their subformulas in s_t (as in Golog). $\langle\delta\rangle\varphi[s_t, s_n]$ holds if and only if there exists a situation s, such that there is an execution of δ from s_t to s *within the log*, such that $\varphi[s, s_n]$ holds. It uses the new abbreviation $Do_{log}(\delta, s, s', s_n)$ that holds if and only if there is an execution of δ from s to s' that is entirely within the log s_n. Note that the program δ may contain the converse operator, and Do_{log} needs to handle it.

$Do_{log}(\delta, s, s', s_n)$ is defined inductively as follows:

$$
\begin{array}{lcl}
Do_{log}(a, s, s', s_n) & \doteq & s' = do(a, s) \wedge Poss(a, s) \\
 & & \wedge\, \mathbf{s'} \sqsubseteq \mathbf{s_n} \\
Do_{log}(\delta_1; \delta_2, s, s', s_n) & \doteq & \exists s''.Do_{log}(\delta_1, s, s'', s_n) \wedge \\
 & & Do_{log}(\delta_2, s'', s', s_n) \\
Do_{log}(\delta_1|\delta_2, s, s', s_n) & \doteq & Do_{log}(\delta_1, s, s', s_n) \vee \\
 & & Do_{log}(\delta_2, s, s', s_n) \\
Do_{log}(\pi x.\delta, s, s', s_n) & \doteq & \exists x.Do_{log}(\delta, s, s', s_n) \\
Do_{log}(\delta^*, s, s', s_n) & \doteq & \forall P. \\
 & & [\forall s_1.\mathbf{s_1} \sqsubseteq \mathbf{s_n} \supset P(s_1, s_1, s_n)] \wedge \\
 & & [\forall s_1, s_2, s_3.Do_{log}(\delta, s_1, s_2, s_n) \wedge P(s_2, s_3, s_n) \supset P(s_1, s_3, s_n)] \\
 & & \supset P(s, s', s_n) \\
Do_{log}(\varphi?, s, s', s_n) & \doteq & s' = s \wedge \varphi[s, s_n] \wedge \mathbf{s} \sqsubseteq \mathbf{s_n} \\
Do_{log}(a^-, s, s', s_n) & \doteq & Do_{log}(a, s', s, s_n) \\
Do_{log}((\delta_1; \delta_2)^-, s, s', s_n) & \doteq & Do_{log}((\delta_2)^-; (\delta_1)^-, s, s', s_n) \\
Do_{log}((\delta_1|\delta_2)^-, s, s', s_n) & \doteq & Do_{log}((\delta_1)^-|(\delta_2)^-, s, s', s_n) \\
Do_{log}((\pi x.\delta)^-, s, s', s_n) & \doteq & Do_{log}(\pi x.(\delta)^-, s, s', s_n) \\
Do_{log}((\delta^*)^-, s, s', s_n) & \doteq & Do_{log}(((\delta)^-)^*, s, s', s_n) \\
Do_{log}((\varphi?)^-, s, s', s_n) & \doteq & Do_{log}(\varphi?, s, s', s_n) \\
Do_{log}((\delta^-)^-, s, s', s_n) & \doteq & Do_{log}(\delta, s, s', s_n)
\end{array}
$$

Golog program constructs are handled as in Do, but we additionally ensure that the execution remains within the log s_n (see the parts in bold). For converse programs δ^-, we have various cases according to the form of δ. For the primitive action case, a converse execution of a, i.e., an execution of a^-, occurs between s and s' if a "forward" execution of a occurs between s' and s (i.e., if s is $do(a, s')$ and a was executable in s') within the log s_n,. For the case of the sequence, $(\delta_1; \delta_2)^-$, the converse is distributed into the sequence and the order of the subprograms is reversed. In the cases of $(\delta_1 \mid \delta_2)^-$, $(\pi x.\delta)^-$ and $(\delta^*)^-$ the converse operator is simply pushed inwards. In the case of a test, the converse operator can be dropped. Finally, double converse cancels out.

We say that an ElGolog program δ (or test formula φ) is in *converse normal form* (NFC) if every occurrence of the converse construct it contains applies to a primitive action only.

We can define a transformation $nfc(\delta)$ which for any ElGolog program δ returns an equivalent program δ' that is in converse normal form.

