# Reasoning at the Right Level

## Abstraction and Synthesis of High-Level Programs

Matteo Mancanelli

# Reasoning at the Right Level

**When solving a problem, humans abstract and reason without even noticing.**

- We focus on what matters, hiding low-level details behind actions we trust to "just work".
- We leave choices open, resolving them only when the context demands it.

**Can we give agents the same ability — formally and verifiably?**

Two complementary problems:

- **Part I — Synthesis over high-level programs:** the reasoning *engine*.
- **Part II — Abstraction of action theories:** what makes reasoning *scale* and *generalize*.

# Foundations

# Situation Calculus & Basic Action Theories

**Situation Calculus Basics:**

- FO formalism for specifying dynamically-evolving worlds and reasoning about actions.
- A situation is a sequence of actions, starting from an initial situation $S_0$; $do(a, s)$ is the situation that results from doing action $a$ in situation $s$.
- A fluent is a predicate that depends on the situation.

**Basic Action Theories (BAT):**

$$D = \Sigma \cup D_{una} \cup D_{pre} \cup D_{ssa} \cup D_{S_0}$$

- $\Sigma$ are the foundational axioms for situations
- $D_{una}$ is the set of unique names axioms for actions
- $D_{pre}$ is a set of action precondition axioms (i.e., $Poss(A(\vec{x}), s) \equiv \Phi_A^{pre}(\vec{x}, s)$ for each action)
- $D_{ssa}$ is a set of successor state axioms (i.e., $F(\vec{x}, do(a, s)) \equiv \Phi_F^{ssa}(\vec{x}, s)$ for each fluent)
- $D_{S_0}$ is the initial situation description

# Nondeterministic Situation Calculus & Strategic Reasoning

**NDBAT:** every agent action is coupled with an environment reaction.

- The agent picks an action $a(\vec{x})$; the environment picks a reaction $e$.
- The actual transition is the system action $a(\vec{x}, e)$, yielding $do(a(\vec{x}, e), s)$.
- Reaction existence & independence: the agent can always act, and some environment reaction is always available.

**Angelic vs. adversarial execution:**

- Angelic: the agent resolves all nondeterminism.
- Adversarial: the environment is hostile; find a strategy $f$ (from situations to agent actions) that forces $\delta$ to termination against *all* reactions (written $AgtCanForceBy(\delta, f, S_0)$)

# Synthesis over High-Level Programs

# High-Level Programming in the Situation Calculus

**Motivation:**

- Express complex actions/programs for an agent
- Reason about their possible executions, preconditions, effects, etc.
- Use them to control the agent

**High-Level Programming:**

- A Middle Ground between Planning and Programming
- Planning can be very hard, but often we can sketch what a good plan might look like
- Instead of planning, view the agent's task as executing a high-level plan/program
- We allow nondeterministic programs to leave certain choices to be resolved at execution time through reasoning.
- This approach supports both deliberation and full scripting when appropriate.

# Golog

**Syntax:**

| | |
|---|---|
| $a(\vec{x})$ | agent action |
| $\phi?$ | test a condition |
| $\delta_1; \delta_2$ | sequence |
| $\delta_1 \mid \delta_2$ | nondeterministic branch |
| $\pi x.\delta(x)$ | nondeterministic choice of $x$ |
| $\delta^*$ | nondeterministic iteration |

**Semantics:** specified in terms of single steps, using $Trans(\delta, \vec{x}, s, \delta', \vec{x}', s')$ and $Final(\delta, \vec{x}, s)$.

**Program Execution Task:** Given a domain theory $\mathcal{D}$ and a program $\delta$, find a sequence of actions $\vec{a}$ such that:

$$\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$$

# Golog: Syntactic Closure

**Syntactic closure ($\Gamma_{\delta_0}$):** defined inductively as follows:

$$\delta_0, \mathit{nil} \in \Gamma_{\delta_0}$$
**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** $\delta; \delta^* \in \Gamma_{\delta_0}$

**Theorem**

*The syntactic closure $\Gamma_{\delta_0}$ is linear in the size of the program $\delta_0$.*

# Program Graph: Construction

**Key Idea:**

- Each node represents a subprogram from $\Gamma_{\delta_0}$ (the remaining part of the execution).
- Edges correspond to possible execution steps, annotated with guards (preconditions and test formulas).
- A label is associated with each node to indicate whether the subprogram is final.

