Reactive Synthesis for Golog Specifications in the Propositional Situation Calculus

Giuseppe De Giacomo[1,2] , **Yves Lespérance**[3] , **Matteo Mancanelli**[2] , **Gianmarco Parretti**[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, parretti}@diag.uniroma1.it

Abstract

Golog programs over Situation Calculus action theories were introduced as a specification of desired agent behavior, very much like temporally extended goals in planning, but with a focus on procedural aspects typical of programs. In the words of the original paper: "Golog allows the programmer to strike a compromise between the often computationally infeasible classical planning task, in which a plan must be deduced entirely from scratch, and detailed programming, in which every little step must be specified." In this paper, we study temporal synthesis with Golog programs as specifications over nondeterministic propositional action theories. We show that Golog has the same expressive power as linear dynamic logics on finite traces (LDLf), namely that of regular languages or monadic second-order logic (MSO) over finite traces, while exhibiting a markedly lower synthesis complexity: synthesis can be performed by constructing a polynomial-size program graph and taking its cross-product with the domain, whereas LDLf synthesis requires building a deterministic automaton of worst-case doubly exponential size. This advantage is confirmed experimentally.

1 Introduction

Golog programs over Situation Calculus action theories (Reiter 2001) were introduced as a specification of desired agent behavior, very much like temporally extended goals in planning, but with a focus on procedural aspects typical of programs. In the words of the original paper (Levesque et al. 1997): "Golog allows the programmer to strike a compromise between the often computationally infeasible classical planning task, in which a plan must be deduced entirely from scratch, and detailed programming, in which every little step must be specified." While these words were originally written with running Golog programs over a deterministic action theory in mind, we show that they assume a very precise meaning also when we consider nondeterministic action theories (De Giacomo and Lespérance 2021).

In this paper, we focus on propositional nondeterministic basic action theories (NDBATs). This allows us to study agent decision making within Golog programs under the lens of temporal synthesis over finite traces, which has become popular for temporal specification expressed in Linear Temporal Logic over finite traces (LTL_f) and Linear Dynamic Logic over finite traces (LDL_f) (De Giacomo and Vardi 2013; De Giacomo and Vardi 2015; Camacho, Bienvenu, and McIlraith 2019; Duret-Lutz et al. 2025).

Since we use Golog as an alternative language for linear temporal specifications, we restrict our attention to *situation-determined* programs (De Giacomo, Lespérance, and Muise 2012), i.e., where the remaining program after an action must be determined by the resulting situation. This allows us to focus on traces that do not include remaining programs, and so are directly comparable with those handled by LTL_f/LDL_f. Note that situation determinacy is a semantical notion that might be difficult to check in general. Hence, we define syntactic situation determinacy, which can be easily checked without accessing the action theory.

Golog is often used without recursive procedures. In that case, situation-determined Golog programs would correspond to *1-deterministic regular languages* (Brüggemann-Klein and Wood 1998; Han and Wood 2008; Caron, Mignot, and Miklarz 2017), which are a proper subclass of regular languages. Here, we do allow for procedures with tail recursion, so that we avoid compromising the expressive power. Indeed, we show that our variant of Golog has the same expressive power as LDL_f, namely that of regular languages or monadic second-order logic (MSO) over finite traces.

Crucially, due to its more procedural nature (Levesque et al. 1997; Baier, Fritz, and McIlraith 2007), Golog exhibits a markedly lower synthesis complexity compared with, e.g., LDL_f. Indeed, we prove that synthesis can be performed by constructing a polynomial-size program graph and taking its cross-product with the domain, whereas LDL_f synthesis requires the construction of a deterministic automaton of worst-case doubly exponential size. We then implement this using an automata-based approach, and compare performance experimentally against LDL_f.

2 Related Work

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 address nondeterministic domains. Our work complements (De Giacomo, Lespérance, and Mancanelli 2026), which studied Golog synthesis in nondeterministic environments in a first-order setting. It provided a sound but incomplete procedure to synthesise strategies, but did not

study actual implementation. Here, we focus on the propositional setting, and develop sound and complete procedures based on LTL$_f$/LDL$_f$ synthesis methods.

Our work is closely related to that of (Hofmann and Claßen 2025), which also addresses Golog synthesis under nondeterministic environments. Their framework is not limited to finite domains, but instead assumes a basic action theory expressed in the C^2 decidable fragment of FOL. As a result, it is more complex, and the implementation exhibits very limited scalability, as acknowledged by the authors. Here we study a simpler propositional setting and develop actual synthesis algorithmic techniques that are as scalable as those in the state-of-the-art LTL$_f$/LDL$_f$ synthesis tools (Zhu and Favorito 2025; Duret-Lutz et al. 2025).

3 Preliminaries

The *situation calculus* is a predicate logic language designed for specifying and reasoning about dynamically evolving worlds (McCarthy and Hayes 1969; Reiter 2001). Here, we consider the propositional variant of the situation calculus, in which we do not have explicit objects. 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 action a in s. Fluents are atomic propositions varying with the situation, which are represented as predicates that take a situation term as their (only) argument. In this language, we define domains represented by a *basic action theory (BAT)*, where successor state axioms encode the causal laws of the domain (Reiter 2001). $Poss(a, s)$ is used to state that a is executable in s; $executable(s)$ means that every action in s was executable in the situation in which it occurred. We can always 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.

Following (De Giacomo and Lespérance 2021), we consider an extension of the situation calculus handling nondeterministic actions. For any agent action in a nondeterministic domain, there can be several different outcomes, depending on how the environment behaves. This is modeled by adding an environment reaction parameter e, which can take only finitely many values, ranging over a new sort *Reaction*. We call the reaction-suppressed version of the action a an *agent action*, and the full version $a(e)$ a *system action*.

Given an alphabet $\mathscr{A}$, formed by a finite set of fluents $\mathcal{F}$, a finite set of actions $\mathcal{A}$, and a finite set of reactions $\mathcal{E}$, a *nondeterministic basic action theory (NDBAT)* $\mathcal{D}$ over $\mathscr{A}$ is a special kind of BAT, which is the union of the following disjoint sets: foundational, domain independent, axioms of the situation calculus (Σ), unique name axioms for actions and domain closure for action types ($\mathcal{D}_{ca}$), axioms describing the initial situation ($\mathcal{D}_{S_0}$), successor state axioms for each fluent $F \in \mathcal{F}$ describing how (system) actions change the fluents ($\mathcal{D}_{ssa}$), and (system) action precondition axioms ($\mathcal{D}_{poss}$) as well as agent action preconditions using $Poss_{agt}$ ($\mathcal{D}_{poss_{agt}}$), for each action $a \in \mathcal{A}$, stating when the system (resp. agent) action can occur. The theory must entail the *reaction independence* requirement (i.e., $\forall e.Poss(a(e), s) \supset Poss_{agt}(a, s)$), and the *reaction existence* requirement (i.e., $Poss_{agt}(a, s) \supset \exists e.Poss(a(e), s)$).

4 Golog as a Trace Specification Language

Golog. Golog is a high-level language for writing programs that are executed over NDBATs. Obviously, a Golog program can only be executed on an NDBAT that shares the same alphabet $\mathscr{A}$. We assume wlog that this is always the case. The syntax of Golog programs has the following form:

$$\delta ::= a \mid \varphi? \mid \delta_1; \delta_2 \mid \delta_1 | \delta_2 \mid \delta^*$$

where a is an *agent action*, and $\varphi?$ is a test for the condition φ to hold in the current situation. Note that φ is a situation-suppressed formula, and we denote by $\varphi[s]$ the formula obtained by restoring the s into all fluents in φ. As in (Claßen and Lakemeyer 2008; De Giacomo, Lespérance, and Pearce 2010), we assume that tests do not yield transitions. Moving to complex constructs, $\delta_1; \delta_2$ denotes the sequential execution of δ_1 and δ_2, $\delta_1 | \delta_2$ denotes the nondeterministic choice (under the agent control) of executing δ_1 or δ_2, and δ^* denotes the execution of δ zero or more times (again the choice of how many times δ is executed is under the agent control). We do not include the pick operator $\pi x.\delta$, denoting the execution of the program δ for some nondeterministic choice of the object variable x, since in propositional action theories we do not have objects.[1] As usual, we define the abbreviations **if** ϕ **then** δ_1 **else** $\delta_2 \doteq \phi?; \delta_1 | \neg\phi?; \delta_2$ and **while** ϕ **do** $\delta \doteq (\phi?; \delta)^*; \neg\phi?$. We use nil as an abbreviation for `True`? to denote the empty program.