$nfc(\delta)$ is defined inductively as follows:

$$
\begin{array}{lcl}
nfc(a) & \doteq & a \\
nfc(\varphi?) & \doteq & nfc(\varphi)? \\
nfc(P(\vec{x})) & \doteq & P(\vec{x}) \\
nfc(\neg\varphi) & \doteq & \neg nfc(\varphi) \\
nfc(\varphi_1 \wedge \varphi_2) & \doteq & nfc(\varphi_1) \wedge nfc(\varphi_2) \\
nfc(\exists x.\varphi) & \doteq & \exists x.nfc(\varphi) \\
nfc(\langle\delta\rangle\varphi) & \doteq & \langle nfc(\delta)\rangle nfc(\varphi) \\
nfc(\delta_1; \delta_2) & \doteq & nfc(\delta_1); nfc(\delta_2) \\
nfc(\delta_1|\delta_2) & \doteq & nfc(\delta_1)|nfc(\delta_2) \\
nfc(\pi x.\delta) & \doteq & \pi x.nfc(\delta) \\
nfc(\delta^*) & \doteq & nfc(\delta)^* \\
nfc(a^-) & \doteq & a^- \\
nfc((\delta_1; \delta_2)^-) & \doteq & nfc(\delta_2^-); nfc(\delta_1^-) \\
nfc((\delta_1|\delta_2)^-) & \doteq & nfc(\delta_1^-)|nfc(\delta_2^-) \\
nfc((\pi x.\delta)^-) & \doteq & \pi x.nfc(\delta^-) \\
nfc((\delta^*)^-) & \doteq & nfc(\delta^-)^* \\
nfc((\varphi?)^-) & \doteq & nfc(\varphi?) \\
nfc((\delta^-)^-) & \doteq & nfc(\delta)
\end{array}
$$

Essentially, $nfc(\delta)$ applies the definition of Do_{log} to non-atomic converse programs to push the converse construct down to the atomic action level.

It is easy to show that:

Lemma 1. $Do_{log}(nfc(\delta), s, s', s_n) = Do_{log}(\delta, s, s', s_n)$

Observe that for any ElGolog program δ in converse normal form, $Do_{log}(\delta, s, s', s_n)$ is defined purely in terms of Do_{log} for subprograms of δ. Also the cases of the definition of Do_{log} for non-atomic converse program are no longer necessary.

Examples

Let's look at some examples to understand the ElGolog semantics better.

Example 2. Consider the test program $\langle(a; b)^-\rangle P?$. Then we have that :

$$Do(\langle(a; b)^-\rangle P?, s, s') \doteq s' = s \wedge \langle(a; b)^-\rangle P[s, s]$$

The test formula then expands to:

$$
\begin{array}{l}
\langle(a; b)^-\rangle P[s, s] \\
\quad \equiv \exists s''.Do_{log}((a; b)^-, s, s'', s) \wedge P[s'', s] \\
\quad \equiv \exists s''.Do_{log}((a; b)^-, s, s'', s) \wedge P(s'')
\end{array}
$$

i.e., the test holds in $[s, s]$ if there is some situation s'' where P holds and there is an execution of $(a; b)^-$ from s to s'' within the log s.

Then if we expand Do_{log} we have that:

$$
\begin{array}{l}
Do_{log}((a; b)^-, s, s'', s) \\
\quad \equiv Do_{log}((b^-; a^-), s, s'', s) \\
\quad \equiv \exists s'''.Do_{log}(b^-, s, s''', s) \wedge Do_{log}(a^-, s''', s'', s) \\
\quad \equiv \exists s'''.Do_{log}(a, s'', s''', s) \wedge Do_{log}(b, s''', s, s) \\
\quad \equiv \exists s'''.s''' = do(a, s'') \wedge Poss(a, s'') \wedge s''' \sqsubseteq s \wedge \\
\qquad\qquad s = do(b, s''') \wedge Poss(b, s'') \wedge s \sqsubseteq s \\
\equiv s = do(b, do(a, s'')) \wedge Poss(a, s'') \wedge Poss(b, do(a, s''))
\end{array}
$$

Thus $Do(\langle(a; b)^-\rangle P?, s, s')$ holds if and only if $s' = s$ and there is some situation s'' in the past of s such that $s = do(b, do(a, s''))$, $P(s'')$ holds, and a followed by b is executable in s''.

