Abstraction of Nondeterministic Situation Calculus Action Theories

Bita Banihashemi\textsuperscript{1}, Giuseppe De Giacomo\textsuperscript{2}, Yves Lespérance\textsuperscript{3}
\textsuperscript{1}Ronin Institute
\textsuperscript{2}University of Oxford
\textsuperscript{3}York University
bita@ronininstitute.org, giuseppe.degiacomo@cs.ox.ac.uk, lesperan@eecs.yorku.ca

Abstract

We develop a general framework for abstracting the behavior of an agent that operates in a nondeterministic domain, i.e., where the agent does not control the outcome of the nondeterministic actions, based on the nondeterministic situation calculus and the ConGolog programming language. We assume that we have both an abstract and a concrete nondeterministic basic action theory, and a refinement mapping which specifies how abstract actions, decomposed into agent actions and environment reactions, are implemented by concrete ConGolog programs. This new setting supports strategic reasoning and strategy synthesis, by allowing us to quantify separately on agent actions and environment reactions. We show that if the agent has a (strong FOND) plan/strategy to achieve a goal/complete a task at the abstract level, and it can always execute the nondeterministic abstract actions to completion at the concrete level, then there exists a refinement of it that is a (strong FOND) plan/strategy to achieve the refinement of the goal/task at the concrete level.

1 Introduction

When working in realistic dynamic domains, the use of abstraction has proven essential in many areas of artificial intelligence, for example, in improving the efficiency of planning (e.g., [Chen and Bercher, 2021]), explaining agents’ behavior (e.g., [Seegebarth et al., 2012]), and in reinforcement learning (e.g., [Sutton et al., 1999]).

Recently, [Banihashemi et al., 2017] (BDL17) proposed a formal account of agent abstraction based on the situation calculus [McCarthy and Hayes, 1969; Reiter, 2001] and the ConGolog agent programming language [De Giacomo et al., 2000]. They assume that one has a high-level/abstract action theory, a low-level/concrete action theory, both representing the agent’s behavior at different levels of detail, and a refinement mapping between the two. The refinement mapping specifies how each high-level action is implemented by a low-level ConGolog program and how each high-level fluent can be translated into a low-level state formula. This work defines notions of abstractions between such action theories in terms of the existence of a suitable bisimulation relation [Milner, 1971] between their respective models. Abstractions have many useful properties that ensure that one can reason about the agent’s actions (e.g., executability, projection, and planning) at the abstract level, and refine and concretely execute them at the low level. The framework can also be used to generate high-level explanations of low-level behavior.

This framework was formulated assuming a deterministic environment, as usual in the situation calculus. Hence, the only nondeterminism was coming from ConGolog programs and was angelic in nature, i.e., under the control of the agent [Levesque et al., 1997; De Giacomo et al., 2000].

However, many agents operate in nondeterministic environments where the agent does not fully control the outcome of her actions (e.g., flipping a coin where the outcome may be heads or tails). Recently, [De Giacomo and Lespérance, 2021] (DL21) proposed a simple and elegant situation calculus account of nondeterministic environments where they clearly distinguish between the nondeterminism associated with agent choices and that associated with environment choices, the first being angelic for the agent, and the second being devilish, i.e., not under the agent’s control. The presence of environment nondeterminism deeply influences reasoning about action: if the agent wants to complete a task, she cannot simply do standard logical reasoning (i.e., satisfiability/logical implication), but needs to do realizability/synthesis, i.e., devise a strategy that, in spite of the uncontrollable reactions of the environment, guarantees successful completion of the task [Church, 1963; Pnueli and Rosner, 1989; Abadi et al., 1989]. We see this, for example in planning, where in the classical deterministic setting the problem can be solved by heuristic search, while in fully observable nondeterministic (FOND) domains forms of adversarial (AND/OR) search are required [Cimatti et al., 1998; Haslum et al., 2019].