**Properties:**

- Provides a fully syntactic, domain-independent, and compact representation.
- Execution paths in the graph correspond to transitions in standard Golog semantics.
- If the program is situation determined, the program graph is deterministic

# Program Graph

**Program Graph:**

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

where

- $\Phi \times \mathcal{A} \times \Psi$ is the alphabet,
  - $\Phi$ and $\Psi$ are sets of formulas over tests and *Poss*
  - $\mathcal{A}$ is the set of actions
- $Q = \Gamma_{\delta_0}$ is the set of nodes, corresponding to the programs in the syntactic closure of $\delta_0$.
- $q_0 = \delta_0$ is the initial program
- $\tau(q, \varphi, a, \psi, q')$ is a transition from $q$ to $q'$ via action $a$ when pre-condition $\varphi$ holds before and post-condition $\psi$ holds after.
- $\mathcal{L}(q) = F(q)$ is a label specifying the condition under which $q$ can terminate.

# Example: Coffee-Delivery Robot

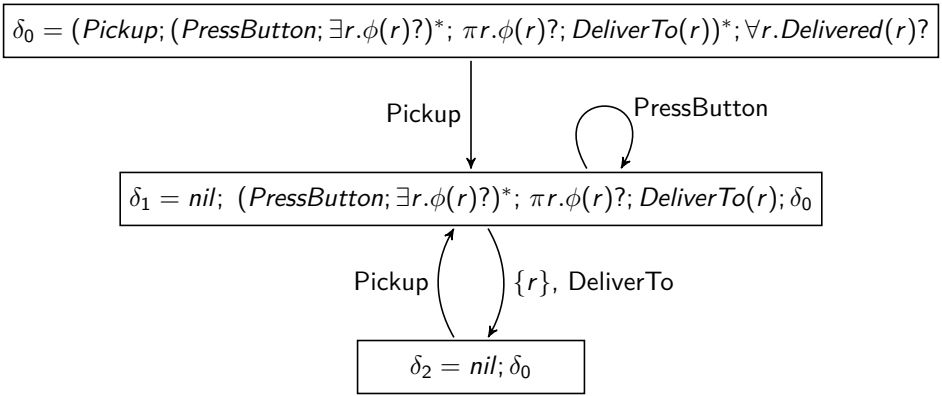
**Description:**

- A robot must deliver coffee to rooms behind doors.
- The doors are closed.
- Pressing a button opens a random door (environment nondeterminism).
- If the room has already been served, it is ignored, and a new room should be selected by pressing the button again.
- If the room has not been served, the robot delivers the coffee.
- Goal: all rooms eventually receive coffee.

**Program:**

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

# Example: Coffee-Delivery Robot



# From First-Order to Propositional

**First-Order Setting** (De Giacomo, Lespérance, Mancanelli — AAAI 2026):

- The general framework: NDBATs over first-order domains with objects and data.
- Program graph $\times$ domain yields a game arena.
- Strategies are computed by a regression-based fixpoint reasoning.
- Price: FO reasoning is undecidable, so the fixpoint is sound but may not terminate.

**Propositional Setting** (De Giacomo, Lespérance, Mancanelli, Parretti — KR 2026)

- The state space becomes finite; synthesis is decidable and complete.
- We can leverage automata-based synthesis tools.
- We can formally compare with $\mathrm{LTL}_f/\mathrm{LDL}_f$ synthesis.
- We can prove tight complexity bounds and run experiments.

# Propositional Setting

**What Changes:**

- No objects, no quantifiers; fluents are propositions $F(s)$.
- No pick operator $\pi x.\delta$; it can be encoded as $\delta(o_1) \mid \cdots \mid \delta(o_n)$.
- We allow for tail-recursive procedures to maintain full expressiveness.

**Syntactic Situation Determinacy (SSD):**

- SD program: after executing an action, the remaining program is uniquely determined by the new situation; SD is a semantic notion (depends on the model $M$).
- SSD program: after executing an action, if different remaining programs are possible, their continuation conditions are mutually exclusive; we can check it without the domain.

**Theorem**

*SSD Golog programs (with tail-recursive procedures) have the same expressive power as regular expression (or MSO, or $LDL_f$) on finite traces.*

# From Program Graphs to Game Arenas

**Goal:** build a finite game arena where we can solve synthesis.

**Intuition:**

1. Program graph $\mathcal{G}$: captures the control flow of the Golog program (nodes = subprograms, edges = guarded transitions). Size is linear in $\delta_0$.
2. Abstract domain $A_M$: the NDBAT induces an infinite TS over situations. Since the domain is propositional, we collapse situations with the same fluent valuation into a single state $p \in 2^{\mathcal{F}}$. This gives a finite TS, and the two are related by a bisimulation.
3. Game arena $\mathcal{A}_{M,\mathcal{G}}$: the cross product $\mathcal{G} \times A_M$ synchronizes program state with domain state. A state is a pair $(\delta, p)$. A play is a sequence of states; a strategy is winning for the agent if every induced play reaches a final state.

