Strategic Reasoning over Golog Programs in the Nondeterministic Situation Calculus

Giuseppe De Giacomo[1,2], **Yves Lespérance**[3], **Matteo Mancanelli**[2]

[1]University of Oxford, Oxford, UK
[2]University of Rome La Sapienza, Rome, Italy
[3]York University, Toronto, ON, Canada
giuseppe.degiacomo@cs.ox.ac.uk, lesperan@eecs.yorku.ca, mancanelli@diag.uniroma1.it

Abstract

We investigate the problem of synthesizing strategies that guarantee the successful execution of a high-level nondeterministic agent program δ in Golog within a nondeterministic first-order basic action theory, considering the environment as adversarial. Our approach constructs a symbolic program graph that captures the control flow of δ independently of the domain, enabling strategy synthesis through the cross product of the program graph with the domain model. We formally relate graph-based transitions to standard Golog semantics and provide a synthesis procedure that is sound though incomplete (in general, the problem is undecidable, given that we have a first-order representation of the state). We also extend the framework to handle the case where the environment's possible behaviors are specified by a Golog program.

Introduction

(Levesque et al. 1997) introduced *high-level program execution* as a way of designing autonomous agents, which seeks a middle ground between planning and normal programming. Instead of looking for a plan/strategy to achieve a goal as in planning, the agent looks for a strategy to successfully execute a *nondeterministic program* that specifies its task. They proposed the Golog language for specifying such programs, which use constructs such as branches, loops, and nondeterministic choice, and are defined over atomic actions and fluent conditions that are specified by a Basic Action Theory (BAT) in the situation calculus (McCarthy and Hayes 1969; Reiter 2001). To find a strategy to execute the program, the agent needs to reason over the action theory. If the program is very deterministic, searching for a successful execution strategy is easy, but as more nondeterminism is introduced, the search becomes harder and begins to resemble planning.

The problem we address in this paper is a variant of the *high-level program execution task*, i.e., given a Golog program δ and an action theory $\mathcal{D}$, find a strategy for executing δ that guarantees successful termination starting from the initial situation S_0 in the domain specified by $\mathcal{D}$. If the domain is deterministic and the program is situation determined, i.e., there is a unique remaining program after executing each step, such a strategy can simply specify a sequence of actions $\vec{a}$ to execute such that $\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$, i.e., doing $\vec{a}$ in the initial situation yields a complete execution of δ. If the domain is deterministic but the program is not situation determined, the strategy should also specify the remaining program after each action in the sequence.

In this paper, we instead consider the case where the *domain is nondeterministic* and specified by a nondeterministic basic action theory (De Giacomo and Lespérance 2021) (DL21). For a situation determined program δ, we want to synthesize a strategy f, i.e., a mapping from situations to actions, such that the agent can force the program to be executed to successful termination by following f, i.e., $\mathcal{D} \models AgtCanForce(\delta, S_0, f)$. If the program is not situation determined, we also need to generate a strategy to select the remaining program, i.e., a mapping g from situations to programs, such that $\mathcal{D} \models AgtCanForce(\delta, S_0, f, g)$. If, as is typical in FOND planning (Geffner and Bonet 2013; Ghallab, Nau, and Traverso 2016), we have complete information about the initial state, i.e., have a model M such that $M \models \mathcal{D}$, then we need to generate a strategy f such that $M \models AgtCanForce(\delta, S_0, f)$ in the situation determined program case; in the non-situation determined program case, we also need to generate a strategy g to chose the remaining program, such that $M \models AgtCanForce(\delta, S_0, f, g)$.

Most prior work on Golog program execution (Baier, Fritz, and McIlraith 2007; Baier et al. 2008; Fritz, Baier, and McIlraith 2008) focuses on deterministic environments, compiling Golog into the domain so classical planners can be used. We instead target nondeterministic domains using a reactive synthesis approach. A recent advancement is (Hofmann and Claßen 2025) (HC25), which uses the C^2 decidable fragment of FOL but suffers from scalability issues. Our framework is simpler and more intuitive, while offering formal guarantees that relate program executions with classical Golog semantics and synthesis techniques. Our framework is also very flexible, and it can easily be extended to accommodate different challenges. We will show how one can specify separate programs on the agent's behavior and on the environment's behavior; this contrasts with (HC25), which takes the Golog program as a constraint on the behavior of the whole system, i.e., both the agent and environment.

Preliminaries

Nondeterministic situation calculus. The *situation calculus* is a predicate logic language designed for specifying dynamically changing worlds. World changes result from performing *actions*, and world histories are modeled as *situations*, which are action sequences. The constant S_0 denotes the initial situation, and the function $do(a, s)$ denotes the successor situation after executing a in s. Fluents are predicates or functions varying with the situation, and take a situation term as their final argument. In this language, we define domains represented by a *basic action theory (BAT)*, where successor state axioms (SSAs) represent the causal laws of the domain (Reiter 2001). A predicate $Poss(a, s)$ is used to state that a is executable in s. We can rely on the mechanism of regression $\mathcal{R}$ (Pirri and Reiter 1999) to reduce reasoning about a given future situation to reasoning about S_0.

(DL21) propose a simple extension of the situation calculus to handle nondeterministic actions. For any primitive action in a nondeterministic domain, there can be a number of different outcomes, depending on how the environment behaves. This is modeled by an additional environment reaction parameter e, ranging over a new sort *Reaction*. We call the reaction-suppressed version of the action $a(\vec{o})$ an *agent action* and the full version $a(\vec{o}, e)$ a *system action*. A *nondeterministic basic action theory (NDBAT)* $\mathcal{D}$ is a special kind of BAT, and it is the union of the following disjoint sets: foundational, domain independent, axioms of the situation calculus (Σ), axioms describing the initial situation ($\mathcal{D}_{S_0}$), unique name axioms for actions and domain closure for action types ($\mathcal{D}_{ca}$), successor state axioms describing how *system* actions change the fluents ($\mathcal{D}_{ssa}$), and *system* action precondition axioms, stating when the complete system action can occur ($\mathcal{D}_{poss}$). One also specifies agent action preconditions using $Poss_{ag}$. The theory must entail the *reaction independence* requirement (i.e., $\forall e.Poss(a(\vec{o}, e), s) \supset Poss_{ag}(a(\vec{o}), s)$) and the *reaction existence* requirement (i.e., $Poss_{ag}(a(\vec{o}), s) \supset \exists e.Poss(a(\vec{o}, e), s)$).

We assume finitely many primitive action types $\mathcal{A}$ and fluent predicates $\mathcal{F}$, no functions beyond constants, and all object terms drawn from a countably infinite set $\mathcal{N}$ of *standard names* (Levesque and Lakemeyer 2001), with the unique name assumption and domain closure. We also assume that $Reaction$ is a sub-sort of $Object$. This means that in all models of the theory, object domains are isomorphic and countably infinite, where each element of the domain is denoted by a syntactic term, and we can use ground terms to denote any value of a domain. More generally, this allows us to confuse wlog substitutions of ground terms and assignments, and ends interpret quantification substitutionally.

