From Abstract Plans to Concrete Strategies: Synthesizing Controllers in Nondeterministic Domains via Situation Calculus and Golog

Matteo Mancanelli

thanks to Giuseppe De Giacomo and Yves Lespérance

April 9, 2025

Table of Contents

Outline

1. Formal Background
   - Situation Calculus
   - Golog
   - NDBATs
2. Abstractions for Generalized Planning
3. Automata-based Golog Synthesis

Knowledge Representation for AI Agents

Knowledge Representation (KR) aims at building systems that **know about their world** and can **act in an informed way** within it.

Key principles of KR:

- Knowledge is represented formally
- Reasoning procedures can derive logical consequences
- Reasoning supports informed decision-making

Robot Example (Domain Description)

A robot, *self*, inhabits an environment formed by rooms connected by doors (open or closed). Some rooms are control rooms that contain a button to open all doors.

Example configuration:

- Rooms: A, B, C
- Control room: B
- Doors: d_{AB} (between A and B), d_{AC} (between A and C)
- Initially: *self* is in room A, door d_{AB} is open, and d_{AC} is closed

Robot Example (Actions)

The robot *self* can **move across rooms** and, when in a control room, can **press a button to open all doors**.

Actions available:

- *goto*(x) *(move to room x)*
  - **Precondition:** there must be an open door between *self*'s current room and x
  - **Effect:** *self* is now in room x
- *openAllDoors*()
  - **Precondition:** *self* must be in a control room
  - **Effect:** all closed doors become open

Robot Example (Predicates)

Static Predicates:

- $Room(x)$: x is a room
- $ControlRoom(x)$: x is a control room
- $Door(x, y, z)$: x is a door between rooms y and z

Dynamic Predicates (Fluents):

- $Open(x)$: door x is open
- $SelfIn(x)$: robot *self* is in room x

Instance:

- **Static Facts:**
  - $Room(x) \equiv (x = A \vee x = B \vee x = C)$; $ControlRoom(x) \equiv (x = B)$
  - $Door(x, y, z) \equiv ((x = d_{AB} \wedge y = A \wedge z = B) \vee (x = d_{AB} \wedge y = B \wedge z = A) \vee (x = d_{AC} \wedge y = A \wedge z = C) \vee (x = d_{AC} \wedge y = C \wedge z = A))$
- **Initial Fluent Values:**
  - $Open(x) \equiv (x = d_{AB})$; $SelfIn(x) \equiv (x = A)$

Robot Example (Formalization)

Action Preconditions and Effects:

- $goto(x)$
  - **PRE:** $\exists r.\, SelfIn(r) \wedge \exists d.\, Door(d, r, x) \wedge Open(d)$
  - **EFF:** $SelfIn(x) \wedge \neg\exists r.\, (r = x \wedge SelfIn(r))$
- $openAllDoors()$
  - **PRE:** $\exists r.\, SelfIn(r) \wedge ControlRoom(r)$
  - **EFF:** $\forall d.\, pre[Closed(d)] \supset Open(d)$

Problem 1: We need to refer to both the **state before** and the **state after** the action!
Problem 2: Is this enough to fully describe the effects? What about the fluents that **do not change**? (Frame Problem)

Situation Calculus

Situation Calculus is a foundational formalism for reasoning about actions and change. It is a first-order, multi-sorted logical language where states are represented as **situations**, defined inductively.

Key Sorts:

- **Objects:** domain elements (e.g., A, B, C, d_{AB})
- **Actions:** events that progress the system (e.g., $goto(x)$)
- **Situations:** histories of actions, describing world states
- **Fluents:** predicates that may change across situations

Situation Calculus: Situations

Situations denote states of the world resulting from sequences of actions.

- S_0: the initial situation (no actions performed yet)
- $do(a, s)$: the situation that results from doing action a in situation s
- Situations form an infinite tree of histories (built inductively)

Example:

$$do(goto(C), do(goto(A), do(openAllDoors(), do(goto(B), S_0))))$$

This represents the situation reached by: going to room B, opening all doors, then going to A, then to C.

Situation Calculus: Fluents

Fluents are predicates (of functions) whose truth may vary across situations. They are written as predicate (or functional) symbols taking an additional situation argument.