In this paper, we develop an account of agent abstraction in presence of nondeterministic environments based on (DL21). The refinement mapping between high-level fluents and low-level state formulas remains as in (BDL17). Instead we consider each high-level action, now nondeterministic, as being...
composed of an agent action and an (implicit) environment reaction. As a consequence, we map the agent action (without the environment reaction) into a low-level agent program that appropriately reflects the nondeterminism of the environment, and the complete high-level action, including both the agent action and the environment reaction, into a low-level system program that relates the high-level environment reaction to the low-level ones.

We show that the (BDL17) notion of m-bisimulation extends naturally to this new setting. This allows for exploiting abstraction when reasoning about executions as in (BDL17), in spite of the nondeterministic environment. But, notably, this new setting now supports strategic reasoning and strategy synthesis, by allowing us to quantify separately on agent actions and environment reactions. As a result, we can exploit abstraction by finding abstract strategies at the high-level and then refine them into concrete low-level ones. In particular, we show that if the agent has a (strong FOND) plan/strategy to achieve a goal/complete a task at the high level (i.e., no matter what the environment does), and she can always execute the nondeterministic high-level actions to completion at the low level (even if not controlling their outcome), then there exists a refinement of it that is a (strong FOND) plan/strategy to achieve the refinement of the goal/task at the low level.

2 Preliminaries

Situation calculus. The situation calculus is a well-known predicate logic language for representing and reasoning about dynamically changing worlds [McCarthy and Hayes, 1969; Reiter, 2001]. All changes to the world are the result of actions, which are terms in the language. A possible world history is represented by a term called a situation. The constant $S_0$ is used to denote the initial situation. Sequences of actions are built using the function symbol $do$, such that $do(a, s)$ denotes the successor situation resulting from performing action $a$ in situation $s$. Predicates and functions whose value varies from situation to situation are called fluents, and are denoted by symbols taking a situation term as their last argument (e.g., $Open(Door_1, s)$). The abbreviation $do([a_1, \ldots, a_n], s)$ stands for $do(\langle a_n, do(\langle a_{n-1}, \ldots, do(\langle a_1, s) \ldots)) \rangle)$; for an action sequence $\bar{a}$, we often write $do(\bar{a}, s)$ for $do([\bar{a}], s)$. In this language, a dynamic domain can be represented by a basic action theory (BAT), where successor state axioms represent the causal laws of the domain and provide a solution to the frame problem [Reiter, 2001]. A special predicate $Poss(a, s)$ is used to state that action $a$ is executable in situation $s$; the precondition axioms characterize this predicate. Abbreviation $Executable(s)$ means that every action performed in reaching situation $s$ was possible in the situation in which it occurred.

Nondeterministic situation calculus. A major limitation of the standard situation calculus and BATs is that atomic actions are deterministic. [De Giacomo and Lespérance, 2021] (DL21) propose a simple extension of the framework to handle nondeterministic actions while preserving the solution to the frame problem. For any primitive action by the agent in a nondeterministic domain, there can be a number of different outcomes. (DL21) takes the outcome as being determined by the agent’s action and the environment’s reaction to this action. This is modeled by having every action type/function $A(\bar{x}, e)$ take an additional environment reaction parameter $e$, ranging over a new sort Reaction of environment reactions. The agent cannot control the environment reaction, so it performs the reaction-suppressed version of the action $A(\bar{x})$ and the environment then selects a reaction $e$ to produce the complete action $A(\bar{x}, e)$. We call the reaction-suppressed version of the action $A(\bar{x})$ an agent action and the full version of the action $A(\bar{x}, e)$ a system action.

Nondeterministic basic action theories (NDBATs). These can be seen as a special kind of BAT, where every action function takes an environment reaction parameter, and moreover, for each agent action $A(\bar{x})$, we have:

1. have its agent action precondition denoted by: $Poss_{sys}(A(\bar{x}), s) \equiv \phi_A(\bar{x}, s)$;
2. have a reaction independence requirement, stating that the precondition for the agent action is independent of any environment reaction: $\forall e. Poss(A(\bar{x}, e), s) \supset Poss_{sys}(A(\bar{x}), s)$;
3. have a reaction existence requirement, stating that if the precondition of the agent action holds then there exists a reaction to it which makes the complete system action executable: $Poss_{sys}(A(\bar{x}), s) \supset \exists e. Poss(A(\bar{x}, e), s)$.