Golog syntax. Golog is a high-level language for writing programs that are executed over a (possibly nondeterministic) Basic Action Theory. We consider programs written in a variant of Golog where the test construct yields no transitions and is final when the condition is satisfied (Claßen and Lakemeyer 2008; De Giacomo, Lespérance, and Pearce 2010). The key feature of our variant is that programs instruct agent actions, rather than system actions. The syntax of Golog programs is as follows:

$$\delta ::= a(\vec{o}) \mid \varphi? \mid \delta_1; \delta_2 \mid \delta_1 | \delta_2 \mid \pi x.\delta \mid \delta^*$$

where $a(\vec{o})$ is an agent action term, $\varphi?$ tests that φ holds, $\delta_1; \delta_2$ is the sequential execution of δ_1 and δ_2, $\delta_1|\delta_2$ is the nondeterministic choice of δ_1 and δ_2, $\pi x.\delta$ executes program δ for *some* nondeterministic choice of the object variable x, and δ^* performs δ zero or more times. Note that φ is a situation-suppressed formula, and we denote by $\varphi[s]$ the formula obtained by restoring the situation argument s into all fluents in φ. We use nil as an abbreviation for $\texttt{True}?$ to denote the empty program.

Golog Semantics and Variable Environments. To construct the program graph of a given program δ_0, we need to talk about the subprograms of δ_0. This is captured by the concept of syntactic closure. Following (De Giacomo et al. 2016), we separate the program terms themselves from the assignments to pick variables (i.e., variables bound by π). Let us assume wlog that all pick variables are renamed apart and there is a predefined ordering on such variables. It is possible to define a n-tuple of object terms $\vec{x} = \langle x_1, ..., x_n \rangle$, called the *environment term*, where x_i is the current value of i-th pick variable of δ_0. We denote the cartesian product of n objects as $\mathcal{O}^n$, and the object assigned to variable z as x_z. We use a triple $(\delta, \vec{x}, s)$ to denote a complete configuration. At the beginning of an execution, the environment term is arbitrarily instantiated, and at each step of the computation it maintains only n values, where n is the number of pick variables. Tracking pick variables' values separately avoids enumerating all the possible instantiations of $\pi x.\delta$, keeping the number of resulting subprograms finite. We can define inductively the syntactic closure Γ_{δ_0} of a program δ_0: *(i)* $\delta_0, nil \in \Gamma_{\delta_0}$, *(ii)* if $\delta_1; \delta_2 \in \Gamma_{\delta_0}$ and $\delta_1' \in \Gamma_{\delta_1}$, then $\delta_1'; \delta_2 \in \Gamma_{\delta_0}$ and $\Gamma_{\delta_2} \subseteq \Gamma_{\delta_0}$, *(iii)* if $\delta_1 \mid \delta_2 \in \Gamma_{\delta_0}$, then $\Gamma_{\delta_1}, \Gamma_{\delta_2} \subseteq \Gamma_{\delta_0}$, *(iv)* if $\pi z.\delta \in \Gamma_{\delta_0}$, then $\Gamma_\delta \subseteq \Gamma_{\delta_0}$, *(v)* if $\delta^* \in \Gamma_{\delta_0}$ then $\delta; \delta^* \in \Gamma_{\delta_0}$

Theorem 1. *The syntactic closure Γ_{δ_0} of a Golog program δ_0 is linear in the size of δ_0.*

The semantics of Golog is specified as usual in terms of single-steps, using two predicates: $Final$, specifying that the program may terminate in a given situation, and $Trans$, specifying that one step of the program δ in situation s may lead to situation s' with δ' remaining to be executed. Differently from previous works, we define *Trans* and *Final* considering both the triple-based configuration version and the environment reactions as follows:

$$\begin{aligned}
&\mathit{Trans}(a, \vec{x}, s, \delta', \vec{x}', s') \equiv \exists e.\mathit{Poss}(a[\vec{x}](e), s) \wedge \\
&\quad \delta' = nil \wedge \vec{x}' = \vec{x} \wedge s' = do(a[\vec{x}](e), s) \\
&\mathit{Trans}(\varphi?, \vec{x}, s, \delta', \vec{x}', s') \equiv \texttt{False} \\
&\mathit{Trans}(\delta_1; \delta_2, \vec{x}, s, \delta', \vec{x}', s') \equiv \mathit{Trans}(\delta_1, \vec{x}, s, \delta_1', \vec{x}', s') \wedge \\
&\quad \delta' = \delta_1'; \delta_2 \vee \mathit{Final}(\delta_1, \vec{x}, s) \wedge \mathit{Trans}(\delta_2, \vec{x}, s, \delta', \vec{x}', s') \\
&\mathit{Trans}(\delta_1 | \delta_2, \vec{x}, s, \delta', \vec{x}', s') \equiv \mathit{Trans}(\delta_1, \vec{x}, s, \delta', \vec{x}', s') \vee \\
&\quad \mathit{Trans}(\delta_2, \vec{x}, s, \delta', \vec{x}', s') \\
&\mathit{Trans}(\pi z.\delta, \vec{x}, s, \delta', \vec{x}', s') \equiv \exists d.\mathit{Trans}(\delta, \vec{x}^z_d, s, \delta', \vec{x}', s') \\
&\mathit{Trans}(\delta^*, \vec{x}, s, \delta', \vec{x}', s') \equiv \mathit{Trans}(\delta, \vec{x}, s, \delta'', \vec{x}', s') \wedge \\
&\quad \delta' = \delta''; \delta^*
\end{aligned}$$

$$\begin{aligned}
&\mathit{Final}(a, \vec{x}, s) \equiv \texttt{False} \\
&\mathit{Final}(\varphi?, \vec{x}, s) \equiv \varphi[\vec{x}][s] \\
&\mathit{Final}(\delta_1; \delta_2, \vec{x}, s) \equiv \mathit{Final}(\delta_1, \vec{x}, s) \wedge \mathit{Final}(\delta_2, \vec{x}, s) \\
&\mathit{Final}(\delta_1 | \delta_2, \vec{x}, s) \equiv \mathit{Final}(\delta_1, \vec{x}, s) \vee \mathit{Final}(\delta_2, \vec{x}, s) \\
&\mathit{Final}(\pi z.\delta, \vec{x}, s) \equiv \exists d.\mathit{Final}(\delta, \vec{x}^z_d, s) \\
&\mathit{Final}(\delta^*, \vec{x}, s) \equiv \texttt{True}
\end{aligned}$$

where $a[\vec{x}]$ and $\varphi[\vec{x}]$ denotes the action term a and formula φ obtained by replacing all free pick variables with the object terms to which they are bound in $\vec{x}$, and $\vec{x}^z_d$ denotes the environment term obtained by replacing the element corresponding to the pick variable z with d.