Examples:

- $Open(d, s)$: door d is open in situation s
- $SelfIn(r, s)$: robot *self* is in room r in situation s

Initial Fluent Values:

- $Open(x, S_0) \equiv (x = d_{AB})$
- $SelfIn(x, S_0) \equiv (x = A)$

Situation Calculus: Poss

We use a special predicate symbol $Poss(a, s)$ to express that an action a is executable in situation s.

Examples:

- $Poss(goto(B), S_0)$: robot *self* can move from its current room (which is A in S_0) to room B.
  This holds, since d_{AB} is open.
- $Poss(openAllDoors(), S_0)$: robot *self* can push the button to open all doors in situation S_0.
  This does NOT hold, because A is not a control room.

Running Example: Formalizing Actions

Action: $goto(x)$

- **PRE:** $Poss(goto(x), s) \equiv \exists r.\, SelfIn(r, s) \wedge \exists d.\, Door(d, r, x) \wedge Open(d, s)$
- **EFF:** $SelfIn(x, do(goto(x), s)) \wedge \neg \exists r.\, (r = x \wedge SelfIn(r, do(goto(x), s)))$

Action: $openAllDoors()$

- **PRE:** $Poss(openAllDoors(), s) \equiv \exists r.\, SelfIn(r, s) \wedge ControlRoom(r)$
- **EFF:** $\forall d.\, Closed(d, s) \supset Open(d, do(openAllDoors(), s))$

Note: This solves Problem 1: we can now reference both the situation before (s) and after ($do(a, s)$) the action.

Frame Problem

Frame Problem: How do we specify that most fluents do not change after an action?

Examples:

- Pushing the button does not change where the robot is:
  $SelfIn(r, s) \supset SelfIn(r, do(openAllDoors(), s))$
- Moving the robot doesn't change the state of any door:
  $Open(d, s) \supset Open(d, do(goto(x), s))$
  $\neg Open(d, s) \supset \neg Open(d, do(goto(x), s))$

These are called frame axioms - and we need many of them, one per fluent-action pair!

Problem: This leads to a large number of axioms and is error-prone.

Solution to the Frame Problem

Reiter's Solution:

- Use successor state axioms instead of effect axioms; one successor state axiom per fluent.
- Use precondition axioms for specifying preconditions; one precondition axiom per action.
- The length of a successor state axiom is roughly proportional to the number of actions which affect the truth value of the fluent.

Note: The conciseness and perspicuity of the solution relies on:

- quantification over actions
- the assumption that relatively few actions affect each fluent
- the completeness assumption for effects

Successor State Axioms

Step 1: Normalize effect axioms (for fluent F):

- Positive effect: $\Phi_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$
- Negative effect: $\Phi_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$

Step 2: Enforce explanation closure:

- If F becomes true:
  $\neg F(\vec{x}, s) \wedge F(\vec{x}, do(a, s)) \supset \Phi_F^+(\vec{x}, a, s)$
- If F becomes false:
  $F(\vec{x}, s) \wedge \neg F(\vec{x}, do(a, s)) \supset \Phi_F^-(\vec{x}, a, s)$

Step 3: Define successor state axiom:

$$F(\vec{x}, do(a, s)) \equiv \Phi_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg \Phi_F^-(\vec{x}, a, s))$$

Running Example: Successor State Axioms

Successor State Axiom for $SelfIn(x, s)$:

$$SelfIn(x, do(a, s)) \equiv (a = goto(x)) \vee (SelfIn(x, s) \wedge \neg\exists y.\,(a = goto(y) \wedge y \neq x))$$

Successor State Axiom for $Open(d, s)$:

$$Open(d, do(a, s)) \equiv (a = openAllDoors() \wedge Closed(d, s)) \vee (Open(d, s) \wedge \neg False)$$

Note: These axioms compactly encode both the effects *and* the persistence of fluents.

Comparison with STRIPS

STRIPS operator for *goto*:

- **PRE:** $SelfIn(f) \wedge Door(d, f, t) \wedge Open(d)$
- **ADD:** $SelfIn(t)$
- **DEL:** $SelfIn(f)$

STRIPS Representation Characteristics:

- States are represented as databases of ground atoms
- Actions update these databases by adding/removing atoms
- Assumes fully known initial state
- Not a full logic: lacks quantifiers, functions, and expressive reasoning

By contrast, Situation Calculus is a proper logical language with formal semantics and supports reasoning beyond state updates.