We extend the above constructs by allowing for recursive procedures (De Giacomo, Lespérance, and Levesque 2000). Specifically, a Golog program can include procedure calls of a finite number n of procedures, defined as

$$P_i : body(P_i), \qquad \text{for } i = 1, \ldots, n.$$

We call P_i the *procedure name*, and $body(P_i)$ the *procedure body*. We require the procedure body to have the form

$$body(P_i) = \delta_{i,1}; Q_{i,1} \mid \cdots \mid \delta_{i,m}; Q_{i,m}$$

where $\delta_{i,j}$ does not include procedure calls, and $Q_{i,j}$ is either nil or a procedure call P_j with $j = 1, \ldots, n$. In the latter case we require that $\delta_{i,j}$ cannot be considered terminated (i.e., it must be the case that $\neg Final(\delta_{i,j}, s)$ holds for every s, see below). Note that we do allow for *tail recursion* only. Tail recursion suffices to capture all regular languages (see Section 5), while avoiding unbounded control stacks. Since we are in the propositional case, there are no parameters in procedures. Hence, a *procedure call* consists of the procedure name P_i itself, and intuitively activates the execution of $body(P_i)$. Given a finite set of procedure definitions $Procs$, we will consider executions of a Golog program δ over a NDBAT $\mathcal{D}$, such that all procedure calls in δ and $Procs$ are to procedures defined in $Procs$.

[1] Note that we get a propositional action theory also in the case where we have finitely many objects. Then, the pick operator is not really necessary since it can be expressed as a nondeterministic branch $\pi x.\delta(x) \equiv \delta(o_1) \mid \delta(o_2) \mid ... \mid \delta(o_n)$.

Example. This is the program for the task of putting every block on the table in a nondeterministic BlocksWorld, where every other time a block is put on the table another one may be stacked by the environment:

$$P_0 : \bigwedge_{Block_b} OnTable_b?; nil \mid \bigvee_{Block_{b_1}, Block_{b_2}} On_{b_1,b_2}?; unstack_{b_1,b_2}; P_0$$

Golog Semantics. The semantics of Golog can be specified as usual in terms of single-steps, using the predicates $Trans$ and $Final$ (De Giacomo, Lespérance, and Levesque 2000). *Trans* specifies that one step of the program δ in situation s leads to situation s' with δ' remaining to be executed:

$$\begin{aligned}
&\mathit{Trans}(a, s, \delta', s') \equiv \\
&\quad \exists e. Poss(a(e), s) \wedge \delta' = nil \wedge s' = do(a(e), s) \\
&\mathit{Trans}(\varphi?, s, \delta', s') \equiv \texttt{False} \\
&\mathit{Trans}(\delta_1; \delta_2, s, \delta', s') \equiv \mathit{Trans}(\delta_1, s, \delta_1', s') \wedge \\
&\quad \delta' = \delta_1'; \delta_2 \vee \mathit{Final}(\delta_1, s) \wedge \mathit{Trans}(\delta_2, s, \delta', s') \\
&\mathit{Trans}(\delta_1 | \delta_2, s, \delta', s') \equiv \\
&\quad \mathit{Trans}(\delta_1, s, \delta', s') \vee \mathit{Trans}(\delta_2, s, \delta', s') \\
&\mathit{Trans}(\delta^*, s, \delta', s') \equiv \mathit{Trans}(\delta, s, \delta'', s') \wedge \delta' = \delta''; \delta^* \\
&\mathit{Trans}(P_i, s, \delta', s') \equiv \mathit{Trans}(body(P_i), s, \delta', s')
\end{aligned}$$

Final specifies that the program may terminate in a given situation, and is defined as:

$$\begin{aligned}
&\mathit{Final}(a, s) \equiv \texttt{False} \\
&\mathit{Final}(\varphi?, s) \equiv \varphi[s] \\
&\mathit{Final}(\delta_1; \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s) \\
&\mathit{Final}(\delta_1 | \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \vee \mathit{Final}(\delta_2, s) \\
&\mathit{Final}(\delta^*, s) \equiv \texttt{True} \\
&\mathit{Final}(P_i, s) \equiv \mathit{Final}(body(P_i), s)
\end{aligned}$$

We use $\mathit{Trans}^*(\delta, s, \delta', s')$ to denote the transitive closure of *Trans*, i.e., there exists a sequence of one-step transitions taking the configuration (δ, s) into the configuration (δ', s'). As in (De Giacomo, Lespérance, and Levesque 2000), we define $Do(\delta, s, s') \doteq \exists \delta'. \mathit{Trans}^*(\delta, s, \delta', s') \wedge \mathit{Final}(\delta', s')$, where s' is the situation resulting from the full execution of δ from s (there can be many such s' due to the nondeterministic choices of the agent and responses of the environment).

Situation determinacy. Here, we want to use Golog as a specification language for finite traces, i.e., sequences of actions and fluent evaluations. For this reason, we first adapt the notion of *situation determined* programs , introduced by (De Giacomo, Lespérance, and Muise 2012), and then define *syntactically situation determined* (SSD) programs.

We say that a configuration (δ, s) is a *dead end* if it cannot proceed nor terminate:

$$DeadEnd(\delta, s) \doteq \neg Final(\delta, s) \wedge \neg \exists \delta', s'. Trans(\delta, s, \delta', s')$$

We say that (δ, s) is situation-determined (SD) iff

$$SitDet(\delta, s) \doteq \forall s', \delta', \delta''. \; Trans(\delta, s, \delta', s') \wedge Trans(\delta, s, \delta'', s') \supset \delta' = \delta'' \vee DeadEnd(\delta', s') \vee DeadEnd(\delta'', s')$$

Intuitively, situation determinacy requires that whenever two executions of the same action lead to the same situation but with different remaining programs, then one of them is a dead end, i.e., cannot proceed nor terminate. Thus, there can be only one remaining program that is not a dead end, and we can ignore all others if we are looking for complete executions. Given a NDBAT $\mathcal{D}$ and a model $M \models \mathcal{D}$, a program δ_0 is SD if every configuration reachable from (δ_0, S_0) is SD, i.e. $M \models Trans^*(\delta_0, S_0, \delta, s) \supset SitDet(\delta, s)$.

SD is crucial if we want to use Golog to express properties of traces because it allows us to determine a unique remaining part of the program at the various points of a trace.[2] However, this notion is semantical and depends on the model M. Instead, we want a notion that is easier to check. Hence, we introduce two relations T and F, which are a sort of syntactical counterpart of *Trans* and *Final*, respectively.

Given a program δ and an agent action a, $T(\delta, a)$ returns a set of pairs (φ, δ') of a condition φ holding in the current situation and the remaining program δ' after the execution of a when φ holds. T is inductively defined as follows:

$$\begin{aligned}
&T(a, a) = \{(Poss_{agt}(a), nil)\} \\
&T(b, a) = \{\}, \qquad \text{where } b \neq a \\
&T(\varphi?, a) = \{\} \\
&T(\delta_1; \delta_2, a) = \{(\varphi, \delta_1'; \delta_2) \mid (\varphi, \delta_1') \in T(\delta_1, a)\} \cup \\
&\qquad \{(F(\delta_1) \wedge \varphi, \delta_2') \mid (\varphi, \delta_2') \in T(\delta_2, a)\} \\
&T(\delta_1 | \delta_2, a) = T(\delta_1, a) \cup T(\delta_2, a) \\
&T(\delta^*, a) = \{(\varphi, \delta'; \delta^*) \mid (\varphi, \delta') \in T(\delta, a)\} \\
&T(P_i, a) = T(body(P_i), a)
\end{aligned}$$

$F(\delta)$ returns the condition under which δ can be considered *Final*, and is inductively defined as follows:

$$\begin{aligned}
&F(a) = False \\
&F(\varphi?) = \varphi \\
&F(\delta_1; \delta_2) = F(\delta_1) \wedge F(\delta_2) \\
&F(\delta_1 | \delta_2) = F(\delta_1) \vee F(\delta_2) \\
&F(\delta^*) = \texttt{True} \\
&F(P_i) \equiv F(body(P_i))
\end{aligned}$$