We use *Trans** to denote the reflexive transitive closure of *Trans*, i.e., *Trans*$^*(\delta, \vec{x}, s, \delta', \vec{x}', s')$ means that there exists a sequence of one-step transitions taking the configuration $(\delta, \vec{x}, s)$ into the configuration $(\delta', \vec{x}', s')$. We use also an adapted version of the notion of *situation determined* (SD) programs (De Giacomo, Lespérance, and Muise 2012):

$$\begin{array}{l} SituationDetermined(\delta, \vec{x}, s) \doteq \\ \quad \forall s', \delta', \delta'', \vec{x}', \vec{x}''.Trans^*(\delta, \vec{x}, s, \delta', \vec{x}', s') \wedge \\ \qquad Trans^*(\delta, \vec{x}, s, \delta'', \vec{x}'', s') \supset \delta' = \delta'' \wedge \vec{x}' = \vec{x}'' \end{array}$$

Now, we have guarantees that all programs that δ_0 can evolve into according to $Trans$ and $Final$, must be in its syntactic closure Γ_{δ_0}.

Lemma 2. *Let $\mathcal{D}$ be an NDBAT over which δ_0 is executed and M a model of $\mathcal{D}$. If $\delta \in \Gamma_{\delta_0}$ and $M \models Trans(\delta, \vec{x}, s, \delta', \vec{x}', s')$, then $\delta' \in \Gamma_{\delta_0}$.*

Proof. It follows directly from the recursive definition of $Trans$ and the construction of Γ_{δ_0}. □

Lemma 3. *Let $\mathcal{D}$ be an NDBAT over which δ_0 is executed and M a model of $\mathcal{D}$. If $M \models Trans^*(\delta_0, \vec{x}_0, s_0, \delta, \vec{x}, s)$, then $\delta \in \Gamma_{\delta_0}$.*

Proof (sketch). By induction on the number of steps in the derivation of $Trans^*(\delta_0, \vec{x}_0, s_0, \delta, \vec{x}, s)$. □

Example. A service robot must deliver a cup of coffee from the kitchen k to some offices *officeA*, *officeB*, and so on. The doors between the kitchen and the rooms are locked, and the robot must press a button to request that a door be opened. Which door opens is under the control of the environment. We assume the agent can execute the following actions: $Pickup$ (grab a coffee), $PutDown$ (place the currently held coffee), $PressButton(e)$, (request to open some door, where the environment chooses the door via the environment reaction $e = \mathit{openOffice\_r}$), and $MoveTo(r)$ (move to room r). Note that when it is not explicit, we assume an action has only one environment reaction $Success$. The fluents we need tell us where the agent is ($At(r)$), if the agent has grabbed a coffee ($\mathit{HasCoffee}$), if the coffee has been delivered ($Delivered(r)$), and if the door is open ($DoorOpen(r)$). The action preconditions and effects are as expected (e.g., the agent can move to a given office only if the corresponding door is open). For brevity, we use $DeliverTo(r) \doteq MoveTo(r); PutDown; MoveTo(k)$, and also $\varphi(r) \doteq DoorOpen(r) \wedge \neg Delivered(r)$. The agent program to deliver coffee is $\delta = (Pickup; (PressButton; \exists r.\varphi(r)?)^*; \pi r.\varphi(r)?; DeliverTo(r))^*; \forall r.Delivered(r)?$, that is, the agent picks up coffee, repeatedly presses the button, and each time delivers coffee to an office with an open door that has not yet been served (if any). We assume the environment cannot act "unfairly", i.e. eventually every door will be opened. A counter associated with the button prevents the environment from opening the same door infinitely often. The fluent $CanOpen(r)$ indicates if the door can still be opened, i.e. the counter has not reached its bound.

First-Order Program Graphs

We aim at performing transition system (TS)-based synthesis for generating a strategy compliant with a given Golog program. Synthesis can be viewed as a 2-player game among agent and environment (this is common also when dealing with temporal properties and reactive synthesis; see (Abadi, Lamport, and Wolper 1989; Pnueli and Rosner 1989)). Here we construct the game arena by considering both the FOND domain where the agent is acting and the program graph, whose paths represent executions of the specified Golog program. In this section, we show how the program graphs are constructed and what their properties are.

Constructing the Program Graph. Inspired by (Claßen 2013; Claßen and Lakemeyer 2008), we construct the program graph of a Golog program δ_0, characterizing all possible executions of δ_0 conditioned on the domain, which we do not fix until later. The program graph is thus a symbolic structure that captures the control flow semantics of Golog at the syntactic level, decoupled from domain dynamics, which are only integrated later via the cross product with the domain. Each node of the graph corresponds to a program in the syntactic closure of δ_0 (i.e., a possible remaining program in the execution of δ_0) and each edge is a guarded one-step transition from such a program to the remaining one.

We define properties of the nodes and relationships among them based on $Trans$ and $Final$. Since we keep the pick operator, the program graph must capture all execution paths of the Golog program parametrized over the pick variables. In particular, pick variables are unbounded parameters of actions and tests, and no evaluation or binding can be performed before performing the cross product with the domain. Thus, we have to keep track of the pick variables for which we must assign a value of the domain. Specifically, we define two new relations T and F in a recursive way:

$$\begin{array}{l} T(a,a) = \{(Poss(a), \emptyset, nil)\} \\ T(a,b) = \{\} \\ T(\varphi?, a) = \{\} \\ T(\delta_1;\delta_2, a) = \\ \quad \{(\neg F(\delta_1) \wedge \varphi, P, \delta_1'; \delta_2) \mid (\varphi, P, \delta_1') \in T(\delta_1, a)\} \cup \\ \quad \{(F(\delta_1) \wedge \varphi, P, \delta_2') \mid (\varphi, P, \delta_2') \in T(\delta_2, a)\} \\ T(\delta_1 | \delta_2, a) = T(\delta_1, a) \cup T(\delta_2, a) \\ T(\pi z.\delta, a) = \{(\varphi, P \cup z, \delta') \mid (\varphi, P, \delta') \in T(\delta, a)\} \\ T(\delta^*, a) = \\ \quad \{(\neg F(\delta) \wedge \varphi, P, \delta'; \delta^*) \mid (\varphi, P, \delta') \in T(\delta, a)\} \end{array}$$

$$\begin{array}{l} F(a) = False \\ F(\varphi?) = \varphi \\ F(\delta_1;\delta_2) = F(\delta_1) \wedge F(\delta_2) \\ F(\delta_1|\delta_2) = F(\delta_1) \vee F(\delta_2) \\ F(\pi z.\delta) = \exists z.F(\delta) \\ F(\delta^*) = True \end{array}$$

Given a Golog program δ and and action a, $T(\delta, a)$ returns a set of triples (φ, P, δ'), where φ is a guard condition representing the cases in which the transition can be performed,

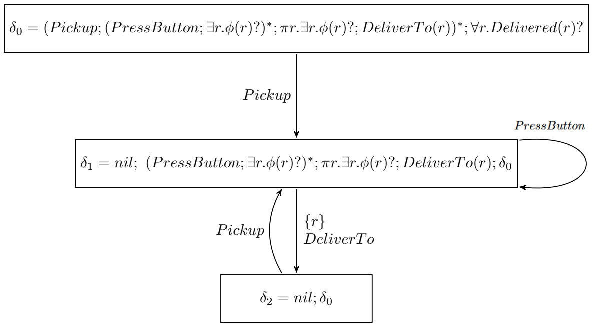
$\delta_0 = (Pickup; (PressButton; \exists r.\phi(r)?)^*; \pi r.\exists r.\phi(r)?; DeliverTo(r))^*; \forall r.Delivered(r)?$
$Pickup$
$PressButton$
$\delta_1 = nil;\ (PressButton; \exists r.\phi(r)?)^*; \pi r.\exists r.\phi(r)?; DeliverTo(r); \delta_0$
$Pickup$
$\{r\}$
$DeliverTo$
$\delta_2 = nil; \delta_0$

Figure 1: Program graph for the coffee-delivery robot.