Situation Calculus: Basic Action Theories (BATs)

Basic Action Theory (BAT)

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

Components:

- Σ: foundational axioms for situations (SOL)
- D_{una}: unique names axioms for actions
  - e.g., $A_1(\vec{x}) \neq A_2(\vec{y})$ for distinct action names A_1 and A_2
- D_{pre}: precondition axioms
  - $Poss(A(\vec{x}), s) \equiv \Phi_A^{pre}(\vec{x}, s)$
- D_{ssa}: successor state axioms
  - $F(\vec{x}, do(a, s)) \equiv \Phi_F^{ssa}(\vec{x}, a, s)$
- D_{S_0}: initial situation description
  - Only S_0 is used in fluents within D_{S_0}

Reasoning in Situation Calculus

Key Reasoning Tasks:

- Satisfiability: is the basic action theory consistent?
- Projection: what holds after executing a sequence of actions?
- Executability: can a sequence of actions be legally performed?
- Planning: find a sequence of actions that achieves a desired goal

Regression: reduces reasoning about future situations (second-order) to reasoning about the initial situation only (first-order)

- Transforms formulas about $do(a, s)$ into equivalent ones about s
- Achieved by replacing fluents using their successor state axioms
- By iterating, we regress any future situation back to S_0

Limitations: Temporal Reasoning

Regression is not sufficient for more expressive temporal properties:

- "There exists a future situation where α holds"
- "α always holds in all reachable situations"
- "Eventually, no matter what actions are taken, α will hold"
- "Whenever α holds, then eventually β will hold"

Beyond Projection and Executability: When we deal with such temporal properties we need verification techniques, most of which assume finite number of states (i.e., finite object domain in the SitCalc).

- If we assume finite number of objects, then we can model check SitCalc Action Theories!

These require temporal logic and verification techniques.

High-Level Programming in the Situation Calculus

Motivation: We want to be able to:

- Express complex actions/programs for an agent.
- Reason about their executions, preconditions, and effects.
- Use these programs to control the agent's behavior.

High-Level Programming serves as a middle ground between planning and scripting:

- Instead of complete planning, we let the agent execute a high-level plan or 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.
- It relates closely to work on planning with domain-specific search control.

Golog

Golog constructs include:

α	(primitive action)
$\phi?$	(test a condition)
$\delta_1 ; \delta_2$	(sequential composition)
$\delta_1 \mid \delta_2$	(nondeterministic branching)
$\pi \vec{x}.\, \delta$	(nondeterministic choice of arguments)
δ^*	(nondeterministic iteration)

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

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

Here, $Do(\delta, s, s')$ means that program δ, when executed starting in situation s, can legally terminate in situation s'.

Nondeterminism in Golog

A nondeterministic program may have several possible executions. For example:

- Let $ndp_1 = (a \mid b); c$.
- Assuming all actions are executable, we have:

$$Do(ndp_1, S_0, s) \equiv \big(s = do([a, c], S_0)\big) \vee \big(s = do([b, c], S_0)\big)$$

When a test condition or action precondition fails, the interpreter backtracks to try alternative nondeterministic choices. For instance:

- Let $ndp_2 = (a \mid b); c; P?$.
- If the test P is initially true but becomes false when a is executed, then:

$$Do(ndp_2, S_0, s) \equiv \big(s = do([b, c], S_0)\big)$$

- The interpreter will arrive at this result by backtracking.

Golog Semantics

Two complementary semantics for Golog:

- **Standard Semantics:** based on the predicate $Do(\delta, s, s')$
- **Computational Semantics:** based on transition systems, defined via:
  - $Trans(\delta, s, \delta', s')$: The configuration (δ, s) can take a single execution step, transitioning to (δ', s') (a primitive action or a test).
  - $Final(\delta, s)$: The configuration (δ, s) may be considered completed.