A concrete instance of the above program is the following:

$$Do(\langle(goto(O_2); deliverCoffee(O_2))^-\rangle At(O_1)?, s, s')$$

i.e., the robot was at office O_1 before it went to office O_2 and then delivered coffee there to end up in situation s.

It is easy to show that the programs involving nested tests below are equivalent to the one in the previous example:

Example 3.

$$
\begin{array}{l}
Do(\langle(a; b)^-\rangle P?, s, s') \\
\quad \equiv Do(\langle b^-\rangle(\langle a^-\rangle P)?, s, s') \\
\quad \equiv Do(\langle(b^-; (\langle a^-\rangle P?))\rangle True?, s, s')
\end{array}
$$

The example in the introduction section shows how one can use rich tests in ElGolog to specify complex history dependent tasks. Another useful application of rich tests is to ensure that the "pick" construct choses a different entity at each iteration of a loop:

Example 4. The following program makes a robot deliver coffee to some office zero or more times such that no office is visited twice (and infinite loops are avoided):

$$
\begin{array}{l}
(\pi o.\neg\langle(deliverCoffee(o); any^*)^-\rangle True?; \\
\qquad goto(o); deliverCoffee(o))^*
\end{array}
$$

We can also ensure that every office gets a coffee delivery by adding a test at the end:

$$
\begin{array}{l}
(\pi o.\neg\langle(deliverCoffee(o); any^*)^-\rangle True?; \\
\qquad goto(o); deliverCoffee(o))^*; \\
\forall o.Office(o) \supset \langle(deliverCoffee(o); any^*)^-\rangle True?
\end{array}
$$

Properties

First, we show that an execution of a converse program δ^- is effectively a backwards execution δ:

Theorem 5. *For any ElGolog program δ, we have that $\mathcal{D} \models Do_{log}(\delta^-, s, s', s_n) \equiv Do_{log}(\delta, s', s, s_n)$.*

Proof. By induction on the structure of δ. Base cases: In case δ is an atomic action a, the thesis follows by the definition of $DoLog$. In case δ is a test $\phi?$, we have that

$$\begin{aligned}
&\mathcal{D} \models Do_{log}(\delta^-, s, s', s_n) \\
&\quad \equiv Do_{log}((\varphi?)^-, s, s', s_n) \\
&\quad \equiv Do_{log}((\varphi?), s, s', s_n) \\
&\quad \equiv s' = s \wedge \varphi[s, s] \wedge s \sqsubseteq s_n \\
&\quad \equiv Do_{log}((\varphi?), s', s, s_n) \\
&\quad \equiv Do_{log}(\delta, s', s, s_n)
\end{aligned}$$

Otherwise, assume as induction hypothesis that the thesis holds for all sub-program terms in δ. In case $\delta = (\delta_1; \delta_2)$, we have that

$$\begin{aligned}
&\mathcal{D} \models Do_{log}(\delta^-, s, s', s_n) \\
&\quad \equiv Do_{log}((\delta_1; \delta_2)^-, s, s', s_n) \\
&\quad \equiv Do_{log}((\delta_2)^-; (\delta_1)^-, s, s', s_n) \\
&\quad \equiv \exists s''. Do_{log}(\delta_2^-, s, s'', s_n) \wedge Do_{log}(\delta_1^-, s'', s', s_n) \\
&\quad \equiv \exists s''. Do_{log}(\delta_2, s'', s, s_n) \wedge Do_{log}(\delta_1, s', s'', s_n) \\
&\qquad \text{by the induction hypothesis} \\
&\quad \equiv Do_{log}((\delta_1; \delta_2), s', s, s_n) \\
&\quad \equiv Do_{log}(\delta, s', s, s_n)
\end{aligned}$$

All the other cases, i.e., where δ is $(\delta_1|\delta_2)$, $(\pi x.\delta_1)$, and (δ_1^*), can be proven in a similar way. □

We can also show the following result about Do_{log} :

Lemma 6. *For any ElGolog program δ:*

$$\mathcal{D} \models Do_{log}(\delta, s, s', s_n) \supset s \sqsubseteq s_n \wedge s' \sqsubseteq s_n$$

This can be shown by induction on situations and on the structure of δ.