P is the set of pick variables to be instantiated, and δ' is the remaining program; $F(\delta)$ is the condition under which δ may terminate successfully. Note that φ is always a Boolean condition over the predicate $Poss(a)$ and formulas in tests from δ and it is closed under conjunction and disjunction operators. Note also that neither φ nor F can be evaluated at this stage, because this would require knowing the domain.

Now, we can provide a definition for the program graph.

Definition 4. *Let δ_0 be a Golog program. The corresponding program graph is $\mathcal{G} = \langle \Phi \times 2^V \times \mathcal{A}, Q, q_0, \tau, \mathcal{L}\rangle$, where*

- *$\Phi \times 2^V \times \mathcal{A}$ is the alphabet, with Φ a boolean formula over tests and $Poss$ and V the set of pick variables*
- *$Q = \Gamma_{\delta_0}$ is the syntactic closure of δ_0*
- *$q_0 = \delta_0$ is the initial program*
- *$\tau(q, \varphi, P, a, q')$ iff $(\varphi, P, q') \in T(q, a)$*
- *$\mathcal{L}(q) = F(q)$ is a label assigned to q*

Example Cont. The program graph corresponding to the coffee-delivery program previously introduced is represented in Figure 1 (guards are omitted for clarity). Note that the transition from the subprogram after $PressButton$ depends on which door was opened by the environment, captured symbolically by a guarded transition of the form $\tau(q, DoorOpen(r) \wedge \neg Delivered(r), \{r\}, MoveTo(r), q')$.

Results about Program Graphs. We now summarize key properties of the program graph used to symbolically encode the structure of Golog programs. Unless otherwise specified, we will write $\mathcal{G}$ for the program graph of a given Golog program δ_0, $\mathcal{D}$ for the NDBAT over which δ_0 is executed and M for a model of $\mathcal{D}$. First, we can talk about the dimensions of the program graph:

Theorem 5. *The number of nodes and the number of edges in $\mathcal{G}$ are linear in the size of δ_0.*

Proof. By Theorem 1, the number of subprograms in Γ_{δ_0} is linear in the size of δ_0, which bounds the number of nodes in $\mathcal{G}$. Each node generates only a constant number of transitions via the recursive definition of $T(\delta, a)$, so the number of edges is also linear. □

We also relate program graphs with Golog's operational semantics. In particular, symbolic transitions in the graph correspond to executable transitions in the model, and vice versa, provided guard conditions are satisfied, and $F(\delta)$ characterizes the same terminating configurations as $Final$.

Theorem 6. *For all subprograms $\delta, \delta' \in \Gamma_{\delta_0}$, environment terms $\vec{x}, \vec{x}'$, agent action a, and situations s:*

1. *If $\tau(\delta, \varphi, P, a, \delta') \in \mathcal{G}$, $\forall z \notin P.x'_z = x_z$ and $M \models \varphi[\vec{x}'][s]$, then there exists a reaction e such that $M \models Trans(\delta, \vec{x}, s, \delta', \vec{x}', do(a[\vec{x}'](e), s))$.*
2. *For every reaction e, if $M \models Trans(\delta, \vec{x}, s, \delta', \vec{x}', do(a[\vec{x}'](e), s))$, then there exists φ s.t. $\tau(\delta, \varphi, P, a, \delta') \in \mathcal{G}$, $\forall z \notin P.x'_z = x_z$, and $M \models \varphi[\vec{x}'][s]$.*

Proof (sketch). We proceed by structural induction on the syntax of δ. The base case is $\delta = a$, where the graph contains $\tau(a, Poss(a), \emptyset, a, nil)$. By the semantics of $Trans$ and the reaction existence/independence properties, $M \models Poss(a)[\vec{x}][s]$ iff there exists e s.t. $M \models Trans(a, \vec{x}, s, nil, \vec{x}, do(a[\vec{x}](e), s))$. All other constructs follow because the graph mimics the structure of the $Trans$ rules. □

Theorem 7. *For all subprograms $\delta \in \Gamma_{\delta_0}$, environment terms $\vec{x}$ and situations s, $M \models Final(\delta, \vec{x}, s)$ iff $M \models F(\delta)[\vec{x}][s]$.*

Proof (sketch). We proceed by structural induction on δ. For $\delta = a$, both $Final(\delta, \vec{x}, s)$ and $F(\delta)$ are `False`. For $\delta = \varphi?$ they are both φ, respectively. For compound constructs, $F(\delta)$ is built to match the structure of $Final(\delta, \vec{x}, s)$, and the result follows from the inductive hypothesis. □

Now, we can prove that a full execution trace to a final configuration exists in the model if and only if there is a corresponding path in the program graph.

Theorem 8. *Let $(\delta_0, \vec{x}_0, s_0)$ be an initial configuration. Then, we have that $M \models Trans^*(\delta_0, \vec{x}_0, s_0, \delta, \vec{x}, s) \wedge Final(\delta, \vec{x}, s)$ iff there is a sequence of transitions $\tau(q_0, \phi_1, P_1, a_1, q_1), \tau(q_1, \phi_2, P_2, a_2, q_2)...\tau(q_{n-1}, \phi_n, P_n, a_n, q_n)$ in $\mathcal{G}$, environment terms $x_0, ..., x_n$ and reactions $e_1, ..., e_n$ such that $q_0 = \delta_0$, $q_n = \delta, \vec{x}_n = \vec{x}$ and $s = s_n$, and for each $i = 1, ..., n$,* (i) *$\forall z \notin P_i, x'_{i,z} = x_{i,z}$,* (ii) *$s_i = do(a_i[x_i](e_i), s_{i-1})$,* (iii) *$M \models \phi_i[x_i][s_{i-1}]$, and* (iv) *$M \models F(q_n)[x_n][s_n]$.*

Proof (sketch). It follows from Theorem 6 and 7. □

Finally, if we have a SD program, then the symbolic transition relation becomes functional.