# Game Arena

**Game Arena:**

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

where

- $\Sigma_M = \{a(e) \mid a \in \mathcal{A}, e \in React_a\}$ is the alphabet
- $Q \times P$ is the set of game states (program state $\times$ domain state)
- $(\delta_0, p_0)$ is the initial state
- $Tr_A((\delta, p), a(e)) = (\delta', p')$ when:
  (i) there is a matching transition in $\mathcal{G}$ with guard $\varphi$ satisfied by $p$,
  (ii) the domain transitions from $p$ to $p'$ under $a(e)$
- $Fin_A = \{(\delta, p) \mid p \models F(\delta)\}$

**Theorem**

*A winning strategy exists in $\mathcal{A}_{M,\mathcal{G}_\delta}$ iff the agent can force successful execution of $\delta$ in $M$.*

# Complexity: Golog vs $\text{LDL}_f$

| | **Golog** | **$\text{LDL}_f$** |
|---|---|---|
| Specification $\rightarrow$ TS | linear | doubly exponential |
| Game arena size (in spec.) | linear | doubly exponential |
| Solving the game | linear | linear |
| **Overall (in spec.)** | **linear** | **2EXPTIME** |

*Note:* complexity remains EXPTIME in the size of the NDBAT domain for both approaches.

# Implementation: Syft4Golog

**Architecture:**

- Built on top of LydiaSyft.
- FOND domains specified in PDDL, converted to symbolic DFAs.
- Game arena constructed symbolically via BDDs.

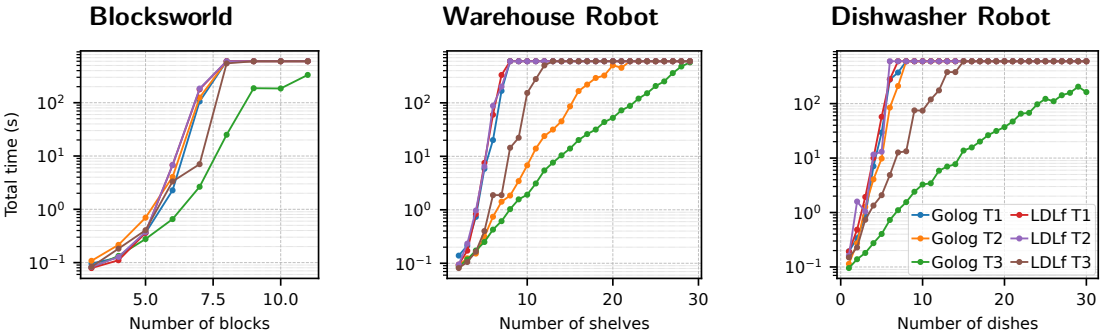
**Baseline:** translate Golog $\rightarrow$ LDL$_f$ and use LDL$_f$ synthesis.

**Three program templates per domain:**

- **Type 1** (low guidance): $(pick\_any\_action)^*; goal?$
- **Type 2** (medium): $(subtask_1 | \ldots | subtask_n)^*; goal?$
- **Type 3** (high): $subtask_1; \ldots; subtask_n; goal?$

# Experimental Results

End-to-end synthesis time (log scale), timeout = 600s




**Key takeaways:**

- With low guidance (Type 1), Golog $\approx$ LDL$_f$ — procedural structure doesn't help.
- With more guidance (Types 2–3), Golog consistently outperforms LDL$_f$.

# Experimental Results

| | Type 1 | | Type 2 | | Type 3 | |
|---|---|---|---|---|---|---|
| $n$ | Golog | $\text{LDL}_f$ | Golog | $\text{LDL}_f$ | Golog | $\text{LDL}_f$ |
| 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: Size of TSs for Golog/$\text{LDL}_f$ specifications in Warehouse.

Scalability is driven by procedural structure, not TS size: more structured programs guide the synthesis process more effectively.