Theorem 1. *Let δ be a program and s a situation. Then, $M \models Final(\delta, s)$ iff $M \models F(\delta)[s]$.*

In terms of these, we define the *continuation condition of a program* $C(\delta)$ that, given a program δ, returns implied syntactic test conditions for the selection of δ itself, which ensure that it is not a dead end:

$$C(\delta) = F(\delta) \vee \bigvee_{a \in \mathcal{A}, (\varphi, \delta') \in T(\delta, a)} \varphi$$

We can show that:

Lemma 2. *For any model M of any NDBAT $\mathcal{D}$, if $M \models Final(\delta, s) \vee \exists \delta', s'. Trans(\delta, s, \delta', s')$ then $M \models C(\delta)[s]$.*

We say that a program δ is SSD if

1. For every action a, if $(\varphi_1, \delta_1) \in T(\delta, a)$ and $(\varphi_2, \delta_2) \in T(\delta, a)$ with $\delta_1 \neq \delta_2$, then either $\varphi_1 \wedge \varphi_2$ or $C(\delta_1) \wedge C(\delta_2)$ are propositionally unsatisfiable, considering atoms of the form $Poss_{agt}(a)$ as new distinct propositions.

[2] Note that our notion of SSD programs allows one to make the remaining programs depend on the environment reactions, e.g., $\delta = move; flat?; change\_tire | move; \neg flat?$. This cannot always be done with the notion of SD program in (De Giacomo, Lespérance, and Muise 2012), unless one extends the language with synchronized post-action tests.

2. For each P_i s.t. $body(P_i) = \delta_{i,1}; Q_{i,1} | \cdots | \delta_{i,m}; Q_{i,m}$, if $Q_{i,j}$ is a procedure call P_j then $F(\delta_{i,j})$ is unsatisfiable.

These conditions are easy to check and do not require access to the NDBAT. In the following, we restrict our attention to programs δ_0 such that δ_0 is SSD, and any remaining program δ that can be obtained by executing a sequence of transitions from δ_0 is also SSD (i.e, for all δ is in the syntactic closure of δ_0, Γ_{δ_0}, which we will define later, δ is SSD). If δ is SSD then it is also SD in the following strong sense:

Theorem 3. *If δ is SSD then for every model M of any NDBAT $\mathcal{D}$ we have that $M \models \forall s.SitDet(\delta, s)$.*

Examples. Having explicit tests controlling possible execution of remaining programs is a key aspect in SSD programs. Consider the program $\delta_1 = (a; b)|(a; c)$. We have that $T(\delta_1, a) = \{(Poss(a), (nil; b)), (Poss(a), (nil; c))\}$, while $C(nil; b) = \texttt{True}$ and $C(nil; c) = \texttt{True}$. Since $\texttt{True} \wedge \texttt{True}$ is satisfiable, the program is not SSD. Now, consider $\delta_2 = (a; e_1?; b)|(a; \neg e_1?; c)$. Then, $T(\delta_1, a) = \{(Poss(a), (nil; e_1?; b)), (Poss(a), (nil; \neg e_1?; c))\}$, while $C(nil; e_1?; b)) = e_1$ and $C(nil; e_2?; c) = \neg e_1$. Since $e_1 \wedge \neg e_1$ is unsatisfiable, it follows that δ_2 is SSD. Similarly, it is easy to check that $\delta_3 = (a|b)^*; a$ is not SSD, while $\delta_4 = ((p?; a)|b)^*; \neg p?; a$ is SSD.

Traces. For every model M of any NDBAT D, and any situation s, we define:

$$\begin{aligned} act(s) &= \begin{cases} d_{act} & \text{if } s = S_0 \\ a & \text{if } s = do(a(e), s') \text{ for some } s' \end{cases} \\ fl(s) &= \{Fl \in \mathcal{F} \mid M \models Fl[s]\} \end{aligned}$$

where d_{act} is a dummy agent action used for technical reasons, and $Fl \in fl(s)$ are the situation suppressed fluents that are true in the situation s in M. In this way, given a model M and a situation $s = do(a_\ell(e_\ell), do(\cdots do(a_1(e_1), S_0) \cdots))$, we can build the trace $trc(s)$ consisting of sequences of instants, formed by an action, and a propositional interpretation over situation suppressed fluents, namely:

$$(act(s_0), fl(s_0)), (act(s_1), fl(s_1)), \ldots, (act(s_\ell), fl(s_\ell))$$

where $s_0 = S_0, s_1 = do(a_1(e_1), S_0), \ldots, s_\ell = s$. We define the language $L(\delta_0)$ of a Golog program δ_0 (in M) as the set of traces corresponding to complete executions of δ_0 in M, i.e., $L(\delta_0) = \{trc(s) \mid M \models Do(\delta_0, S_0, s)\}$.

On the other hand, given a trace

$$\tau = (d_{act}, \mathcal{F}_0), (a_1, \mathcal{F}_1), \ldots, (a_\ell, \mathcal{F}_\ell)$$

we can check if τ is a trace of M (i.e., $M \models \tau$) by building situations $s_0 = S_0$, and $s_i = do(a_i(e_i), s_{i-1})$ for some reaction e_i and check $\mathcal{F}_i = fl(s_i)$, for $i = 0, \ldots, \ell$, and $M \models Poss(a_i(e_i), s_{i-1})$, for $i = 1, \ldots, \ell$. We denote *situation* s_ℓ as $sit(\tau)$. Note that $\tau = trc(sit(\tau))$.

5 Golog Expressiveness on Traces

We next show that Golog programs as defined above have exactly the same expressive power as regular languages, or MSO, over finite traces. In doing this, we assume that the Golog programs are defined over a NDBAT $\mathcal{D}$ and model $M \models \mathcal{D}$ that are completely unconstrained, i.e., such that $\{trc(s) \mid M \models executable(s)\} = \Sigma^*$, with $\Sigma = \mathcal{A} \times 2^{\mathcal{F}}$.[3]

We show that every SSD Golog program δ_0 can be translated into a (two-way) regular expression through the following inductively defined function (for simplicity, we abuse notation by allowing conjunction of the action components and the fluents component of the atomic symbols):

$$\begin{aligned} &re(a) = a \\ &re(\varphi?) = true^-; \varphi \\ &re(\delta_1; \delta_2) = re(\delta_1); re(\delta_2) \\ &re(\delta_1|\delta_2) = re(\delta_1) + re(\delta_2) \\ &re(\delta^*) = re(\delta)^* \\ &re(P_i) = P_i \end{aligned}$$

where $true^-$ matches any transition backward. Then, from procedure definitions, we obtain a regular grammar $RG(\delta_0)$:

$$\begin{aligned} &P_0 := re(\delta_0) \\ &P_i := re(body(P_i)), \quad i = 1, \ldots n \end{aligned}$$

Theorem 4. *Let δ_0 and $P_i : body(P_i)$, for $i = 1, \ldots, n$ be a Golog program with procedures, then $L(\delta_0) = L(RG(\delta_0))$.*

On the other hand, given a deterministic finite automaton (DFA) $A_d = (\mathcal{A} \times 2^{\mathcal{F}}, Q, q_0, \varrho_d, F_d)$ we can obtain the following SSD program $\delta_{A_d} \doteq q_0$ with procedure definitions

$$q : a; \varphi?; \varrho_d(q, (a, \varphi))$$

for every $q \in Q \setminus F_d$ and every $(a, \varphi) \in A \times 2^{\mathcal{F}}$, and

$$q : nil | a; \varphi?; \varrho_d(q, (a, \varphi))$$

for every $q \in F_d$ and every $(a, \varphi) \in A \times 2^{\mathcal{F}}$.