Theorem 9. *Suppose that δ_0 is SD in S_0 wrt $\mathcal{D}$. For any transitions $\tau(q, \varphi_1, P, a, q'_1)$ and $\tau(q, \varphi_2, P, a, q'_2)$ in $\mathcal{G}$, if there exists $\vec{x}$, $\vec{x}'$ and s such that $\forall z \notin P.x'_z = x_z$ and $M \models \varphi_1[\vec{x}'][s] \wedge \varphi_2[\vec{x}'][s]$, then $q'_1 = q'_2$.*

Proof (sketch). Suppose two transitions from q in $\mathcal{G}$ for the same action a lead to q'_1 and q'_2, and both guards are satisfied in M under the same $\vec{x}$ and s. By Theorem 6, both transitions correspond to valid $Trans$ steps. Since the program is situation-determined, executing a from $(q, \vec{x}, s)$ must yield a unique successor configuration, so $q'_1 = q'_2$. □

Thus, if δ_0 is situation determined in S_0 wrt $\mathcal{D}$, the characteristic graph becomes deterministic, and we can replace the relation $\tau(q, \varphi, P, a, q')$ by the function $\tau(q, \varphi, P, a) = q'$.

Synthesis and Strategic Reasoning.

We now turn to the problem of synthesizing strategies that allow an agent to successfully execute a Golog program in the presence of adversarial nondeterminism. Building on the symbolic program graphs introduced earlier, we integrate these with a model of the domain to define a game arena over which synthesis can be carried out.

TS-based Synthesis for First Order Domains We adopt a transition system–based synthesis approach, where the program graph and the domain model are combined into a game structure that encodes the agent-environment interaction. Consider a nondeterministic domain $\mathcal{D}_M = \langle 2^{\mathcal{F}}, \mathcal{A}, s_0, \rho, \alpha \rangle$ where:

- $2^{\mathcal{F}}$ is the set of states, with $s_0 \in 2^{\mathcal{F}}$ the initial state
- $\alpha(s) \subseteq \mathcal{A}$ represents action preconditions
- $\rho(s, a, s')$ with $a \in \alpha(s)$ is the transition relation

Note that for any NDBAT $\mathcal{D}$ and model $M \models \mathcal{D}$, there is a corresponding nondeterministic domain $\mathcal{D}_M$. We can define a game arena constructed by the cross product of the program graph $\mathcal{G}$ and the nondeterministic domain.

Definition 10. *Let δ_0 be a program and $\mathcal{D}_M$ a ND domain. The cross product is a tuple:*

$$\langle \mathcal{A} \times Q \times \mathcal{O}^n \times 2^{\mathcal{F}}, Q \times \mathcal{O}^n \times 2^{\mathcal{F}}, (\delta_0, \vec{x}_0, s_0), Tr, Fin \rangle$$

where

- $\mathcal{A} \times Q \times \mathcal{O}^n \times 2^{\mathcal{F}}$ *is the alphabet*
- $Q \times \mathcal{O}^n \times 2^{\mathcal{F}}$ *is a set of states*
- $(\delta_0, \vec{x}_0, s_0)$ *is the initial state*
- $Tr((\delta, \vec{x}, s), (a, \delta', \vec{x}^P, s')) = (\delta', \vec{x}', s')$ *is the transition function where* (i) $\exists \varphi . \tau(\delta, \varphi, P, a, \delta')$, (ii) $\forall z \notin P . x'_z = x_z$ *and* $\forall z \in P . x'_z = x^P_z$, (iii) $s \models \varphi[\vec{x}']$, *and* (iv) $\rho(s, a[\vec{x}'], s')$
- $Fin = \{(\delta, \vec{x}, s) \mid s \models F(\delta)[\vec{x}]\}$ *is the set of final states*

This can be viewed as a game arena with alphabet $\Sigma = \mathcal{A} \times Q \times \mathcal{O}^n \times 2^{\mathcal{F}}$, where the agent controls the action $\mathcal{A}$, the remaining program Q, and the binding of the pick variables $\mathcal{O}^n$, and where the environment controls the next state $2^{\mathcal{F}}$. A game strategy is a function $\kappa : G \to \mathcal{A} \times Q \times \mathcal{O}^n$ mapping states of the game $g \in G$ to agent actions, remaining programs, and bindings for the pick variables. The set of plays induced by a game strategy κ in the game arena A, $Play(\kappa, A)$, is the set of all plays $g_0, g_1, ... \in G^\omega$ such that g_0 is the initial state of A and there exists an environment state s' such that $Tr(g_i, (\kappa(g_i), s')) = g_{i+1}$. A game strategy κ is winning in A if, for every play $g_0, g_1, ...$ in $Play(\kappa, A)$, there exists some i such that $Fin(g_i)$.

Agent Control and Strategic Reasoning.[1] In order to represent the ability of the agent to execute an agent program in a ND domain, (DL21) introduce $AgtCanForceBy(\delta, f, s)$ as an adversarial version of Do in presence of environment reactions. This predicate states that strategy f, a function from situations to agent actions (including the special action $stop$), executes SD Golog agent program δ in situation s considering its nondeterminism angelic, as in the standard Do, but also considering the nondeterminism of environment devilish/adversarial. Here is a version of $AgtCanForceBy$ that considers the presence of an environment term $\vec{x}$:

$$\begin{array}{l}
AgtCanForceBy(\delta, \vec{x}, f, s) \doteq \forall P.[\ldots \supset P(\delta, \vec{x}, s)] \\
\text{ where } \ldots \text{stands for} \\
{[f(s) = stop \land Final(\delta, \vec{x}, s) \supset P(\delta, \vec{x}, s)] \land} \\
{[\exists a.(f(s) = a \neq stop \land} \\
\exists e.\exists \delta'.\exists \vec{x}'.Trans(\delta, \vec{x}, s, \delta', \vec{x}', do(a[\vec{x}'](e), s)) \land \\
\forall e.(\exists \delta'.\exists \vec{x}'.Trans(\delta, \vec{x}, s, \delta', \vec{x}', do(a[\vec{x}'](e), s))) \supset \\
\exists \delta'.\exists \vec{x}'.Trans(\delta, \vec{x}, s, \delta', \vec{x}', do(a[\vec{x}'](e), s)) \land \\
P(\delta', \vec{x}', do(a[\vec{x}'](e), s)) \supset P(\delta, \vec{x}, s)]
\end{array}$$

We say that predicate $AgtCanForce(\delta, \vec{x}, s)$ holds iff there exists a strategy f s.t. $AgtCanForceBy(\delta, \vec{x}, f, s)$ holds.