# Abstraction of Action Theories

# Motivation: Generalized Planning

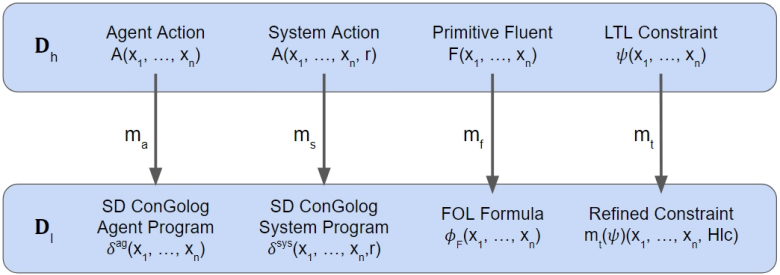
**Generalized planning:** find a *single* strategy that solves many (possibly infinitely many) planning instances at once.

**Idea — reason at an abstract level:**

- each instance is a model of a concrete low-level (LL) action theory;
- a high-level (HL) action theory abstracts away LL detail and captures *all* instances at once;
- synthesize a strategy automatically at the HL...
- ...and **refine** it back into a correct solution for *every* LL instance.

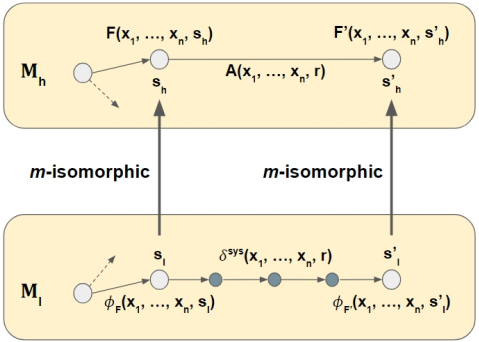
# Refinement Mapping $m$

A **refinement mapping** $m$ is a tuple $\langle m_a, m_s, m_f, m_t \rangle$. In defining $m_t$ to map HL constraints, we suppose that the LL theory tracks when refinements of HL actions end using a state formula $Hlc(s)$, meaning that a HL action has just been completed in situation $s$.



# *m*-Simulation

Two situations $s_h$ and $s_l$ are ***m*-isomorphic** iff they evaluate all HL fluents the same. Two models $M_h$ and $M_l$ are ***m*-similar** if *(i)* the initial situations are *m*-isomorphic and *(ii)* the resulting $s_l'$ after executing $m(A)$ at the LL is *m*-isomorphic to the resulting $s_h'$ after executing $A$ at the HL.



# Temporally Lifted Abstractions

Consider an HL NDBAT $\mathcal{D}_h$ equipped with a set of HL state constraint $\Psi$, a model $M_h$ of $\mathcal{D}_h$, a LL NDBAT $\mathcal{D}_l$ and a refinement mapping $m$.

**Definition**

We have a **temporally lifted abstraction** wrt $m$ if and only if

- a model $M_h$ of $D_h$ ***m*-simulates every model** $M_l$ of $D_l$
- **for every** high-level LTL **trace constraint** $\psi$,
  $M_h \models \exists p_h.Starts(p_h, S_{0_h}) \wedge Holds(\psi, p_h)$ and
  $D_l \models \forall p_l.Starts(p_l, S_{0_l}) \supset Holds(m_t(\psi), p_l)$

# Example (Minimum in a List)

**Description:** We illustrate our framework addressing the problem of finding the minimum value in a singly-linked list.

**LL:** A list is described **deterministically** by its head and each node's value and successor. We also use an iterator and a register.

**HL:** We abstract details using **nondeterministic** actions: *next* (moves the cursor) and *update* (updates the register). The **environment reaction** of *next* indicates if the end is reached.

**LTL Trace Constraint:** $(\Box\Diamond \mathit{doneNext}) \rightarrow \Diamond\neg \mathit{hasNext}$
moving repeatedly to the next node eventually leads to the last one

# Temporally Lifted Abstractions

We define $AgtCanForceByIf(Goal, Cstr, f, s)$, meaning that the agent **can force** a LTL $Goal$ **by following strategy** $f$ in $s$ if LTL path constraint $Cstr$ **holds**.

**Theorem**

Consider a temporally lifted abstraction and a LTL goal.

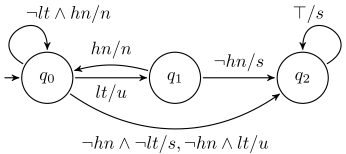
If $M_h \models \exists f_h.AgtCanForceIf(Goal, Cstr, f_h, S_0)$,
then there exist a refined strategy $f_l$ such that
$\mathcal{D}_l \models AgtCanForceByIf(m(Goal), True, f_l, S_0)$

# Example Cont. (Minimum in a List)

**HL LTL Goal:**

$$\Diamond\Box\neg \mathit{hasNext}$$
$$\Box(\mathit{lowerThan} \leftrightarrow \bigcirc \mathit{doneUpdate})$$

The controller can be **generated automatically** by an LTL synthesis engine like Strix. By the theorem, we know that there exists a refinement of this strategy at the LL.