Note that $Do(\delta, s, s')$ can be defined in terms of $Trans^*$ and $Final$, where $Trans^*$ is the transitive closure of $Trans$

Nondeterministic Situation Calculus

Motivation:

- In standard Situation Calculus, atomic actions are deterministic
- The resulting situation from doing a in s is uniquely $do(a, s)$
- But many real-world actions are nondeterministic — e.g., flipping a coin
- Prior solutions don't distinguish between agent choices and environment outcomes

Nondeterministic Situation Calculus:

- Simple, elegant extension of standard SitCalc to capture nondeterminism
- Preserves Reiter's solution to the frame problem
- Allows regression for projection queries

Nondeterministic Situation Calculus

The Approach:

- Outcome of a nondeterministic action is determined by the agent action and the environment's reaction
- Every action type/function $A(\vec{x}, e)$ takes an additional environment reaction parameter e ranging over new sort Reaction
- We call $A(\vec{x}, e)$ a system action, and the relative reaction-suppressed version $A(\vec{x})$ an agent action
- This lets us quantify **separately** over agent decisions and environmental responses
- The nondeterminism associated with agent choices is angelic (goal-directed), and that associated with environment choices is devilish (adversarial)

Nondeterministic Basic Action Theories (NDBATs)

A **Nondeterministic Basic Action Theory** (NDBAT) is a BAT where:

- Each action has a reaction parameter e: $A(\vec{x}, e)$
- For each agent action, we have an agent action precondition:

$$Poss_{ag}(A(\vec{x}), s) \doteq \phi_A^{agPoss}(\vec{x}, s)$$

- Reaction independence: agent action precondition must be independent of any environment reaction:

$$\forall e.\, Poss(A(\vec{x}, e), s) \supset Poss_{ag}(A(\vec{x}), s)$$

- Reaction existence: if $Poss_{ag}$ holds, some environment reaction must make the system action possible:

$$Poss_{ag}(A(\vec{x}), s) \supset \exists e.\, Poss(A(\vec{x}, e), s)$$

- These conditions must be entailed by the theory to qualify as an NDBAT

Structure of NDBATs

As in standard SitCalc, we define:

- **System action preconditions:** specify how environment can react
- **Successor state axioms:** describe fluent changes under system actions
- **Initial situation axioms:** state what holds in S_0
- **Unique names + foundational axioms**

Key difference: agent actions are abstracted away from actual outcomes — outcomes are mediated by reactions.

FOND Planning

FOND = Fully Observable Nondeterministic Planning

- Planning with actions that may have multiple possible outcomes
- Goal: synthesize a **strong plan**, i.e. a strategy that succeeds **no matter how** the environment behaves
- We represent a strategy as a function f from situations to agent actions
- We define $AgtCanForceBy(Goal, s, f)$ to state that f forces $Goal$ in s
- It has been shown that **any** FOND domain can be encoded as an NDBAT

Outline

Planning Setting

- **Environment Model (DOM):**
  - The domain specifies the environment's behaviors in response to agent's action.
  - DOM is expressed by specific formalisms such as STRIPS, ADL, and PDDL.
  - DOM generates a (possibly nondeterministic) transition system
- **Agent Task (GOAL):**
  - The GOAL specifies the task to achieve.
  - It is expressed as reaching a state of the domain with desired properties.
- **Planning Problem:**
  - Find a strategy that ensures the GOAL is met within the domain

Generalized Planning

Key Features:

- A domain class defines a (possibly infinite) family of planning problems
- Each instance differs in its specific initial state or set of objects
- The strategy must solve every instance in the class

Goal: Synthesize a single strategy that solves **multiple instances** of a planning problem.

Challenge:

- Solutions must generalize beyond fixed initial states
- Need to reason over a symbolic abstraction that captures all instances

Our Approach

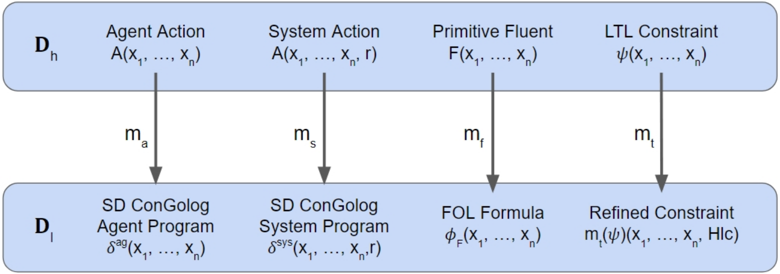
Objective: We introduce a novel formal framework to address Generalized Planning problems by generating a strategy that solves multiple (possibly infinite) similar planning problem instances.