Theorem 11. *For any SD program δ_0, subprogram $\delta \in \Gamma_{\delta_0}$, environment term $\vec{x}$, situation s and strategy f, we have that:*

$M \models AgtCanForceBy(\delta, \vec{x}, f, s)$
iff $(\delta, \vec{x}, s)$ is in the least set P_μ such that
if $f(s)$=$stop \land M \models Final(\delta, \vec{x}, s)$, then $(\delta, \vec{x}, s) \in P_\mu$
if $f(s)$=$a, a \neq stop$ and there exists e, δ', and $\vec{x}'$ s.t.
$M \models Trans(\delta, \vec{x}, s, \delta', \vec{x}', do(a[\vec{x}'](e), s))$
and for all e the existence of δ' and $\vec{x}'$ s.t.
$M \models Trans(\delta, \vec{x}, s, \delta', \vec{x}', do(a[\vec{x}'](e), s))$ implies
$(\delta', \vec{x}', do(a[\vec{x}'](e), s)) \in P_\mu$, then $(\delta, \vec{x}, s) \in P_\mu$
iff $(\delta, \vec{x}, s)$ is in the least set P_μ such that
if $f(s)$=$stop$ and $M \models F(\delta)[\vec{x}][s]$, then $(\delta, \vec{x}, s) \in P_\mu$
if $f(s)$=$a, a \neq stop$ and there exists φ, δ', and $\vec{x}'$ s.t.
$\tau(\delta, \varphi, P, a, \delta')$, for all z not in $P, x'_z = x_z$ and
$M \models \varphi[\vec{x}'][s]$, and for all e the existence of δ' and $\vec{x}'$
s.t. $\tau(\delta, \varphi, P, a, \delta')$ and $M \models \varphi[\vec{x}'][s]$ implies that
$(\delta', \vec{x}', do(a[\vec{x}'](e), s))$ is in P_μ, then $(\delta, \vec{x}, s) \in P_\mu$

Proof. This follows directly from Theorems 6 and 7. □

Finally, the next result shows that the existence of a winning strategy in the game arena means that the agent is able to execute the program starting from the initial situation in the underlying situation calculus model.

Theorem 12. *Let δ_0 be a SD Golog program, $\mathcal{D}$ an NDBAT, M a model of $\mathcal{D}$, $\mathcal{D}_M$ the nondeterministic domain corresponding to M, and A the game arena generated by δ_0 and $\mathcal{D}_M$. Then $M \models AgtCanForce(\delta_0, \vec{x}, S_0)$ iff there exists a winning strategy in A.*

Proof (sketch). The agent can force successful execution in the model iff there is a strategy s.t. all executions lead to a final configuration. The executions correspond to paths in the game arena that reach a final state. Hence, the strategy exists iff there is a winning strategy in the arena. □

Example Cont. In the coffee delivery domain, the agent aims to ensure that every office eventually receives a cup of coffee, despite not controlling which door opens after each button press. The synthesis problem consists of coming up with a strategy f such that the agent can force the successful termination of the program δ from an initial situation S_0 where no coffee has been delivered yet. A valid strategy repeatedly executes the $PressButton$ action until some room r satisfies both $DoorOpen(r)$ and $\neg Delivered(r)$, at which

[1] These results can be extended to non-SD programs.

point it moves to r and delivers the coffee. The nondeterminism due to the environment's control is handled by the agent's loop structure, which scans for a suitable target once the environment has revealed an open room.

Fixpoint Verification Procedure. To implement the predicate $AgtCanForceBy$ in practice, one can adopt a sound symbolic fixpoint procedure that explores the executions of the agent program under all possible environment reactions. (De Giacomo, Lespérance, and Pearce 2010, 2016) present such a procedure for verifying first-order (FO) μ-ATL properties (Alur, Henzinger, and Kupferman 2002; Bradfield and Stirling 2007) over game structures. In particular, they develop a procedure $[[\Psi]]$ for verifying a property Ψ by labeling nodes in a program graph, leveraging regression and fixpoint approximates (Tarski 1955).

Here, we want to verify $AgtCanForce$, which can be defined as a least fixpoint formula in their logic. The labeling of a program graph $\mathcal{G}$ is denoted by $\mathcal{Z}$ and produces a set $\{\langle\delta, \phi\rangle \mid \delta \in \mathcal{G}\}$ where each node δ is associated with a FO formula ϕ that characterizes the situations s and environment terms $\vec{x}$ in which $AgtCanForce(\delta, \vec{x}, s)$ is satisfied, i.e., the situations starting from which the agent can execute δ by following a certain strategy. We call a configuration $(\delta, \vec{x}, s)$ winning if we have a formula ϕ that guarantees $AgtCanForce$ for that configuration, i.e. $\mathcal{D} \models \phi(s)$, and if we fix δ and $\vec{x}$, a strategy is winning if the configuration $(\delta, \vec{x}, s)$ is winning.

For each program, we need to compute the formula ϕ that captures all the configurations that are winning for $AgtCanForce$, and we can proceed iteratively knowing that if in configuration $(\delta, \vec{x}, s)$ the agent can execute an action such that all possible resulting configurations $(\delta', \vec{x}', s')$ are winning, then $(\delta, \vec{x}, s)$ is also winning. We can attempt to verify $AgtCanForce$ by a least fixpoint procedure that iteratively generates better approximations of the labeling.

The least fixpoint approximation procedure is given by:

```
$\mathcal{Z} \leftarrow [[False]]$
$\mathcal{Z}_{new} \leftarrow Final(\mathcal{Z}) \vee PreAdv(\mathcal{Z})$
while$(\mathcal{Z} \neq \mathcal{Z}_{new})$ {
  $\mathcal{Z} \leftarrow \mathcal{Z}_{new}$
  $\mathcal{Z}_{new} \leftarrow Final(\mathcal{Z}) \vee PreAdv(\mathcal{Z})$}
```

where we use $\mathcal{Z} \neq \mathcal{Z}_{new}$ as an abbreviation for $\mathcal{D}_{ca} \not\models \bigwedge_{\langle\delta,\phi\rangle\in\mathcal{Z},\langle\delta,\phi_{new}\rangle\in\mathcal{Z}_{new}} \phi \equiv \phi_{new}$. Now we just need to define the adversarial preimage of $\mathcal{Z}$:

$$\begin{aligned}
&PreAdv(\mathcal{Z}) = \{\langle\delta, \phi\rangle \mid \delta \in \mathcal{G} \text{ and} \\
&\quad \phi = \textstyle\bigvee_{\tau(\delta,\varphi,P,a,q')\in\mathcal{G},\langle q',\phi'\rangle\in\mathcal{Z}} \\
&\qquad \exists\vec{x}^P.[\forall z \notin P.x'_z = x_z \wedge \forall z \in P.x'_z = x^P_z] \wedge \\
&\qquad \varphi[\vec{x}'] \supset \forall e \in React.\mathcal{R}(\phi'[do(a[\vec{x}'](e), s)])\}
\end{aligned}$$

where $\mathcal{R}$ is one-step regression applied to the labels. Intuitively, $PreAdv$ computes the set of configurations from which the agent can take an action s.t. all environment reactions lead to winning configurations. Note that, in general, the procedure is incomplete due to undecidability and the infiniteness of FO domains. Termination can be ensured by restricting to the propositional setting, to a decidable fragment of FOL or to the bounded situation calculus (De Giacomo, Lespérance, and Patrizi 2016; De Giacomo et al. 2021).