# From Using to Building Abstractions

**So far:** given an abstraction, we can synthesize a HL strategy and refine it to the LL.

**The missing piece:** the framework assumes the abstraction is already given.

- Designing a sound abstraction by hand is hard and error-prone
- Prior work characterizes **what** a sound/complete abstraction is, not **how** to obtain one

**Goal:** an incremental, tool-supported method to **construct** abstractions step by step, with formal guarantees at each step.

# Synthesizing Action Theory Abstractions

**Idea:** an engineer builds the abstraction incrementally, and the system **validates** each step.

**At each step, the engineer can:**

- introduce abstract actions, each implemented by a ConGolog program over the LL
- introduce abstract fluents, each mapped to a LL state formula
- drop the LL actions/fluents that should be hidden

**The system guarantees soundness/completeness:**

- it **accepts** a step only if it yields a sound (and/or complete) abstraction
- a **failing** step is rejected with an explanation, guiding the engineer
- the relation is transitive, so steps compose: $\mathcal{D}_l = \mathcal{D}^0 \rightarrow \mathcal{D}^1 \rightarrow \cdots \rightarrow \mathcal{D}^k = \mathcal{D}_h$

# Generating the Axioms

**Action abstraction:** for a new HL action $a(\vec{x})$ implemented by a program $\delta_a$:

- the precondition is obtained by **regression**: $Poss(a(\vec{x}), s) \equiv \mathcal{R}[True,\ \delta_a(\vec{x})]$
- regressing $\delta_a$ also reveals **which HL fluents it makes true/false**, which are then assembled into successor state axioms

**Validation:** an SMT solver (e.g., Z3) checks that the synthesized HL preconditions and effects agree with the LL implementation; soundness/completeness conditions are entailment checks.

# Example (Travel Booking)

**LL:** fine-grained booking actions — *selectHotelRoom*, *applyHotelReward*, *payHotelRoom*, *bookAirBnbRoom*, ...

**Abstraction step:** introduce one HL action $bookHotelRoom(c, r, h)$ mapped to

$$selectHotelRoom;\ \textbf{if}\ HasRewardPlan\ \textbf{then}\ applyHotelReward;\ payHotelRoom$$

- drop the intermediate fluents *SelectedHotelRoom*, *PaidHotelRoom*
- introduce $BookedHotelRoom \doteq SelectedHotelRoom \wedge PaidHotelRoom$

**The method** regresses the program to synthesize *Poss* and the SSAs of *bookHotelRoom*, and verifies via SMT that the step is a sound and complete abstraction.

# Closing the Loop

# Abstract, then Synthesize

**The two parts close a loop:**

- **Part II** lifts an action theory to a propositional HL abstraction with LTL constraints, reducing generalized planning to synthesis over the abstraction.
- But it assumed an unspecified translation of NDBATs into a synthesis problem.
- **Part I** supplies exactly that: a native, linear NDBAT-to-game reduction for high-level programs.

**The idea:** use abstraction not only to *generalize*, but to make strategic reasoning practical:

**abstract a theory, then synthesize over it** — as we do when we reason at the right level.

# Other Ongoing Work: Neuro-Symbolic AI

**Injecting LTL$_f$ knowledge into neural models** via differentiable automata (DeepDFA):

- Predictive process monitoring: constrain neural suffix predictions to satisfy temporal background knowledge.
- Safe offline RL: enforce LTL$_f$ safety constraints inside Decision/Trajectory Transformers.

**New direction** — neuro-symbolic runtime monitoring and specification adaptation:

- Adapt the specification from data via gradient-based updates.
- Bridge to Part II: the same verification-feedback loop could learn to construct action-theory abstractions automatically, without a human knowledge engineer.

# Open Directions

**Where this goes next:**

- Engineer the construction of the Golog TS: improving the efficiency of building process.
- Construct more sophisticated abstractions: including nondeterministic programs/theories.
- Procedural + declarative + abstraction: mixing Golog and $\text{LTL}_f/\text{LDL}_f$ specifications, in multi-agent and reactive settings.
- Complexity of abstract synthesis: does abstracting first *provably* reduce synthesis cost?
- Discovering abstractions automatically: learn good "views" of a domain (even with LLMs), rather than hand-crafting them.