The above requirements must be entailed by the action theory for it to be an NDBAT.

A NDBAT $D$ is the union of the following disjoint sets: foundational, domain independent, axioms of the situation calculus ($\Sigma$) as in standard BATs, axioms describing the initial situation $D_{B0}$ as in standard BATs, unique name axioms for actions ($D_{una}$) as in standard BATs, successor state axioms (SSAs) describing how fluents change after system actions are performed ($D_{ss}$), and system action precondition axioms, one for each action type, stating when the complete system action can occur ($D_{pos}$); these are of the form: $Poss(A(\bar{x}, e), s) \equiv \phi_A(\bar{x}, e, s)$.

Example. Our running example is based on a triangle tireworld domain (see Fig. 1). The agent’s goal is to drive from location 11 to location 13. When driving from one location to the next, the possibility of a tire going flat exists. If there is a spare tire in the location where the car is at (indicated by circles in Fig. 1), the agent can use it to fix a flat.

NDBAT $D^{D^T}$ models the domain at the low level. The system action $drive(o, d, r)$ can be performed by the agent to drive from origin location $o$ to destination location $d$, and $r$ indicates environment’s reaction. This action is executable

![Figure 1: A 3-Level Triangle Tireworld](image-url)
when the agent action \( \text{drive}(o, d) \) is executable (i.e., when the agent is at \( o \), a road connects \( o \) to \( d \), and agent does not have a flat tire), and \( r \) can take on two values: \( \text{FlatTire} \) if the tire goes flat and \( \neg \text{FlatTire} \) otherwise. The fluent \( \text{At}_{LL}(l, s) \) indicates agent’s location, \( \text{Road}_{LL}(o, d, s) \) specifies the road connections in Fig. 1, \( \text{Flat}_{LL}(s) \) describes whether there is a flat tire, and \( \text{Visited}_{LL}(l, s) \) indicates if the location \( l \) has already been visited by the agent.

The system action \( \text{fixFlatTire}(l, r) \) fixes a flat tire, and the environment reaction \( r \) is \( \text{Success}_{LL}P \) (assumed for simplicity). It is executable if the fluent \( \text{Spare}_{LL}(l, s) \) holds, i.e., there is a spare tire. The action \( \text{wait}_{LL} \) can be performed by the agent to remain idle at a location.

\( D_f \) includes the following action precondition axioms

\[
\begin{align*}
\text{Poss}_o(\text{drive}(o, d), s) & \equiv \\
o \neq d \land \text{At}_{LL}(o, s) \land \text{Road}_{LL}(o, d, s) \land \neg \text{Flat}_{LL}(s) \\
\text{Poss}_o(\text{fixFlatTire}(l), s) & \equiv \\
\text{At}_{LL}(l, s) \land \text{Spare}_{LL}(l, s) \land \text{Flat}_{LL}(s) \\
\text{Poss}_o(\text{wait}_{LL}, s) & \equiv \text{TRUE} \\
\text{Poss}(\text{drive}(o, d, r), s) & \equiv \\
\neg \text{Flat}_{LL}(s) \land \neg \text{Flat}_{LL}(r) \\
\text{Poss}(\text{fixFlatTire}(l, r), s) & \equiv \\
\text{Poss}(\text{wait}_{LL}(r), s) & \equiv \\
\neg \text{Flat}_{LL}(s) \land r = \text{Success}_{LL}P \\
\end{align*}
\]

For the other fluents, we have SSAs specifying that they are unaffected by any action.

\( \hat{D}_f \) also contains the following initial state axioms:

\[
\begin{align*}
\text{At}_{LL}(l, do(a, s)) & \equiv \exists o, \text{r, a} = \text{drive}(o, l, r) \\
\text{Flat}_{LL}(do(a, s)) & \equiv \\
\text{At}_{LL}(l, s) \land \exists d, \text{r, a} \neq \text{drive}(l, d, r) \\
\text{Visited}_{LL}(l, do(a, s)) & \equiv \\
\exists o, \text{r, a} = \text{drive}(o, l, r) \lor \text{Visited}_{LL}(l, s) \\
\end{align*}
\]

For other fluents, we have SSAs specifying that they are unaffected by any action.