Theorem 13. *Let (δ_0, x_0, S_0) be an initial configuration. If the labeling procedure terminates and $\langle\delta_0, \phi\rangle$ is in the returned set, then $\mathcal{D} \models AgtCanForce(\delta_0, x_0, S_0)$ if and only if $\mathcal{D}_{S_0} \cup \mathcal{D}_{ca} \models \phi[S_0]$.*

Proof (sketch). By induction on the fixpoint construction. The labeling procedure mirrors the characterization of $AgtCanForce$ (Thm 11), using $Final$ for base cases and $PreAdv$ to propagate winning configurations. Soundness follows from the correctness of these constructions. □

Using Programs to Constrain the Environment

Handling Environment Programs. Our framework naturally extends to scenarios where the environment's behavior is constrained by its own program. We assume that the environment program contains only one action, namely $DoReaction$, taking the reaction e selected by the environment as a parameter, and that it can refer to the current action selected by the agent, which is denoted by the special symbol ac. Here, we need to adapt the semantics of the programs and define $Trans$ and $Final$ for both the agent program, denoted δ_a, and the environment program, denoted δ_e. Below is a sketch of $Trans_a$ and $Trans_e$:

$$\begin{aligned}
&\mathit{Trans}_a(a, \vec{x}, s, \delta'_a, \vec{x}', a) \equiv \\
&\quad Poss_{ag}(a[\vec{x}], s) \wedge \delta' = nil \wedge \vec{x}' = \vec{x} \\
&\mathit{Trans}_a(\varphi?, \vec{x}, s, \delta'_a, \vec{x}', a) \equiv \texttt{False} \\
&\dots \\
&\\
&\mathit{Trans}_e(\mathit{DoReaction}(e), \vec{x}, a, s, \delta'_e, \vec{x}', e) \equiv \\
&\quad Poss(a(e), s) \wedge \delta'_e = nil \wedge \vec{x}' = \vec{x} \\
&\mathit{Trans}_e(\varphi?, \vec{x}, a, s, \delta'_e, \vec{x}', e) \equiv \texttt{False} \\
&\dots
\end{aligned}$$

$Trans_a$ is the same as in the original definition, but it has the action selected by the agent a as its last parameter instead of the next situation s'; $Trans_e$ is similar, but it takes as input the agent action a and selects the environment reaction e (because the reaction depends on the action chosen by the agent), and again drops s'. $Final_a$ and $Final_e$ are equal to the original definition, except that $Final_e$ has a among its parameters, and in the final condition for tests we substitute the symbol ac with the actual action a:

$$\begin{aligned}
&\mathit{Final}_e(\mathit{DoReaction}(e), \vec{x}, a, s) \equiv \texttt{False} \\
&\mathit{Final}_e(\varphi?, \vec{x}, a, s) \equiv \varphi[\vec{x}, ac/a][s] \\
&\dots
\end{aligned}$$

Finally, system transitions result from the interleaved execution of the agent program and the environment program:

$$\begin{aligned}
&Trans(\delta_a, \vec{x}, \delta_e, \vec{y}, s, \delta'_a, \vec{x}', \delta'_e, \vec{y}', s') \equiv \\
&\quad Trans_a(\delta_a, \vec{x}, s, \delta'_a, \vec{x}', a) \wedge \\
&\quad Trans_e(\delta_e, \vec{y}, a[\vec{x}'], s, \delta'_e, \vec{y}', e) \wedge \\
&\quad s' = do(a[\vec{x}'](e), s)
\end{aligned}$$

A final configuration is reached when both programs have terminated:

$$\begin{aligned}
&Final(\delta_a, \vec{x}, \delta_e, \vec{y}, s) \equiv \\
&\quad Final(\delta_a, \vec{x}, s) \wedge Final(\delta_e, \vec{y}, a, s)
\end{aligned}$$

Note that we could easily construct a program graph for the environment programs, and if we compute the cross product between the program graph of δ_a and the program graph of δ_e, we still obtain a program graph. It means that everything we have shown carries over even in this setting. This leads to a joint fixpoint definition of agent-environment interaction. We extend the predicate $\mathit{AgtCanForceByIf}$ to capture whether an agent strategy f can enforce the successful execution of δ_a in the presence of an adversarial but constrained environment running δ_e.

$$\begin{array}{l}
\mathit{AgtCanForceByIf}(\delta_a, \vec{x}, \delta_e, \vec{y}, f, s) \doteq \\
\quad \forall P.[\ldots \supset P(\delta_a, \vec{x}, \delta_e, \vec{y}, s)], \text{ where } \ldots \text{stands for} \\
[(f(s) = \mathit{stop} \land Final(\delta_a, \vec{x}, s)) \supset P(\delta_a, \vec{x}, \delta_e, \vec{y}, s)] \land \\
[\exists a.(f(s) = a \neq \mathit{stop} \land \exists e.\exists \vec{x}', \vec{y}'.\exists \delta'_a, \delta'_e. \\
Trans(\delta_a, \vec{x}, \delta_e, \vec{y}, s, \delta'_a, \vec{x}', \delta'_e, \vec{y}', do(a[\vec{x}'](e), s)) \land \\
\forall e.\forall \vec{y}'.(\exists \delta'_a, \delta'_e.\exists \vec{x}. \\
Trans(\delta_a, \vec{x}, \delta_e, \vec{y}, s, \delta'_a, \vec{x}', \delta'_e, \vec{y}', do(a[\vec{x}'](e), s)) \supset \\
P(\delta'_a, \vec{x}', \delta'_e, \vec{y}', do(a[\vec{x}'](e), s)) \supset P(\delta_a, \vec{x}, \delta_e, \vec{y}, s)]
\end{array}$$

We can use an analogous fixpoint procedure as before to do synthesis for this.

Example Cont. In our running example, the environment program can specify how the environment selects which door to open in response to the robot's $PressButton$ action. In general, the environment reaction will be $Success$ if the action is not $PressButton$, and it will be door to be opened ($\mathit{openOfficeA}$, $\mathit{openOfficeB}$, ...) otherwise:

$$\begin{array}{rl}
\delta_e = & (((\pi x.ac \neq PressButton \land x = Success?; \\
& \quad DoReaction(x))^*); \\
& (\pi y.ac = PressButton \land CanOpen(y)?; \\
& \quad DoReaction(y)))^*
\end{array}$$

As explained before, we ensure the environment not to be "unfair" using the fluent $CanOpen$, which prevents the environment from maintaining a door closed indefinitely.

Discussion

We have presented a general approach for strategic reasoning over high-level agent programs in the situation calculus, focusing on the first-order nondeterministic setting, where programs constrain the behaviors of both the agent and the environment. This is related to the broader problem of planning with domain control knowledge (DCK), where traditional goal specifications are replaced by procedural, action-centric descriptions that express preferences over the manner in which goals are achieved. (Baier, Fritz, and McIlraith 2007; Baier et al. 2008) addressed this challenge by compiling DCK and temporally extended user preferences into planning problem instances, thereby ensuring that only action sequences adhering to the procedural constraints are generated. They proved how Golog-like programs can be translated into PDDL, enabling the use of domain-independent heuristic planners. (Fritz, Baier, and McIlraith 2008) extended this compilation approach to ConGolog, a variant of Golog that supports explicit concurrency between programs.