Theorem 5. *Let A_d be a DFA and δ_{A_d} (including procedure definitions) be the SSD program defined as above, then $L(\delta_{A_d}) = L(A_d)$.*

Theorem 6. *SSD Golog programs have the same expressive power as regular languages or MSO on finite traces.*

We observe that LDL$_f$ also has the same expressive power as regular languages or MSO on finite traces (De Giacomo and Vardi 2013; De Giacomo and Vardi 2015; Brafman, De Giacomo, and Patrizi 2018). It follows that it has the same expressive power as Golog on finite traces. More specifically, note that we can translate any Golog program δ_0 without procedure calls into an LDL$_f$ formula of the form $\langle rt(\delta_o) \rangle end$ where end denotes the end of the the trace (Brafman, De Giacomo, and Patrizi 2018), and where $rt(\delta_o)$ is defined inductively as follows:

$$\begin{aligned} &rt(a) = a \\ &rt(\varphi?) = \varphi? \\ &rt(\delta_1; \delta_2) = rt(\delta_1); rt(\delta_2) \\ &rt(\delta_1|\delta_1) = rt(\delta_1) + rt(\delta_2) \\ &rt(\delta^*) = rt(\delta)^* \end{aligned}$$

We will exploit this translation in the experimental section.

[3] Alternatively, we can translate the theory and model as well as the program, along the lines of (De Giacomo and Vardi 2015).

6 Program Graphs

A program graph is a structure that captures the control flow of a given program δ_0 at the syntactic level, decoupled from domain dynamics. This idea is introduced by (Claßen 2013; Claßen and Lakemeyer 2008). Here, we follow (De Giacomo, Lespérance, and Mancanelli 2026), though we specialize the definition of the program graph to the specific framework we consider.

We start by defining the syntactic closure of a program.

Syntactic closure. Given a program δ_0 with procedure definitions $P_i : body(P_i)$, for $i = 1, \ldots, n$, we define its syntactic closure Γ_{δ_0} by induction as:

$\delta_0, nil \in \Gamma_{\delta_0}$
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 $\Gamma_{\delta_2} \subseteq \Gamma_{\delta_0}$
if $\delta_1 \mid \delta_2 \in \Gamma_{\delta_0}$, then $\Gamma_{\delta_1}, \Gamma_{\delta_2} \subseteq \Gamma_{\delta_0}$
if $\delta^* \in \Gamma_{\delta_0}$, then $\delta; \delta^* \in \Gamma_{\delta_0}$
if $P_i \in \Gamma_{\delta_0}$, then $body(P_i) \in \Gamma_{\delta_0}$

It is easy to show the following results:

Theorem 7. *The syntactic closure Γ_{δ_0} of a program δ_0 is linear in the size of δ_0 and $P_i : body(P_i)$, for $i = 1, \ldots, n$.*

Lemma 8. *Let δ be a program in Γ_{δ_0}, then for every action a, any program δ' occurring in $T(\delta, a)$ is such that $\delta' \in \Gamma_{\delta_0}$.*

We can also show that every program that δ_0 can evolve into according to $Trans$, is in its syntactic closure Γ_{δ_0}.

Lemma 9. *Let δ be a program in Γ_{δ_0}, then for every action a, if δ' is such that $M \models Trans(\delta, s, \delta', s')$, then $\delta' \in \Gamma_{\delta_0}$.*

Lemma 10. *Let δ_0 be a Golog program, then for every model M of any NDBAT D over which δ_0 is executed we have that if $M \models Trans^*(\delta_0, s_0, \delta, s)$, then $\delta \in \Gamma_{\delta_0}$.*

Constructing the Program Graph. Formally, the program graph of a Golog program is defined as follows:

Definition 11. *Let δ_0 be a Golog program with procedure definitions. The corresponding program graph is*

$$\mathcal{G} = \langle \Phi \times \mathcal{A} \times \Psi, Q, q_0, \tau, \mathcal{L} \rangle$$

where

- $\Phi \times \mathcal{A} \times \Psi$ *is the alphabet, with Φ and Ψ sets of formulas over tests and $Poss$, and $\mathcal{A}$ the set of actions.*
- $Q = \Gamma_{\delta_0}$ *is the set of nodes, corresponding to the programs in the syntactic closure of δ_0.*
- $q_0 = \delta_0$ *is the initial program*
- $\tau(q, \varphi, a, \psi, q')$ *iff* $(\varphi, q') \in T(q, a)$ *and* $\psi = C(q')$.
- $\mathcal{L}(q) = F(q)$ *is a label assigned to q.*

Note that for SSD Golog program, for all q' s.t. $(\varphi, q') \in T(q, a)$, we have that $\psi = C(q')$ are disjoint, and hence we have only one tuple such that $\tau(q, \varphi, a, \psi, q')$ for each q'.

We now summarize key properties of the program graph. Unless otherwise specified, we will write $\mathcal{G}_{\delta_0}$, or simply $\mathcal{G}$, 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, by definition, the nodes being the programs in the syntactic closure, we have:

Theorem 12. *The number of nodes and the number of edges in $\mathcal{G}$ are linear in the size of δ_0.*

Next we relate program graphs with Golog 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 13. *For all subprograms $\delta, \delta' \in \Gamma_{\delta_0}$, agent action a, and situation s:*

1. *If $\tau(\delta, \varphi, a, \psi, \delta') \in \mathcal{G}$, then for every situation s such that $M \models \varphi[s]$, there exists a reaction e s.t. $M \models Trans(\delta, s, \delta', do(a(e), s))$ and $M \models C(\delta')[s']$.*
2. *If $M \models Trans(\delta, s, \delta', do(a(e), s)) \wedge \neg DeadEnd(\delta', do(a(e), s))$, then there exists φ and $\psi = C(\delta')[s']$ s.t. $\tau(\delta, \varphi, a, \psi, \delta') \in \mathcal{G}$ and $M \models \varphi[s]$ and $M \models C(\delta')[s']$.*

A full execution trace to a final configuration exists in the model iff there is a corresponding path in the program graph.

Theorem 14. *Let (δ_0, s_0) be an initial configuration. Then, for every situation s and program in Γ_{δ_0} we have that $M \models Trans^*(\delta_0, s_0, \delta, s) \wedge Final(\delta, s)$ iff there is a sequence of transitions $\tau(q_0, \varphi_1, a_1, \psi_1, q_1), \ldots, \tau(q_{\ell-1}, \varphi_\ell, a_\ell, \psi_\ell, q_\ell)$ in $\mathcal{G}$ and reactions $e_1, ..., e_\ell$ such that $q_0 = \delta_0$, $q_\ell = \delta$ and $s = s_\ell$, and for each $i = 1, ..., \ell$:* (i) $s_i = do(a_i(e_i), s_{i-1})$; (ii) $M \models \varphi_i[s_{i-1}]$; (iii) $M \models \psi_i[s_i]$; (iv) $M \models F(q_\ell)[s_\ell]$.

7 Synthesis and Strategic Reasoning

Agent Control and Strategic Reasoning. To represent the ability of the agent to execute an agent program in an ND domain, (De Giacomo et al. 2021) introduces $AgtCanForceBy(\delta, f, s)$ as an adversarial version of Do in the presence of environment reactions. This predicate states that situation strategy f, a function from situations to agent actions (including the special action *stop*), executes Golog agent program δ in situation s considering its nondeterminism angelic, as in the standard Do, but also considering the nondeterminism of environment reactions devilish/adversarial.

$$AgtCanForceBy(\delta, f, s) \doteq \forall P.[\ldots \supset P(\delta, s)]$$
where ... stands for
$[f(s) = stop \wedge Final(\delta, s) \supset P(\delta, s)] \wedge$
$[\exists a.(f(s) = a \neq stop \wedge$
$\exists e.\exists \delta'.Trans(\delta, s, \delta', do(a(e), s)) \wedge$
$\forall e.(\exists \delta'.Trans(\delta, s, \delta', do(a(e), s))) \supset$
$\exists \delta'.Trans(\delta, s, \delta', do(a(e), s)) \wedge$
$P(\delta', do(a(e), s)) \supset P(\delta, s)]$

We say that predicate $AgtCanForce(\delta, s)$ holds iff there exists a strategy f s.t. $AgtCanForceBy(\delta, f, s)$ holds. In the following, we refer to f as a situation strategy, in contrast with other kinds of strategies that we will define.

Induced and Abstract TSs. Before constructing the game arena over which we perform strategy synthesis, we introduce some preliminary notions.