Now, we observe that for any ElGolog program δ, $Do(\delta, s, s')$ stands for a situation calculus formula $\psi_\delta(s, s')$ that does not mention any program. In particular, note that $Do(\varphi?, s, s')$ stands for $\psi_{\varphi?}(s, s') = s' = s \wedge \varphi[s, s]$ by definition of Do. Notice as well that $\varphi[s, s]$ actually stands for a situation calculus formula with only one free situation variable s. Such a formula is not necessarily first-order as it may contain temporal operators involving programs of the form δ_i^*, that expand to second-order subformulas. However, we show next that if the situation parameter s is ground, then the formula that $\varphi[s, s]$ stands for is equivalent to a *first-order* formula which is moreover *regressable*.

First, let's define some bounded iteration abbreviations δ^k, meaning that δ is performed exactly k times, and $\delta^{\leq k}$, meaning that δ is performed 0 or more times, but at most k times:

$$\delta^k \doteq \begin{cases} True? & \text{if } k = 0 \\ (\delta^{k-1}; \delta) & \text{otherwise} \end{cases}$$

$$\delta^{\leq k} \doteq \begin{cases} \delta^0 & \text{if } k = 0 \\ (\delta^{\leq (k-1)} | \delta^k) & \text{otherwise} \end{cases}$$

Also, for any ground situation term S, we define $|S|$ to stand for the *length*, i.e., number of actions, in S.

Then we prove the following lemma about nondeterministic iteration programs, which says that, if there is an execution of δ^* from ground situation term S to ground situation term S' within the log S_n, then there is an execution of δ^* from S to S' within the log S_n that performs at most $|S_n|$ iterations of δ:[3]

Lemma 7. *For any ElGolog program δ and any ground situation terms, S, S', and S_n, we have that:*

$$\mathcal{D} \models DoLog(\delta^*, S, S', S_n) \supset DoLog(\delta^{\leq |S_n|}, S, S', S_n)$$

Proof. Take an arbitrary model M of $\mathcal{D}$ and assume that $M \models DoLog(\delta^*, S, S', S_n)$. Then there exists situations $s_1, \ldots, s_k$ with $s_1 = S$ and $s_k = S'$ such that $M \models DoLog(\delta, s_i, s_{i+1}, S_n)$ for $1 \leq i < k$. By Lemma 6, this implies that $M \models s_i \sqsubseteq S_n$ for $1 \leq i \leq k$. Thus, there are ground situation terms $S_1, \ldots, S_k$ such that $S_1 = S$ and $S_k = S'$, $M \models S_i \sqsubseteq S_n$ for $1 \leq i \leq k$ and $M \models DoLog(\delta, S_i, S_{i+1}, S_n)$ for $1 \leq i < k$. If $k \leq |S_n|$, then the thesis follows. Otherwise, since there are only $|S_n| + 1$ distinct ground situation terms S_g such that $S_g \sqsubseteq S_n$, there exists j and m such that $j < m$ and $S_j = S_m$, that is, there must be a "cycle" in the execution of δ^*. We can remove this cycle and still have an execution of δ^*, that is, $M \models DoLog(\delta, S_1, S_2, S_n) \wedge \ldots \wedge DoLog(\delta, S_{j-1}, S_j, S_n) \wedge DoLog(\delta, S_j, S_{m+1}, S_n) \wedge \ldots \wedge DoLog(\delta, S_{k-1}, S_k, S_n)$. We can repeat this cycle removal process until we have an execution of δ^* from S to S' such that performs δ at most $|S_n|$ times. □

We can use this lemma to show that an ElGolog rich test on ground situations expands to what is essentially a regressable (first-order) formula, and similarly for Do_{log} of an ElGolog program on a ground situation log:

Theorem 8. *For any ElGolog test φ, ElGolog program δ, and ground situation terms S_t, S, S', and S_n, such that $S_t \sqsubseteq S_n$, there exist first-order regressable situation calculus formulas ψ and ψ' such that $\mathcal{D} \models \varphi[S_t, S_n] \equiv \psi$ and $\mathcal{D} \models Do_{log}(\delta, S, S', S_n) \equiv \psi'$*