In the case of Golog, the progression of procedural control knowledge is modeled using a nondeterministic finite automaton with ϵ-transitions (ϵ-NFA), where ϵ denotes a transition that does not consume any action. However, the environment remains entirely unconstrained, which significantly limits the ability to model realistic scenarios involving adversarial or unpredictable behavior. Moreover, only angelic nondeterminism originating from favorable program choices is considered, while devilish nondeterminism, stemming from hostile environmental influences, is not captured. Consequently, their frameworks are not suitable for modeling or reasoning about adversarial environments.

Our framework is also related to the work of (HC25), which investigates a game-theoretic approach to synthesizing strategies for agents whose behavior is guided by a Golog program and constrained by an LTL$_f$ temporal specification. In their approach, a finite game arena is constructed that encodes all possible executions of the Golog program under environment nondeterminism, while concurrently tracking the satisfaction of the LTL$_f$ temporal goal. To guarantee decidability, they restrict the underlying basic action theories to C^2 (Gradel, Otto, and Rosen 1997; Zarrieß and Claßen 2016), a decidable two-variable fragment of FOL, and similarly constrain the temporal formulas by substituting propositions with quantifier-free C^2 first-order formulas. While these restrictions ensure computational tractability, they significantly limit the expressive power of the framework.

Although their work provides an important contribution toward bridging agent programming with temporal logic synthesis, its applicability is limited to very simple scenarios. The experimental evaluation offers a proof-of-concept but reveals substantial scalability challenges. In contrast, our framework is capable of addressing substantially more complex domains while being simpler and more natural. We provide formal results on the relation between the execution paths in the program graph and Golog transition semantics (that can be translated also to characteristic graphs), and between TS-based synthesis and agent's ability to execute a program in a first-order domain, as well as a fixpoint evaluation procedure for performing synthesis. Furthermore, they use Golog programs as constraints for the entire system, while our approach is more flexible and richer since it includes distinct programs constraining the agent and the environment, opening to possible extension to multi-agent system settings.

Golog provides procedural task specifications to contrast with declarative ones such as LTL$_f$/LDL$_f$ (De Giacomo and Vardi 2013, 2015). When the object domain is finite and the initial state is known, our method is sound and complete. Its cost is polynomial in the game arena, which in turn is polynomial in the initial program, whereas LTL$_f$/LDL$_f$ synthesis is 2EXPTIME in general. A natural direction for future work is therefore to investigate finite-state settings and apply concrete synthesis algorithms, enabling a direct comparison between procedural programs and temporally extended specifications like LDL$_f$.

Another possible direction is to explore alternative ways of constraining the environment, for instance by using LTL trace constraints as in (De Giacomo, Lespérance, and Mancanelli 2025), where an LTL formula $Cstr$ is imposed on environment behaviours and $\mathit{AgtCanForceByIf}$ is defined to express that an agent can force δ along all paths satisfying $Cstr$ when following f from s.

Acknowledgments

This work has been supported by the ERC Advanced Grant WhiteMech (No. 834228), the PRIN project RIPER (No. 20203FFYLK), the PNRR MUR project FAIR (No. PE0000013), the UKRI Erlangen AI Hub on Mathematical and Computational Foundations of AI (No. EP/Y028872/1), the Italian National Ph.D. on Artificial Intelligence at Sapienza University of Rome, the National Science and Engineering Research Council of Canada, and York University.

References

Abadi, M.; Lamport, L.; and Wolper, P. 1989. Realizable and Unrealizable Specifications of Reactive Systems. In *ICALP*, volume 372 of *Lecture Notes in Computer Science*, 1–17. Springer.

Alur, R.; Henzinger, T. A.; and Kupferman, O. 2002. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5): 672–713.

Baier, J. A.; Fritz, C.; Bienvenu, M.; and McIlraith, S. A. 2008. Beyond Classical Planning: Procedural Control Knowledge and Preferences in State-of-the-Art Planners. In *AAAI*, 1509–1512.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners. In *ICAPS*, 26–33.

Bradfield, J.; and Stirling, C. 2007. 12 Modal mu-calculi. *Studies in logic and practical reasoning*, 3: 721–756.

Claßen, J. 2013. *Planning and Verification in the agent language Golog*. Ph.D. thesis, RWTH Aachen University.

Claßen, J.; and Lakemeyer, G. 2008. A Logic for Non-Terminating Golog Programs. In *KR*, 589–599.

De Giacomo, G.; Felli, P.; Logan, B.; Patrizi, F.; and Sardina, S. 2021. Situation Calculus for Controller Synthesis in Manufacturing Systems with First-Order State Representation. *Artif. Intell.*

De Giacomo, G.; and Lespérance, Y. 2021. The Nondeterministic Situation Calculus. In *KR*, 216–226.

De Giacomo, G.; Lespérance, Y.; and Mancanelli, M. 2025. Situation Calculus Temporally Lifted Abstractions for Generalized Planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 14848–14857.

De Giacomo, G.; Lespérance, Y.; and Muise, C. J. 2012. On supervising agents in situation-determined ConGolog. In *AAMAS*, 1031–1038. IFAAMAS.

De Giacomo, G.; Lespérance, Y.; and Patrizi, F. 2016. Bounded situation calculus action theories. *Artif. Intell.*, 237: 172–203.

De Giacomo, G.; Lespérance, Y.; Patrizi, F.; and Sardina, S. 2016. Verifying ConGolog programs on bounded situation calculus theories. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

De Giacomo, G.; Lespérance, Y.; and Pearce, A. R. 2010. Situation Calculus-Based Programs for Representing and Reasoning about Game Structures. *Proc. of KR*, 445–455.

De Giacomo, G.; Lespérance, Y.; and Pearce, A. R. 2016. Situation Calculus Game Structures and GDL. In *ECAI*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, 408–416. IOS Press.

De Giacomo, G.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *Ijcai*, volume 13, 854–860.

De Giacomo, G.; and Vardi, M. Y. 2015. Synthesis for LTL and LDL on Finite Traces. In *IJCAI*, 1558–1564. AAAI Press.

Fritz, C.; Baier, J. A.; and McIlraith, S. A. 2008. Congolog, sin trans: Compiling congolog into basic action theories for planning and beyond. In *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning*, 600–610.

Geffner, H.; and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated planning and acting*. Cambridge University Press.

Gradel, E.; Otto, M.; and Rosen, E. 1997. Two-variable logic with counting is decidable. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*, 306–317. IEEE.

Hofmann, T.; and Claßen, J. 2025. LTLf Synthesis on First-Order Agent Programs in Nondeterministic Environments. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 14976–14986.

Levesque, H. J.; and Lakemeyer, G. 2001. *The logic of knowledge bases*. Mit Press.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 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.

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

Pnueli, A.; and Rosner, R. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 179–190.

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

Tarski, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*.

Zarrieß, B.; and Claßen, J. 2016. Decidable verification of Golog programs over non-local effect actions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.