Consider a model M of $\mathcal{D}$ with object domain Δ and situation domain $\mathcal{S}$. The TS induced by M is the labeled TS $T_M = \langle \Sigma_M, \mathcal{S}_M, S_0, \rightarrow_T, L_M \rangle$ such that:

- $\Sigma_M = \{a(e) \mid a \in \mathcal{A}, e \in React_a\}$ is the alphabet
- $\mathcal{S}_M$ is the set of executable situations in M
- S_0 is the initial situation S_0^M
- $\rightarrow_T \subseteq \mathcal{S}_M \times \Sigma_M \times \mathcal{S}_M$ is the transition relation s.t. $s \rightarrow_T^{a(e)} s'$ iff $M \models Poss(a(e), s)$ and $s' = do(a(e), s)$
- $L_M : \mathcal{S}_M \rightarrow 2^{\mathcal{F}}$ is a labeling function assigning to each situation the set of fluents true in it, i.e., $L_M(s) = \{Fl \in \mathcal{F} \mid M \models Fl[s]\}$

Intuitively, every node of T_M corresponds to a situation, and each transition corresponds to the execution of a system action. Note that the object domain, the set of action functions and the set of environment reactions are finite, but since the situation domain is infinite, this transition system is generally infinite. Note that multiple situations can share the same label; we say that two situations are isomorphic, written $s_1 \sim_M s_2$, if they agree on the truth value of all fluents, i.e., $\forall Fl \in \mathcal{F}$, $M \models Fl[s_1]$ iff $M \models Fl[s_2]$. The relation $\sim_M$ is an equivalence relation, and each equivalence class $[s]_M = \{s' \mid s \sim_M s'\}$ corresponds to a unique state in the propositional domain.

It can be shown that finite paths in T_M are in one-to-one correspondence with executable sequences of system actions $a(e)$ in the model M:

Lemma 15. *Let M be a model of $\mathcal{D}$ and T_M be the TS induced by M. For every initial situation $s_0 \in S_M$ and every finite sequence of agent actions $a_0, \ldots, a_{n-1} \in \mathcal{A}$, the following are equivalent:*

- *There exist environment reactions $e_0, \ldots, e_{n-1}$ and situations $s_1, \ldots, s_n \in \mathcal{S}_M$ s.t. for all $i = 0, \ldots, n-1$, $M \models Poss(a_i(e_i), s_i)$ and $s_{i+1} = do(a_i(e_i), s_i)$.*
- *There exist environment reactions $e_0, \ldots, e_{n-1}$ and situations $s_1, \ldots, s_n \in S_M$ s.t. for all $i = 0, \ldots, n-1$ we have $s_i \rightarrow_T^{a_i(e_i)} s_{i+1}$ in T_M.*

Now, we define an abstract finite structure A_M induced by T_M. Formally, A_M is a tuple $\langle \Sigma_M, P, p_0, \rightarrow_A \rangle$ where:

- $\Sigma_M = \{a(e) \mid a \in \mathcal{A}, e \in React_a\}$ is the alphabet
- $P = 2^{\mathcal{F}}$ is the set of states
- $p_0 = L_M(S_0)$ is the valuation induced by S_0
- $\rightarrow_A \subseteq P \times \Sigma_M \times P$ is the transition relation where $p \rightarrow_A^{a(e)} p'$ iff $\exists s, s' \in \mathcal{S}_M$ s.t. $p = L_M(s)$, $s \rightarrow_T^{a(e)} s'$, and $p' = L_M(s')$

Intuitively, A_M collapses all situations of T_M with the same fluent valuation into a single state. There is a transition $p \rightarrow_A^{a(e)} p'$ whenever some situation s with valuation $p = L_M(s)$ performs action a and reaches a situation s' with valuation $p' = L_M(s')$. Since the set of fluents is finite, A_M is a finite transition system.

We can extend the notation $\sim_M$ to relate situations and states, writing $s \sim_M p$ iff $\forall Fl \in \mathcal{F}$, $M \models Fl[s]$ iff $Fl \in p$. Note that if $p = L_M(s)$, then $s \sim_M p$. To relate T_M and A_M, it can be shown that there exists a bisimulation relation $R \subseteq \mathcal{S}_M \times P$ between T_M and A_M such that $R(s, p)$ hold iff $s \sim_M p$.

Theorem 16. *Let $R(s, p)$ hold iff $s \sim_M p$. Then R is a bisimulation between T_M and A_M.*

Lemma 17. *Let $R(s, p)$ hold iff $s \sim_M p$, and let φ be a situation-suppressed propositional formula over $\mathcal{F}$. If $R(s, p)$, then $M \models \varphi[s]$ iff $p \models \varphi$.*

Game Arena and Synthesis. Now we are ready to write a game-theoretic characterization of reactive synthesis. Intuitively, we leverage on the cross product between the program graph $\mathcal{G}_{\delta_0}$ and the TSs to synchronize the procedural control state of the program δ_0 with the evolving domain state induced by the action theory, making explicit the interaction between control flow and nondeterministic dynamics.

Definition 18. *The cross product between a program graph $\mathcal{G}$ and T_M is defined by the following tuple:*

$$T_{M,\mathcal{G}} = \langle \Sigma_M \times Q, Q \times S, (\delta_0, s_0), Tr_T, Fin_T \rangle$$

where

- *Σ_M is the alphabet*
- *$Q \times S$ is a set of states*
- *(δ_0, s_0) is the initial state*
- *$Tr_T((\delta, s), a(e)) = (\delta', s')$ is a transition such that $M \models Trans(\delta, s, \delta', s')$ and $s' = do(a(e), s)$*
- *$Fin_T = \{(\delta, s) \mid M \models F(\delta)[s]\}$ is the set of final states*

Note that $T_{M,\mathcal{G}}$ is infinite wrt the situation domain.

A run in $T_{M,\mathcal{G}}$ is an infinite sequence $(\delta_1, s_1), (\delta_2, s_2), \ldots$ such that for every i there exists some action $a_i \in \mathcal{A}$, environment reaction $e_i \in React_{a_i}$ and next configuration (δ_{i+1}, s_{i+1}) such that $Tr_T((\delta_i, s_i), a_i(e_i)) = (\delta_{i+1}, s_{i+1})$. A strategy for a program in $T_{M,\mathcal{G}}$ is a function $\hat{f}_T : (Q \times \mathcal{S})^+ \rightarrow \mathcal{A}$, defined on nonempty finite prefixes of runs of $T_{M,\mathcal{G}}$, that assigns an action to each such prefix.

Given a strategy $\hat{f}_T$, a run induced by $\hat{f}_T$ is a run $(\delta_1, s_1), (\delta_2, s_2), \ldots$ such that for every i:

- the agent strategy selects $a_i = \hat{f}_T((\delta_1, s_1), \ldots, (\delta_i, s_i))$
- there exist e_i s.t. $Tr_T((\delta_i, s_i), a_i(e_i)) = (\delta_{i+1}, s_{i+1})$

The set of runs induced by $\hat{f}_T$ in $T_{M,\mathcal{G}}$ is denoted by $Runs(\hat{f}_T, T_{M,\mathcal{G}})$. A strategy $\hat{f}_T$ is winning from (δ, s) if for every run in $Runs(\hat{f}_T, T_{M,\mathcal{G}})$ starting from (δ, s) there exists some index i such that $(\delta_i, s_i) \in Fin_T$.

We say that a situation strategy f for M and a strategy $\hat{f}_T$ on $T_{M,\mathcal{G}}$ are equivalent if for every finite run $(\delta_1, s_1), \ldots, (\delta_k, s_k)$ in $T_{M,\mathcal{G}}$ we have $\hat{f}_T((\delta_1, s_1), \ldots, (\delta_k, s_k)) = f(s_k)$. Every situation strategy f corresponds to a unique equivalent strategy $\hat{f}_T$, and vice versa.

Theorem 19. *Let f and $\hat{f}_T$ be equivalent strategies. Then, for any program $\delta \in Q$, $M \models AgtCanForceBy(\delta, f, s)$ iff $\hat{f}_T$ is winning in $T_{M,\mathcal{G}}$ from (δ, s).*

This result establishes a precise correspondence between strategic reasoning in the situation calculus and winning strategies in the induced $T_{M,\mathcal{G}}$. In particular, it shows how adversarial execution of a Golog program can be computed at the level of TSs.

To obtain a finite game suitable for algorithmic synthesis, we abstract away from concrete situations and instead reason over propositional valuations of fluents. This yields a finite game arena that preserves all strategic choices relevant to program execution, enabling at the same time the use of well-known synthesis tools. Here, we restrict to SSD programs so that strategy synthesis amounts to choosing only the next action; as a consequence, the product construction does not need to read the next program on a transition, and the alphabet becomes simply Σ_M.