Framework Principles:

- We use an abstraction to encompass all problems instances
- Each instance is a model of a concrete LL action theory
- A HL action theory/model abstracts away LL details
- We can synthetize automatically a strategy at the HL
- We formally prove that there exists a refinement of the strategy at the LL to solve all problem instances

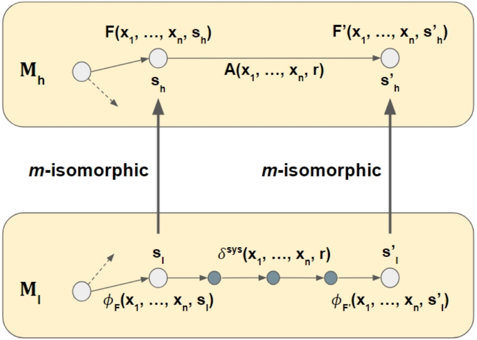
Refinement Mapping m

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 Ψ, 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** ψ,
  $M_h \models \exists p_h.Starts(p_h, S_{0_h}) \wedge Holds(\psi, p_h)$ 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.

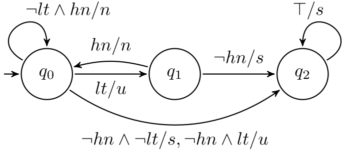
If $M_h \models \exists f_h.AgtCanForceIf(Goal, Cstr, f_h, S_0)$,
then there exist a refined strategy f_l such that
$\mathcal{D}_l \models AgtCanForceByIf(m(Goal), True, f_l, S_0)$

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.



Outline

Deterministic Domains and Transition Systems

A deterministic domain $D = (F, A, I)$ induces a transition system:

$$TD = (S, A, s_0, \alpha, \delta)$$

where:

- F: fluents (propositional variables)
- A: actions
- $S = 2^F$: set of all states
- s_0: initial state
- $\alpha(s) \subseteq A$: available actions in s
- $\delta(s, a) = s'$: deterministic state transition

A trace is a sequence:

$$s_0, a_1, s_1, \ldots, a_n, s_n$$

where each $a_i \in \alpha(s_i)$ and $s_{i+1} = \delta(s_i, a_i)$.

Game Arena Induced by a Nondeterministic Domain

In the nondeterministic setting, we model the domain as a game arena:

$$TD = (2^F, A, s_0, \alpha, \delta)$$

- F: fluents controlled by the environment
- A: agent actions
- s_0: initial state
- $\alpha(s) \subseteq A$: actions available in s
- $\delta(s, a, s')$: environment nondeterministically picks s'

Agent: chooses action a
Environment: chooses resulting state s'

Planning in Nondeterministic Domains

Given a nondeterministic domain D and a goal G:

- Find an agent strategy σ_a such that, for every compliant environment strategy σ_e, we have that $play(\sigma_a, \sigma_e) = s_0, a_1, \ldots, a_n, s_n$ satisfies $s_n \models G$
- That is: G must hold at the end of every possible execution trace

Winning strategy: a strategy σ_a that guarantees reaching G regardless of how the environment behaves

Program Synthesis

Goal: Mechanically translate a formal specification into a program that is guaranteed to satisfy it.

Classical vs. Reactive Synthesis:

- Classical: Synthesize transformational (batch) programs
- Reactive: Synthesize controllers or protocols for ongoing interactive computation

Synthesis

- Agent and environment play a game with LTL/LTL$_f$ specs as winning condition
- Agent picks controllable output $Y \in 2^{\mathcal{Y}}$
- Environment picks uncontrollable input $X \in 2^{\mathcal{X}}$
- A round consists of both choosing their values
- A play is a finite trace τ over $\mathcal{X} \cup \mathcal{Y}$
- Agent decides when to stop
- Specification is an LTL$_f$ formula φ
- Agent wins if $\tau \models \varphi$