Proof. We assume wlog. that φ and δ are in converse normal form. We show the thesis by induction on the structure of φ and δ.
Base cases:
For $\varphi = P(\vec{x})$, the thesis trivially holds.
For $\delta = a$, we have that

$$\begin{aligned}
&Do_{log}(a, S, S', S_n) \doteq \\
&\quad S' = do(a, S) \wedge Poss(a, S) \wedge S' \sqsubseteq S_n
\end{aligned}$$

The thesis then follows immediately by the definition of regressable formula (since S' and S_n are ground, $S' \sqsubseteq S_n$ is

[3] Note that unlike Golog programs, whose execution can only go forward, ElGolog programs may go back and forth within the log, e.g., $(a^-|a)^*$ has executions from S_0 to $do(a, S_0)$ within the log $do(a, S_0)$ that iterate hundreds of times (assuming that a is executable in S_0). So the fact that the lemma holds is significant.

either equivalent to $True$ or to $False$).
For $\delta = a^-$, we have that

$$Do_{log}(a^-, S, S', S_n) \doteq Do_{log}(a, S', S, S_n)$$
$$\equiv S = do(a, S') \wedge Poss(a, S') \wedge S \sqsubseteq S_n$$

The thesis then follows immediately by the definition of regressable formula.

Inductive step:
Assume that the thesis holds for all subtests and subprograms in φ and δ (induction hypothesis, IH for short).
For the cases $\varphi = \neg\varphi'$, $\varphi = \varphi' \wedge \varphi''$, and $\varphi = \exists x.\varphi'$, the thesis follows immediately by the IH and the definition of regressable formula.
For $\varphi = \langle\delta\rangle\varphi'$, we have that

$$(\langle\delta\rangle\varphi')[S_t, S_n] \doteq \exists s.Do_{log}(\delta, S_t, s, S_n) \wedge \varphi'[s, S_n]$$
$$\equiv \bigvee_{S'':S''\sqsubseteq S_n} Do_{log}(\delta, S, S'', S_n) \wedge \varphi'[S'', S_n]$$

since by Lemma 6 s must be a prefix of ground situation S_n.

The thesis then follows immediately by the IH and the definition of regressable formula.
For the cases $\delta = \delta_1|\delta_2$ and $\delta = \pi x.\delta'$, the thesis follows immediately by the IH and the definition of regressable formula.
For $\delta = \delta_1;\delta_2$, we have that

$$Do_{log}((\delta_1;\delta_2), S, S', S_n) \doteq$$
$$\exists s.Do_{log}(\delta_1, S, s, S_n) \wedge Do(\delta_2, s, S', S_n)$$
$$\equiv \bigvee_{S''.S''\sqsubseteq S_n} Do_{log}(\delta_1, S, S'', S_n) \wedge Do_{log}(\delta_2, S'', S', S_n)$$

since by Lemma 6 s must be a prefix of ground situation S_n.

The thesis then follows immediately by the IH and the definition of regressable formula.
For $\delta = \delta_1^*$, by Lemma 7, we have that

$$Do_{log}(\delta_1^*, S, S', S_n) \equiv Do_{log}(\delta_1^{\leq|S_n|}, S, S', S_n).$$

Then we can use the same arguments as in the nondeterministic branch and sequence cases to show that the thesis holds. □

We can also draw some results about Golog from this. First, we have that:

Lemma 9. *For any Golog program δ,*

$$\mathcal{D} \models Do(\delta, s, s') \supset s \sqsubseteq s'$$

This can be proven by induction on situations and on the structure of Golog programs.

We can also show that for Golog programs, Do_{log} is equivalent to Do:

Theorem 10. *For any Golog program δ,*

$$\mathcal{D} \models Do_{log}(\delta, s, s', s') \equiv Do(\delta, s, s')$$

The proof is by induction on the structure of δ, using the fact that for any Golog program δ, $\mathcal{D} \models Do(\delta, s, s') \supset s \sqsubseteq s'$.