\( \hat{D}_f \) also contains the following initial state axioms:

\[
\begin{align*}
\text{Road}_{LL}(o, d, s_0) & \equiv \{ (11, 12), (11, 21), (12, 21), (12, 22), (12, 13), (13, 12), (22, 31), (21, 31), (12, 11), (21, 11), (21, 12), (12, 13), (13, 12), (22, 31), (31, 22), (31, 21) \} \\
\text{Spare}_{LL}(l, s_0) & \equiv \{ (21, 22, 31), \text{Visited}_{LL}(l, s_0) \} \equiv l = 11, \text{At}_{LL}(l, s_0) \equiv l = 11, \text{Dest}_{LL}(l, s_0) \equiv l = 13, \neg \text{Flat}_{LL}(s_0) \\
\end{align*}
\]

**FOND planning and synthesis in NDBATs.** We start with some definitions. A weak plan is one that achieves the goal when the environment “cooperates” and selects environment reactions to nondeterministic actions that make this happen. Formally, we say that a sequence of agent actions \( \hat{a} \) is a weak plan to achieve \( \text{Goal} \) if \( s' \in \text{Poss}_o(\hat{a}, s_0, s') \land \text{Goal}(s') \) holds, i.e., there exists an execution of \( \hat{a} \) that takes us from the initial situation \( s_0 \) to situation \( s' \) where the goal holds. \( \text{Poss}_o(\hat{a}, s, s') \) means that the system may reach situation \( s' \) when the agent executes the sequence of agent actions \( \hat{a} \) depending on environment reactions:

\[
\begin{align*}
\text{Poss}_o(\epsilon, s, s') & \equiv s' = s \text{ (where } \epsilon \text{ is the empty sequence of actions)} \\
\text{Poss}_o([A(x), \sigma], s, s') & \equiv \\
\exists \epsilon. \text{Poss}(A(x, e), s) \land \text{Poss}_o(\sigma, do(A(x, e), s), s')
\end{align*}
\]

A strong plan for the agent guarantees the achievement of a goal no matter how the environment reacts; it is a strategy for the agent to follow to ensure that the goal is achieved. (DL21) define a strategy as a function from situations to (instantiated) agent actions. That is, \( f(s) = A(t) \) denotes that the strategy \( f \) applied to situation \( s \) returns \( A(t) \) as the next action to do. The special agent action \( \text{stop} \) (with no effects and preconditions) may be returned when the strategy wishes to stop. Given a strategy, we can check whether it forces the goal to become true in spite of the environment reactions, i.e., is a strong plan to achieve the goal. Formally, we have \( \text{AgentCanForceBy}(\text{Goal}, s, f) \), i.e., the agent can force \( \text{Goal} \) to become true by following strategy \( f \) in \( s \):

\[
\text{AgentCanForceBy}(\text{Goal}, s, f) \equiv \forall P. [\ldots \lor P(s)]
\]

\[
\text{AgentCanForceBy}(\text{Goal}, s, f) \equiv \begin{cases} 
\text{success}_{LL}P & \text{if } f(s) = \text{stop} \land \text{Goal}(s) \lor P(s) \\
\exists A. \exists \epsilon. f(s) = A(t) \neq \text{stop} \land \text{Poss}(A(t), s) \land \\
\epsilon. (\text{Poss}(A(t), e), s) \lor P(\text{do}(A(t), e), s)) & \lor P(s) 
\end{cases}
\]

We say that \( \text{AgentCanForceBy}(\text{Goal}, s, f) \) holds iff there is a strategy \( f \) such that \( \text{AgentCanForceBy}(\text{Goal}, s, f) \) holds.