Synthesis and Planning from LTL$_f$ Specifications

- LTL$_f$ formulas can be compiled into finite-state automata
- From this, we can perform:
  - Synthesis: create controllers that guarantee satisfaction of φ
  - Planning: build a strong policy from a FOND domain + LTL$_f$ goal
- Both are solved as 2-player games
- Recent work extends this framework:
  - richer logics (e.g., PPLTL)
  - stochastic / fair / partially observable environments

Overview

Context and motivation:

- Automated agent synthesis has been extensively studied in temporal logic frameworks
- LTL_f synthesis is effective for reactive systems but lacks explicit procedural constructs
- We introduce a graph-based synthesis framework tailored for procedural Golog specifications

Core Focus of This Work:

- Leveraging syntactic closure for efficient Golog synthesis
- Constructing program graphs to systematically represent program executions
- Integrating program graphs within FOND domains
- Proving the computational feasibility of our approach

Syntactic closure: after any one-step transition, the set of all possible remaining programs is finite. Following (De Giacomo et. al., 2016), it is possible to define inductively the syntactic closure Γ_{δ_0} of a program δ_0, as follows:

$$\begin{array}{l} \delta_0, \mathit{nil} \in \Gamma_{\delta_0} \\ \text{if } \delta_1; \delta_2 \in \Gamma_{\delta_0} \text{ and } \delta_1' \in \Gamma_{\delta_1}, \text{ then } \delta_1'; \delta_2 \in \Gamma_{\delta_0} \text{ and } \Gamma_{\delta_2} \subseteq \Gamma_{\delta_0} \\ \text{if } \delta_1 \mid \delta_2 \in \Gamma_{\delta_0}, \text{ then } \Gamma_{\delta_1}, \Gamma_{\delta_2} \subseteq \Gamma_{\delta_0} \\ \text{if } \delta^* \in \Gamma_{\delta_0}, \text{ then } \delta; \delta^* \in \Gamma_{\delta_0} \end{array}$$

Theorem

The syntactic closure Γ_{δ_0} is linear in the size of the program δ_0.

Program Graph

To construct the graph $\mathcal{G}$ of a Golog program, we leverage on the auxiliary definition of T and F, based on Trans and Final:

$$
\begin{array}{lcl}
T(a,a) & = & \{(Poss(a), nil)\} \\
T(a,b) & = & \{\} \\
T(\varphi?,a) & = & \{\} \\
T(\delta_1;\delta_2,a) & = & \{(\neg F(\delta_1) \wedge \varphi, \delta_1';\delta_2) \mid (\varphi,\delta_1') \in T(\delta_1,a)\} \cup \\
 & & \{(F(\delta_1) \wedge \varphi, \delta_2') \mid (\varphi,\delta_2') \in T(\delta_2,a)\} \\
T(\delta_1|\delta_2,a) & = & T(\delta_1,a) \cup T(\delta_2,a) \\
T(\delta^*,a) & = & \{(\neg F(\delta) \wedge \varphi, \delta';\delta^*) \mid (\varphi,\delta') \in T(\delta,a)\}
\end{array}
$$

$$
\begin{array}{lcl}
F(a) & = & False \\
F(\varphi?) & = & \varphi \\
F(\delta_1;\delta_2) & = & F(\delta_1) \wedge F(\delta_2) \\
F(\delta_1|\delta_2) & = & F(\delta_1) \vee F(\delta_2) \\
F(\delta^*) & = & True
\end{array}
$$

Program Graph

Now we can introduce the program graph

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

where

- Φ is a Boolean formula over tests and *Poss*
- $\mathcal{A}$ is the set of actions
- $\Phi \times \mathcal{A}$ is the alphabet
- $Q = \Gamma_{\delta_0}$ is the syntactic closure of δ_0
- $q_0 = \delta_0$ is the initial program
- $\sigma(q, \varphi, a, q')$ iff $(\varphi, q') \in T(q, a)$
- $\mathcal{L}(q) = F(q)$ indicates that the state q is assigned a label according to F

Program Graph

Theorem

The number of nodes in $\mathcal{G}$ is linear in the size of the program δ_0. The number of edges in $\mathcal{G}$ is polynomial in the size of δ_0.