From this, it follows that the result in Theorem 8 also applies to Golog programs:

Corollary 11. *For any Golog program δ, and ground situation terms S and S', there exist a first-order regressable situation calculus formula ψ such that $\mathcal{D} \models Do(\delta, S, S') \equiv \psi$.*

In summary, when the situation parameter is ground, ElGolog rich tests can be handled with the same techniques as standard tests without temporal operators. More generally, observe that for both Golog and ElGolog programs δ, if the situation parameters s and s' are ground, $Do(\delta, s, s')$ is equivalent to a first-order situation calculus formula. Also, for both Golog and ElGolog programs δ, if δ contains no iteration, $Do(\delta, s, s')$ is also equivalent to a first-order situation calculus formula. Of course, in the general case, if δ contains iteration and the situation parameters s and s' are not ground, then $Do(\delta, s, s')$ expands to a second-order formula in both the Golog and the ElGolog case. Hence, in spite of its ability to query the execution history, ElGolog can be handled with the same technical machinery as standard Golog.

Prolog Implementation

We have developed an implementation of ElGolog[4] in SWI-Prolog, which is based on the implementation of Golog in (Levesque et al. 1997; Reiter 2001). Our implementation assumes that the basic action theory is represented in a particular way in Prolog and that the initial situation description satisfies the Prolog closed world assumption. We use the same implementation syntax for programs (and action theories) as (Reiter 2001), to which we add `diamond(Delta,Phi)` for $\langle\delta\rangle\varphi$ and `conv(Delta)` for δ^-, where `Delta` and `Phi` are δ and φ expressed in the implementation syntax.

The main predicate of the ElGolog interpreter is `do(Delta,S1,S2)`, which implements $Do(\delta, s_1, s_2)$. The definition contains clauses such as:

```
do(E,S,do(E,S)) :-
   primitive_action(E), poss(E,S).
do(?(P),S,S) :- holds(P,S,S).
do(E1 : E2,S,S1) :-  % handles sequence
   do(E1,S,S2), do(E2,S2,S1).
do(E1 # E2,S,S1) :-  % handles nondet branch
   do(E1,S,S1) ; do(E2,S,S1).
do(star(E),S,S1) :-
   S1 = S ; do(E : star(E),S,S1).
```

The predicate `holds(Phi,St,Sn)` implements the evaluation of a test formula $\varphi[s_t, s_n]$. The definition contains clauses such as:

```
holds(P & Q,St,Sn) :-
   holds(P,St,Sn), holds(Q,St,Sn).
```

As in the original Golog implementation, atomic fluent formulas are handled by restoring the situation argument and invoking the result in Prolog, which first regresses it and then evaluates it in the initial situation. Existentially quantified variables are handled by substituting in a fresh Prolog variable, and using Prolog evaluation. Negation is handled by applying Lloyd-Topor transformations and using Prolog negation-as-failure. See (Reiter 2001) for more details.

[4]The ElGolog interpreter together with some examples is available at https://www.eecs.yorku.ca/~lesperan/code/ElGolog.

The following clause implements the evaluation of the temporal operator:

```
holds(diamond(E,P),St,Sn) :-
   doLog(E,St,S1,Sn), holds(P,S1,Sn).
```

It uses the predicate `doLog(Delta,S1,S2,Sn)`, which implements $Do_{log}(\delta, s_1, s_2, s_n)$. This predicate first pushes the converse construct inwards, all the way down to atomic actions. Then there are various clauses that handle other forms of programs, such as:

```
doLog(E,S,do(E,S),Sn) :- primitive_action(E),
   leq(do(E,S),Sn), poss(E,S).
doLog(conv(E),S,S1,Sn) :-
   primitive_action(E), doLog(E,S1,S,Sn).
doLog(?(P),S,S,Sn) :-
   leq(S,Sn), holds(P,S,S).
doLog(E1 : E2,S,S1,Sn) :-
   doLog(E1,S,S2,Sn), doLog(E2,S2,S1,Sn).
```

These are mostly as in `do(Delta,S1,S2)`, except that we check that the execution remains in the log `Sn`. This uses the auxiliary predicate `leq(S1,S2)`, which implements $s_1 \sqsubseteq s_2$:

```
leq(S,S).
leq(S,do(A,S1)) :- leq(S,S1).
```