Definition 20. *The cross product between a program graph $\mathcal{G}$ and A_M is defined by the following tuple:*

$$A_{M,\mathcal{G}} = \langle \Sigma_M, Q \times P, (\delta_0, p_0), Tr_A, Fin_A \rangle$$

where

- Σ_M *is the alphabet*
- $Q \times P$ *is a set of game states*
- (δ_0, p_0) *is the initial state*
- $Tr_A((\delta, p), a(e)) = (\delta', p')$ *is the transition function where* (i) $\exists \varphi . \tau(\delta, \varphi, a, \delta')$, (ii) $p \models \varphi$, (iii) $p \rightarrow_A^{a(e)} p'$, *and* (iv) $p' \models C(\delta')$
- $Fin_A = \{(\delta, p) \mid p \models F(\delta)\}$ *is the set of final states*

This can be viewed as a game arena, where the agent controls the action $a \in \mathcal{A}$ and where the environment controls the environment reaction $e \in React_a$. A play in the arena is an infinite sequence $(\delta_1, p_1), ...$ such that for every i there exists some action $a_i \in \mathcal{A}$, environment reaction $e_i \in React_{a_i}$ and next configuration (δ_{i+1}, p_{i+1}) such that $Tr((\delta_i, p_i), (a_i(e_i), \delta_{i+1})) = (\delta_{i+1}, p_{i+1})$. A strategy for a program in $A_{M,\mathcal{G}}$ is a function $\hat{f}_A : (Q \times P)^+ \rightarrow \mathcal{A}$, defined on nonempty finite sequences of game states of $A_{M,\mathcal{G}}$, that assigns an action to each such sequence. Given a strategy $\hat{f}_A$, a play induced by $\hat{f}_A$ is a play $(\delta_1, p_1), (\delta_2, p_2), ...$ such that for every i:

- the agent strategy selects the action a_i such that $a_i = \hat{f}_A((\delta, p), ..., (\delta_i, p_i))$
- there exist e_i s.t. $Tr_A((\delta_i, p_i), a_i(e_i)) = (\delta_{i+1}, p_{i+1})$

The set of plays induced by $\hat{f}_A$ in $A_{M,\mathcal{G}}$ is denoted by $Plays(\hat{f}_A, A_{M,\mathcal{G}})$. A strategy $\hat{f}_A$ is winning from (δ, p) if for every play in $Plays(\hat{f}_A, A_{M,\mathcal{G}})$ starting from (δ, p) there exists some index i such that $(\delta_i, p_i) \in Fin_A$.

We can extend the bisimulation relation R to configurations. Since we are restricting to SSD programs, the agent does not choose the successor program state but only the action. Hence, the $T_{M,\mathcal{G}}$ alphabet simplifies to Σ_M, and configurations that are dead ends are filtered out. We introduce a new relation $R^\times \subseteq (Q \times \mathcal{S}_M) \times (Q \times P)$ such that $R^\times((\delta, s), (\delta', p))$ holds iff $\delta = \delta'$, $R(s, p)$ and (δ, s) is not a dead end. We can show that $R^\times$ is a bisimulation between $T_{M,\mathcal{G}}$ and $A_{M,\mathcal{G}}$:

Theorem 21. *Let $R^\times$ defined as above. Then, $R^\times$ is a bisimulation between $T_{M,\mathcal{G}}$ and $A_{M,\mathcal{G}}$.*

The bisimulation result guarantees that the abstract game arena is sound and complete with respect to the concrete execution semantics: no winning strategy is lost, and no spurious winning strategy is introduced by abstraction. If we have a bisimulation relation between $T_{M,\mathcal{G}}$ and $A_{M,\mathcal{G}}$ s.t. $R^\times((\delta_i, s_i), (\delta_i, p_i))$, it is easy to prove that one can always find a run in $T_{M,\mathcal{G}}$ and a play in $A_{M,\mathcal{G}}$ so that the bisimulation relation is preserved along all pairs in the run/play (this follows by repeated applications of the forth/back clauses of the bisimulation). Formally, we can state:

Theorem 22. *Let $R^\times$ be a bisimulation relation between $T_{M,\mathcal{G}}$ and $A_{M,\mathcal{G}}$ such that $R^\times((\delta_i, s_i), (\delta_i, p_i))$. For every run $(\delta_i, s_i), (\delta_{i+1}, s_{i+1}), ...$ in $T_{M,\mathcal{G}}$ such that $Tr_T((\delta_i, s_i), a_i(e_i)) = (\delta_{i+1}, s_{i+1})$, there is a play $(\delta_i, p_i), (\delta_{i+1}, p_{i+1}), ...$ in $A_{M,\mathcal{G}}$ s.t. $R^\times((\delta_j, s_j), (\delta_j, p_j))$ for all $j \geq i$, and vice versa.*

We say that $\hat{f}_T$ and $\hat{f}_A$ are $R^\times$*-equivalent* if for every finite run prefix $(\delta_i, s_i), \ldots, (\delta_k, s_k)$ in $T_{M,\mathcal{G}}$ and for every finite play prefix $(\delta_i, p_i), \ldots, (\delta_k, p_k)$ in $A_{M,\mathcal{G}}$ we have $R^\times((\delta_i, s_i), (\delta_i, p_i))$ and $\hat{f}_T((\delta_i, s_i), \ldots, (\delta_k, s_k)) = \hat{f}_A((\delta_i, p_i), \ldots, (\delta_k, p_k))$. With this notion in place, we are ready to formulate the following results:

Theorem 23. *Let $R^\times((\delta, s), (\delta, p))$, and let $\hat{f}_T$ and $\hat{f}_A$ be $R^\times$-equivalent strategies. Then $\hat{f}_T$ is winning in $T_{M,\mathcal{G}}$ from (δ, s) iff $\hat{f}_A$ is winning in $A_{M,\mathcal{G}}$ from (δ, p).*

Corollary 24. *Let $R^\times((\delta, s), (\delta, p))$. Let f be a situation strategy, $\hat{f}_T$ a strategy on $T_{M,\mathcal{G}}$, and $\hat{f}_A$ a game strategy on $A_{M,\mathcal{G}}$ such that $\hat{f}_T$ is equivalent to f, and $\hat{f}_A$ is $R^\times$-equivalent to $\hat{f}_T$. Then, $M \models AgtCanForce(\delta, f, s)$ iff $\hat{f}_A$ is winning in $A_{M,\mathcal{G}}$ from (δ, p).*

These results show that Golog synthesis under nondeterminism can be reduced to solving a 2-player reachability game.

Complexity Results for Golog Programs. We now analyze the computational complexity of synthesis for Golog programs. Despite Golog programs may have complex constructs, the size of the game arena grows only linearly with the size of the program. This contrasts with declarative temporal logics like LTL$_f$/LDL$_f$, where determinization typically incurs an exponential blow-up.

Lemma 25. *Let δ_0 be a Golog program and $\mathcal{A}_{M,\mathcal{G}}$ a game arena. Then, the number of states and the number of transitions in the game arena are linear in the size of δ_0.*

Once the game arena is constructed, deciding the existence of a winning strategy and synthesizing it can be done in linear time, making synthesis practical even for nontrivial specifications. This is particularly relevant if compared with LDL$_f$ synthesis. In general, realizability in LDL$_f$ can be solved in 2EXPTIME, and synthesis can be done in doubly exponential time (De Giacomo and Vardi 2015).

Theorem 26. *A winning strategy in $\mathcal{A}_{M,\delta}$, if it exists, can be computed in linear time with respect to the size of δ.*

Corollary 27. *Given a Golog program δ, synthesis can be done in time linear in the size of δ.*

Note that obviously, we remain EXPTIME with respect to the size of the NDBAT, as in the case of LTL$_f$/LDL$_f$ specifications (De Giacomo and Rubin 2018).