Definition

We say that a program is situation determined if

$$\begin{aligned} &SituationDetermined(\delta, s) \doteq \forall s', \delta', \delta''. \\ &\quad Trans^*(\delta, s, \delta', s') \wedge Trans^*(\delta, s, \delta'', s') \supset \delta' = \delta'' \end{aligned}$$

Theorem

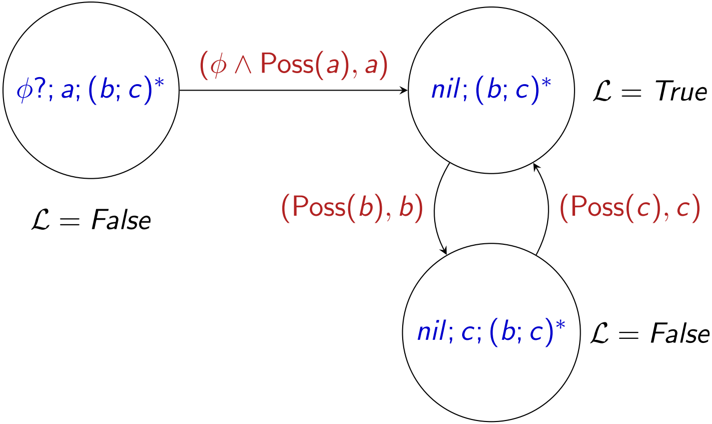
If δ_0 is situation determined, then

$$\sigma(q, \varphi_1, a, q_1') \wedge \sigma(q, \varphi_2, a, q_2') \wedge \varphi_1 \wedge \varphi_2 \supset q_1' = q_2'$$

and the characteristic graph becomes deterministic.

Example

Program: $\phi?; a; (b; c)^*$



FOND Domain

The DFA for a FOND Domain $\mathcal{D}$ is:

$$A_{\mathcal{D}} = \langle 2^{\mathcal{F} \cup \mathcal{A}}, 2^{\mathcal{F}} \cup \{s_{init}\}, s_{init}, \varrho, F \rangle$$

where:

- $2^{\mathcal{F} \cup \mathcal{A}}$ is the alphabet (actions $\mathcal{A}$ include dummy *start* action)
- $2^{\mathcal{F}} \cup \{s_{init}\}$ is the set of states
- s_{init} is the dummy initial state
- $F = 2^{\mathcal{F}}$ (all states of the domain are final)
- $\varrho(s, (a, s')) = s'$ with $a \in \alpha(s)$ and $\rho(s, a, s')$

Cross Product

Taking the cross product of δ_0 and the FOND domain $\mathcal{D}$ we get

$$A = \langle \mathcal{A} \times 2^{\mathcal{F}} \times \Gamma_{\delta_0}, \Gamma_{\delta_0} \times 2^{\mathcal{F}}, (\delta_0, s_0), Tr, Fin \rangle$$

where

- $\mathcal{A} \times 2^{\mathcal{F}} \times \Gamma_{\delta_0}$ is an alphabet
- $\Gamma_{\delta_0} \times 2^{\mathcal{F}}$ is a set of states
- (δ_0, s_0) is the initial state
- $Tr((\delta, s), a, s', \delta') = (\delta', s')$, where $\exists \varphi . \sigma(\delta, \varphi, a, \delta') \wedge s \models \varphi$ and $\rho(s, a, s')$, is the transition function
- $Fin = \{(\delta, s) \mid s \models F(\delta)\}$ is the set of final states

Cross Product

Theorem

If δ_0 is situation determined, then

$$Tr((\delta, s), a, s', \delta'_1) \wedge Tr((\delta, s), a, s', \delta'_2) \supset \delta'_1 = \delta'_2$$

If δ_0 is situation-determined, we can simplify the DFA into:

$$A = \langle \mathcal{A} \times 2^{\mathcal{F}}, \Gamma_{\delta_0} \times 2^{\mathcal{F}}, (\delta_0, s_0), Tr, Fin \rangle$$

where

- $\mathcal{A} \times 2^{\mathcal{F}}$ is an alphabet
- $Tr((\delta, s), a, s') = (\delta', s')$, where $\exists \varphi . \sigma(\delta, \varphi, a, \delta') \wedge s \models \varphi$ and $\rho(s, a, s')$
- $Fin = \{(\delta, s) \mid s \models F(\delta)\}$