Example 12. The basic action theory for a simple version of the coffee delivery domain can be implemented as follows:

```
room(giuseppeOf). room(yvesOf).
room(eugeniaOf).

primitive_action(goto(R)) :- room(R).
primitive_action(deliverCoffee(R)):- room(R).
primitive_action(wait).

poss(goto(R),S) :- room(R), \+ at(R,S).
poss(deliverCoffee(R),S) :- at(R,S).
poss(wait,S).

at(R,do(A,S)) :- A = goto(R) ;
   at(R,S), \+ A = goto(R1).

at(coffeeRoom,s0).
```

We can define a nondeterministic coffee delivery procedure that never delivers to the same office twice as follows:

```
proc(cdp,
   star(pi(r, (
     ?(room(r) &
       -diamond(conv((deliverCoffee(r) :
                      star(any)),
                true))
     : goto(r) : deliverCoffee(r))))).
```

When we execute this procedure, we obtain various executions that deliver coffee to zero or more offices, never going to the same office twice:

```
?- do(cdp,s0,S).
S = s0 ;
S = do(deliverCoffee(giuseppeOf),
do(goto(giuseppeOf), s0)) ;
S = do(deliverCoffee(yvesOf),
do(goto(yvesOf),
do(deliverCoffee(giuseppeOf),
do(goto(giuseppeOf), s0)))) ;
S = do(deliverCoffee(eugeniaOf),
do(goto(eugeniaOf),
do(deliverCoffee(yvesOf),
do(goto(yvesOf),
do(deliverCoffee(giuseppeOf),
do(goto(giuseppeOf), s0))))))) ;
S = do(deliverCoffee(eugeniaOf),
do(goto(eugeniaOf),
do(deliverCoffee(giuseppeOf),
do(goto(giuseppeOf), s0))))
...
```

Discussion

The need to handle domain dynamics which depends on past histories has long been recognized as an important issue in reasoning about action (Giunchiglia and Lifschitz 1995; Mendez, Lobo, and Baral 1996; Gelfond and Lifschitz 1998). In particular, Gabaldon (Gabaldon 2011) extends Reiter's basic action theories (Reiter 2001), which are Markovian, to non-Markovian basic action theories, thus allowing preconditions and effects of actions to be determined by the whole history, not just the current situation. He also extends regression to handle formulas that quantify over situations that are bounded by a ground situation term, thus supporting regression over non-Markovian basic action theories. (Gabaldon 2011) does in fact advocate for future work on a version of Golog would support test conditions that refer to the past, as we have provided in ElGolog.

There is also related work in the "action languages" approach (Gelfond and Lifschitz 1998). For instance, Gonzalez et al. (Gonzalez, Baral, and Gelfond 2005) introduced Alan, an $\mathcal{A}$-like language for modeling non-Markovian domains. One of their motivations is the practical problem of modeling multimedia presentations that involve temporal conditions constraining how a presentation evolves. (Giunchiglia and Lifschitz 1995) also handled non-Markovian domain specifications. More recently, non-Markovian planning domains expressed in Linear Dynamic Logic have been studied in (Brafman and De Giacomo 2019).

Our contribution in this paper is orthogonal to work on handling non-Markovian action theories, since our focus is on allowing the agent's program to make non-Markovian tests consisting of historical queries on its log. For simplicity, we have assumed world dynamics to be specified by classical basic action theories (Reiter 2001), but our work could be easily extended to deal with non-Markovian ones.

To sumarize, in this paper, we have presented ElGolog, a high-level programming language extending Golog that supports rich tests about the execution history, where tests are expressed in a first-order variant of two-way linear dynamic logic that uses ElGolog programs with converse. This allows procedures that involve complex history-dependent conditions to be expressed naturally, without having to introduce new task-specific fluents or data structures into the action theory/domain model. We have shown that in spite of its rich tests, ElGolog can be provided with a semantics based on macroexpansion into situation calculus formulas (Levesque et al. 1997; Reiter 2001), upon which regression can be ap-

plied, just like Golog. We also described an implementation of ElGolog in Prolog.

The ability to refer to the past, as advocated by McCarthy (McCarthy 1992; 2007). is important to keep the model of the world (the fluents and atomic actions) and the model of the task (the Golog/ElGolog program) separate. Indeed on the same world model, we may run several programs performing different tasks. If we cannot refer to the history, then we need to summarize the relevant part of the history in newly introduced fluents in an extended world model. But these fluents depend on the task, so for every new task we would need to add new task-specific fluents. This clutters the world model with extra information that does not pertain to the world itself, but to the (essentially unboundedly many) tasks that we may perform on it.