8 Experiments

Implementation. We implemented our synthesis algorithm for Golog programs (which builds the game arena as in Def. 20) in a tool called SYFT4GOLOG[4] based on the symbolic synthesis framework in (Zhu et al. 2017). SYFT4GOLOG relies on LYDIASYFT (Zhu and Favorito 2025) – one of the best performing publicly available synthesizers for LTL$_f$/LDL$_f$ specifications – to represent symbolically program graphs and DFAs as well as to solve symbolic DFA games. SYFT4GOLOG represents symbolically state space and transition function of DFAs using Binary Decision Diagrams (Bryant 1992), with CUDD-3.0.0 (Somenzi 1998) as the underlying BDD library. Specifically, SYFT4GOLOG represents Golog program graphs as Shared Multi-Terminal Binary Decision Diagrams (ShMTBDDs), a data structure commonly used to compactly represent DFAs, see, e.g., (Henriksen et al. 1995; Duret-Lutz et al. 2025). ShMTBDDs provide a semi-symbolic representation of program graphs: states are represented explicitly; transitions are represented symbolically. Following the approach outlined in Sec. 7, the ShMTBDD of a program graph is converted into a fully symbolic representation and combined with the FOND domain to construct a symbolic game arena. The FOND domain is specified in PDDL (Haslum et al. 2019); the symbolic DFA of the domain is obtained using the PDDL-to-DFA transformation shown in (De Giacomo, Di Stasio, and Parretti 2025). As usual, predicates and action preconditions and effects are specified in a `domain.pddl` file, whereas the initial state and objects are specified in a `problem_n.pddl` file, parametrized by the input size n.

As a baseline, we also implemented an alternative synthesis technique that transforms Golog programs into LDL$_f$ specifications using the rt function of Section 5 and reduces to LDL$_f$ synthesis in FOND domains (De Giacomo and Rubin 2018). This latter implementation also uses the PDDL-to-DFA transformation in (De Giacomo, Di Stasio, and Parretti 2025) to obtain symbolic DFAs of FOND domains and LYDIASYFT to represent and solve symbolic DFA games. In particular, DFAs of LDL$_f$ specifications are obtained using LYDIA (De Giacomo and Favorito 2021). This shared toolchain enables a fair comparison between Golog and LDL$_f$, as discussed below in this section.

Experimental Setup. Our experiments are designed to identify where the computational effort arises and when procedural structure helps. First, as stated previously, we compare against an LDL$_f$-based baseline to assess whether leveraging on procedural structure yields practical benefits in synthesis time and solved instances. Second, we study the impact of program specificity/guidance by generating three templates of increasing specificity for each benchmark; this helps in isolating the effect of procedural guidance from the underlying domain dynamics. These templates range from a very permissive program that can choose any action at each step, to progressively more structured programs that constrain the order of actions. We can also evaluate scalability by comparing against the FO-based approach proposed by (Hofmann and Claßen 2025), assessing whether our propositional implementation scales beyond the instance sizes previously reported while remaining competitive with LDL$_f$.

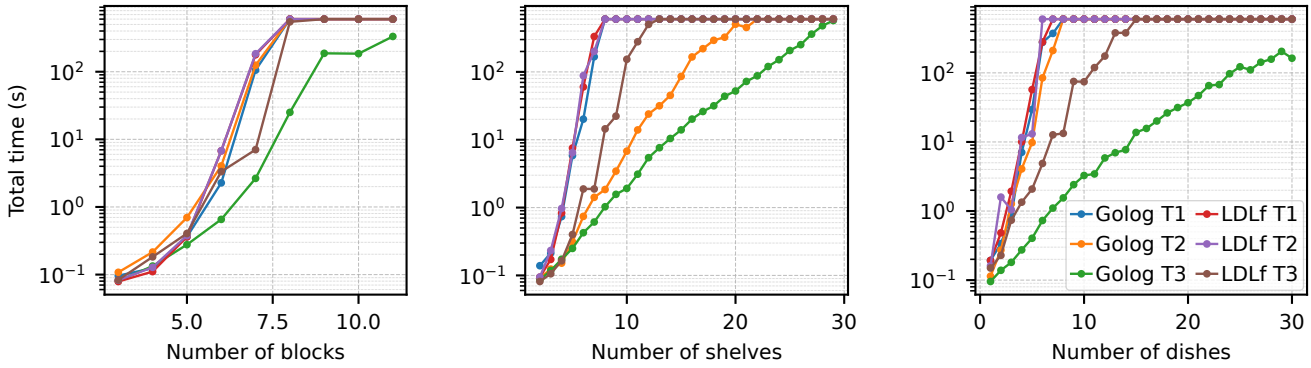
All experiments were conducted on an Intel(R) Core(TM) i7-12700H (2.30 GHz) with 16 GB of RAM, running Ubuntu with WSL2 (WSL version 2.1.5.0, Linux kernel 5.15.146.1-2) under Windows 11. We enforce a timeout of 600 seconds per instance. For each instance and program template, we executed three independent runs and report the mean end-to-end runtime.

Benchmark Domains. We evaluate our approach on three benchmark domains, namely *Blocksworld*, *Warehouse Robot*, and *Dishwasher Robot*. These domains allow us to assess both the effect of procedural guidance and the scalability of the synthesis procedure. Blocksworld is a classical benchmark with well-understood combinatorial structure and is widely used as a baseline; despite simple action dynamics, it is known to be challenging for strategy synthesis. The Warehouse Robot and Dishwasher Robot domains are adapted from (Hofmann and Claßen 2025). We modify them to incorporate the propositional setting and agent-environment interaction model of our framework, while preserving the domain structure. Overall, these domains feature structured tasks with repeated subgoals and are well suited to evaluate whether increasingly fine-grained procedural descriptions of the agent behavior can reduce synthesis effort.

Blocksworld. The Blocksworld domain consists of n blocks initially stacked in a single tower. The binary predicate on encodes the initial configuration, with $on(b_i, b_{i+1})$ for all $i \leq n-1$ and $on(b_n, table)$, where $table$ is a constant. The goal is to place every block on the table, i.e., $\bigwedge_{i=1}^{n} on(b_i, table)$. The agent can execute $unstack$ to pick up a block, and put_down to place a held block onto the table. $unstack$ is subject to a nondeterministic environment reaction: it may fail, in which case the block remains in its current position and the agent does not hold it. To prevent the environment from blocking progress indefinitely, each block can cause at most one such failure. We generate instances by increasing the number of blocks in the tower.

Warehouse Robot. The Warehouse Robot domain models a robot operating in a warehouse with multiple shelves. Initially, each shelf s_i contains a box b_i, while the robot starts at a designated shelf s_0 with an empty hand. The objective is to transport all boxes to s_0, i.e., to satisfy $\bigwedge_i box\_at(b_i, s_0)$. The robot can $move$ between shelves, $pickup$ a box located at its current shelf, optionally $wrap$ it, and $drop$ it at a target shelf. Dropping a box is nondeterministic: when the robot drops a box without wrapping, the environment may cause the box to break; wrapping removes this risk and guarantees a successful delivery.

[4]https://github.com/matteomancanelli/syft4golog



(a) Blocksworld. (b) Warehouse Robot. (c) Dishwasher Robot.

Figure 1: End-to-end synthesis time (log scale) for SYFT4GOLOG and the LDL$_f$ baseline across the three domains, for program types 1–3.

Dishwasher Robot. The Dishwasher Robot domain models a robot that must collect dishes from multiple rooms and deliver them to a designated kitchen room r_0 where it washes them if they are dirty. Initially, dishes are uniformly distributed across rooms. Each dish may be clean or dirty. The goal requires that all dishes are delivered to r_0 and are not dirty. The robot can $move$ between rooms, $load$ a dish at its current location, and $unload$ it in the kitchen. The $unload$ action is nondeterministic: it may succeed if the dish is not dirty or fail if it is dirty. If the dish is dirty, the robot must first $wash$ it, and then perform $unloadClean$, which always succeeds. We generate instances by varying independently the number of rooms and dishes, allowing us to study scalability with respect to both parameters.

Program Patterns. As already mentioned, we generate three program templates of increasing specificity to assess the impact of procedural guidance on synthesis.

Type 1 (Low Guidance). The first template is maximally permissive: at each step, the program may execute any available action. Formally, it consists of a nondeterministic choice over all actions, iterated with Kleene star, followed by a test of the goal condition, i.e. it has the form

$$(pick\_an\_action)^*; goal\_condition?$$

This provides minimal procedural guidance and is used to evaluate synthesis from largely unconstrained behavior.