Note that in the related work on non-Markovian action theories by Gabaldon, Gonzalez et al., etc., avoiding the introduction of auxiliary fluents is one of the main motivations. This argument applies to any kind of system where we distinguish between a world model (including dynamics) that remains valid over time and a variety of control tasks that are performed on it at different times. Even in C or Java it may be advantageous to refer to the log when programming such tasks. However, in these languages, we do not have a convenient data structure for the log. Instead in the situation calculus, we do have one, the current situation, which is in fact the history from the starting state to the current one. In Golog, this log is used only when doing regression, but in ELGolog instead we use it as well for querying the past, and as we have shown, we can still apply regression as we do in the standard Golog. Obviously, from time to time we will "progress" the action theory to the current situation (Reiter 2001), and this corresponds to erasing the log and having a fresh start in the state corresponding to the current situation. This means that we lose access to the history. The question of how to handle progression in ElGolog is an interesting topic for future research.

Our approach should be applicable to other agent programming languages. For instance, one could try to extend Thielscher's FLUX language (Thielscher 2005) along these lines. Our ideas could also be applied to BDI agent programming languages (e.g., Jason (Bordini, Hübner, and Wooldridge 2007), 2APL (Dastani 2008), etc.), although this would be more challenging as these typically only remember the current belief state. Note that it would also interesting to investigate techniques to remember only the part of the past history that is required to evaluate the historical tests that appear in a given program.

In future work, we would like to extend our results to a version of ElGolog based on non-Markovian action theories. It would also be interesting to generalize the approach to handle concurrent programs as in ConGolog. Finally, it would be worthwhile to investigate the usability of ElGolog in practical domains such as cognitive robotics.

Acknowledgments

This work is supported in part by the European Research Council under the European Unions Horizon 2020 programme through the ERC Advanced Grant WhiteMec (No. 834228), and the National Science and Engineering Research Council of Canada.

References

Bordini, R. H.; Hübner, J. F.; and Wooldridge, M. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley.

Brafman, R. I., and De Giacomo, G. 2019. Planning for LTLf /LDLf goals in non-Markovian fully observable nondeterministic domains. In *IJCAI*, 1602–1608.

Dastani, M. 2008. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems* 16(3):214–248.

De Giacomo, G., and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*.

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

Gabaldon, A. 2011. Non-Markovian control in the situation calculus. *Artificial Intelligence* 175(1):25–48.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electron. Trans. Artif. Intell.* 2:193–210.

Giunchiglia, E., and Lifschitz, V. 1995. Dependent fluents. In *IJCAI*, 1964–1969. Morgan Kaufmann.

Gonzalez, G.; Baral, C.; and Gelfond, M. 2005. Alan: An action language for modelling non-Markovian domains. *Studia Logica* 79(1):115–134.

Harel, D.; Kozen, D.; and Tiuryn, J. 2000. *Dynamic Logic*. MIT Press.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programing* 31(1-3):59–83.

McCarthy, J., and Hayes, P. J. 1969. Some Philosophical Problems From the Standpoint of Artificial Intelligence. *Machine Intelligence* 4:463–502.

McCarthy, J. 1992. Elephant 2000: A programming language based on speech acts. Available at http://www-formal.stanford.edu/jmc/elephant.pdf.

McCarthy, J. 2007. Elephant 2000: a programming language based on speech acts. In *OOPSLA*, 723–724.

Mendez, G.; Lobo, J.; and Baral, J. L. C. 1996. Temporal logic and reasoning about actions. In *Commonsense*.

Pirri, F., and Reiter, R. 1999. Some contributions to the metatheory of the situation calculus. *J. ACM* 46(3):325–361.

Reiter, R. 2001. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.

Sardiña, S.; De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2004. On the semantics of deliberation in Indigolog - from theory to implementation. *Ann. Math. Artif. Intell.* 41(2-4):259–299.

Thielscher, M. 2005. FLUX: A logic programming method for reasoning agents. *TPLP* 5(4-5):533–565.