Type 2 (Medium Guidance). The second template introduces a reusable subtask pattern. The program repeatedly selects one subtask from a fixed collection and executes it, before checking the goal. Its abstract form is

$$(subtask_1 \mid subtask_2 \mid \cdots \mid subtask_n)^*; goal\_condition?$$

Each subtask captures the handling of a single object or local objective (e.g., delivering one box, processing one dish, or unstacking one block). This template constrains the space of possible behaviors compared to Type 1, while still allowing flexibility in the order and repetition of subtasks.

Type 3 (High Guidance). The third template provides the strongest procedural guidance. Instead of nondeterministically selecting subtasks, it specifies a fixed sequential order in which they must be executed. Its abstract form is

$$(subtask_1; subtask_2; \ldots; subtask_n); goal\_condition?$$

So, the task is decomposed into a fixed sequence of subtasks, significantly restricting the space of admissible executions.

Results. Figure 1 reports the end-to-end synthesis time (log scale) for SYFT4GOLOG and the LDL$_f$ baseline, across the three program templates introduced above.

In Blocksworld (Figure 1a), programs of Type 1 and Type 2 exhibit similar performance. In both cases, Golog and LDL$_f$ specifications reach the timeout threshold when the number of blocks is around eight. This behavior is consistent with prior observations that Blocksworld is a challenging benchmark for strategy synthesis (De Giacomo, Di Stasio, and Parretti 2025). In contrast, Type 3 programs scale to larger instances: SYFT4GOLOG handles up to 11 blocks within the 600-second limit. This confirms that explicitly encoding a sequential decomposition of subtasks significantly improves scalability in this domain.

Consider now the Warehouse Robot benchmark (Figure 1b), which clearly highlights both the benefit of Golog specifications and the impact of procedural structure. For Type 1 programs, Golog and LDL$_f$ show comparable performance, solving instances up to seven shelves (and boxes). However, when moving to Type 2 and Type 3 programs, scalability improves substantially. In particular, Golog specifications consistently scale better than their LDL$_f$ counterparts. Notably, Type 2 programs with Golog specifications outperform Type 3 programs encoded in LDL$_f$, solving instances up to 21 shelves. Moreover, Type 3 Golog programs do not reach the timeout even at 29 shelves.

Results for the Dishwasher Robot benchmark (Figure 1c) confirm the trends observed in the previous domains. Type 1 programs exhibit similar behaviour for Golog and LDL$_f$, reflecting the limited procedural guidance provided by this template. In contrast, Type 3 programs clearly highlight the advantage of Golog specifications, which scale significantly better as the problem size increases. This benchmark also allows us to evaluate scalability in the presence of multiple

	Type 1		Type 2		Type 3	
n	Golog	LDL_f	Golog	LDL_f	Golog	LDL_f
2	3	4	11	13	12	13
3	3	4	15	18	17	18
4	3	4	19	23	22	23
5	3	4	23	28	27	28
6	3	4	27	33	32	33

Table 1: Size of TSs for Golog/LDL_f specifications in Warehouse.

input parameters, i.e., the number of rooms and the number of dishes, which jointly increase the complexity of the task. Figure 1c reports results with the number of rooms fixed to 9. In this setting, Golog specifications handle up to 6–7 dishes with programs of Type 1 and 2, while Type 3 programs scale to more than 30 dishes within the timeout threshold.

To complement the empirical analysis, we also report the size of the generated program transition systems (before computing the cross product with the domain). Table 1 presents the TS sizes for the Warehouse Robot benchmark. The TSs representing Golog specifications are slightly smaller than those obtained from LDL_f specifications. This is because Golog retains more structural information directly in the TS, which allows for a more compact representation. In particular, action preconditions are encoded on transitions, and whenever a precondition is not satisfied the execution moves to a sink state, effectively pruning infeasible behaviors early. This richer structural encoding has also a direct impact on synthesis. Interestingly, although the TS size increases from Type 1 to Type 3 programs, performance nonetheless improves significantly. This indicates that scalability is not determined by the size of the TS, but rather by the amount of procedural structure encoded in it. More structured programs yield TSs that better guide the synthesis process, resulting in substantially faster game resolution.

9 Conclusion

In this paper, we have shown that Golog provides an expressive and convenient language to define procedural specifications in a reactive synthesis context, and it allows for effective techniques to synthesize strategies in nondeterministic domains. Future research directions include mixing procedural and declarative specifications (Bacchus and Kabanza 2000; Baier et al. 2008), or performing strategic reasoning in multi-agent settings (De Giacomo, Lespérance, and Pearce 2010; De Giacomo, Lespérance, and Pearce 2016) using procedural specifications. A promising direction is combining our approach with abstraction-based frameworks (Banihashemi, De Giacomo, and Lespérance 2025). In particular, (De Giacomo, Lespérance, and Mancanelli 2025) used abstractions for solving generalized planning problems, but relied on an unspecified translation of NDBATs into LTL_f for applying synthesis tools. Our approach enables reasoning natively over action theories, and the procedural structure could help speed up the synthesis process. The notion of action theory abstraction could also be exploited to construct suitable abstractions that further reduce the synthesis effort.

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.

Declaration on Generative AI

During the preparation of this work, the authors used AI-based tools (ChatGPT and Claude) for grammar and spelling checks. The authors reviewed and edited the content as needed and take full responsibility for the content of this publication.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artif. Intell.* 116(1-2):123–191.

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.

Banihashemi, B.; De Giacomo, G.; and Lespérance, Y. 2025. Abstracting situation calculus action theories. *Artificial Intelligence* 104407.

Brafman, R. I.; De Giacomo, G.; and Patrizi, F. 2018. Ltlf/ldlf non-markovian rewards. In *AAAI*, 1771–1778. AAAI Press.

Bryant, R. E. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*

Brüggemann-Klein, A., and Wood, D. 1998. One-unambiguous regular languages. *Information and Computation* 140(2):229–253.

Camacho, A.; Bienvenu, M.; and McIlraith, S. A. 2019. Towards a unified view of AI planning and reactive synthesis. In *ICAPS*, 58–67. AAAI Press.

Caron, P.; Mignot, L.; and Miklarz, C. 2017. On the hierarchy of generalizations of one-unambiguous regular languages. *Theoretical Computer Science* 679:95–106.

Claßen, J., and Lakemeyer, G. 2008. A logic for non-terminating Golog programs. In *KR*, 589–599.

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

De Giacomo, G., and Favorito, M. 2021. Compositional approach to translate LTL_f/LDL_f into deterministic finite automata. In *ICAPS*.

De Giacomo, G., and Lespérance, Y. 2021. The nondeterministic situation calculus. In Bienvenu, M.; Lakemeyer, G.;

and Erdem, E., eds., *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, 216–226.

De Giacomo, G., and Rubin, S. 2018. Automata-theoretic foundations of FOND planning for ltlf and ldlf goals. In *IJCAI*, 4729–4735. ijcai.org.

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.

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.; Di Stasio, A.; and Parretti, G. 2025. PDDL to DFA: A symbolic transformation for effective reasoning. In *TIME*.

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

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 Mancanelli, M. 2026. Strategic reasoning over golog programs in the nondeterministic situation calculus. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

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 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.

Duret-Lutz, A.; Zhu, S.; Piterman, N.; De Giacomo, G.; and Vardi, M. Y. 2025. Engineering an ltlf synthesis tool. In *CIAA*, volume 15981 of *Lecture Notes in Computer Science*, 129–147. Springer.

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.

Han, Y.-S., and Wood, D. 2008. Generalizations of 1-deterministic regular languages. *Information and Computation* 206(9):1117–1125.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Morgan & Claypool.

Henriksen, J. G.; Jensen, J. L.; Jørgensen, M. E.; Klarlund, N.; Paige, R.; Rauhe, T.; and Sandholm, A. 1995. Mona: Monadic second-order logic in practice. In *TACAS*.

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.; 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.

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

Somenzi, F. 1998. Cudd: Cu decision diagram package release 2.3.0. In *University of Colorado at Boulder*.

Zhu, S., and Favorito, M. 2025. Lydiasyft: A compositional symbolic synthesis framework for $\mathrm{ltl_f}$ specifications. In *TACAS (1)*, volume 15696 of *Lecture Notes in Computer Science*, 295–302. Springer.

Zhu, S.; Tabajara, L. M.; Li, J.; Pu, G.; and Vardi, M. Y. 2017. Symbolic LTL_f synthesis. In *IJCAI*.