**Example cont.** Strategies use the \( \text{stop} \) action, hence we define a \( \text{stop}_{LL} \) action at the low level; this action always terminates with the \( \text{success}_{LL}P \) reaction. The strategy that guarantees reaching location 13 is defined as follows:

\[
D_f \models \text{AgentCanForceBy}(\text{At}_{LL}(13), s_0, f)
\]

\[
f(s) = \begin{cases} 
\text{stop}_{LL} & \text{if } \text{At}_{LL}(13, s) \\
\text{fixFlatTire}(l) & \text{if } \text{At}_{LL}(l, s) \land l \neq 13 \land \neg \text{Flat}_{LL}(s) \\
\text{do}(\text{drive}(o, d)) & \text{if } \text{At}_{LL}(o, s) \land o \neq 13 \land \neg \text{Flat}_{LL}(s) \\
\text{do}(\text{fixFlatTire}(l)) & \text{if } \text{Spare}_{LL}(l, d, s) \land \text{Road}_{LL}(o, d, s) \\
\text{success}_{LL}P & \text{otherwise}
\end{cases}
\]

That is, the agent should stop if she is already at location 13, otherwise she drives to a location she has not visited before and that has a spare tire; in case of a flat tire, action \( \text{fixFlatTire} \) is executed; in all other cases the agent waits.

**High-level programs and ConGolog.** To represent and reason about complex actions or processes obtained by suitably executing atomic actions, various so-called high-level programming languages have been defined. Here we concentrate on (a variant of) ConGolog [De Giacomo et al., 2000] that includes the following constructs:

\[
\delta ::= \alpha \mid \varphi \mid \delta_1; \delta_2 \mid \delta_1 \parallel \delta_2 \mid \pi.x.\delta \mid \delta^* \mid \delta_1 || \delta_2
\]

In the above, \( \alpha \) is an action term, possibly with parameters, and \( \varphi \) is a situation-suppressed formula, i.e., a formula with all situation arguments in fluents suppressed. As usual, we denote by \( \varphi[s] \) the formula obtained from \( \varphi \) by restoring the situation argument \( s \) into all fluents in \( \varphi \). The sequence of program \( \delta_1 \) followed by program \( \delta_2 \) is denoted by \( \delta_1; \delta_2 \). Program \( \delta_1 \parallel \delta_2 \) allows for the nondeterministic choice between programs \( \delta_1 \) and \( \delta_2 \), while \( \pi.x.\delta \) executes program \( \delta \) for some nondeterministic choice of a binding for object variable \( x \) (observe that such a choice is, in general, unbounded). \( \delta^* \) performs \( \delta \) zero or more times. Program \( \delta_1 || \delta_2 \) expresses...
the concurrent execution (interpreted as interleaving) of programs \( \delta_1 \) and \( \delta_2 \). The construct if \( \phi \) then \( \delta_1 \) else \( \delta_2 \) end if is defined as \([\phi?; \delta_1] \mid [\neg \phi?; \delta_2] \). We also use nil, an abbreviation for True?, to represent the empty program, i.e., when nothing remains to be performed.

Formally, the semantics of ConGolog is specified in terms of single-step transitions, using the following two predicates: (i) \( \text{Trans}(\delta, s, \delta', s') \), which holds if one step of program \( \delta \) in situation \( s \) may lead to situation \( s' \) with \( \delta' \) remaining to be executed; and (ii) \( \text{Final}(\delta, s) \), which holds if program \( \delta \) may legally terminate in situation \( s \). The definitions of \( \text{Trans} \) and \( \text{Final} \) we use are as in [De Giacomo et al., 2010], differently from [De Giacomo et al., 2000], the test construct \( \phi? \) does not yield any transition, but is final when satisfied.

Predicate \( \text{Do}(\delta, s, s') \) means that program \( \delta \), when executed starting in situation \( s \), has as a legal terminating situation \( s' \), and is defined as \( \text{Do}(\delta, s, s') \equiv \exists \delta'. \text{Trans}(\delta, s, \delta', s') \land \text{Final}(\delta', s') \) where \( \text{Trans} \) denotes the reflexive transitive closure of \( \text{Trans} \). We use \( \mathcal{C} \) to denote the axioms defining the ConGolog programming language.

For simplicity in this paper, we use a restricted class of ConGolog programs which are situation-determined (SD) [De Giacomo et al., 2012], i.e., for every sequence of actions, the remaining program is uniquely determined by the resulting situation:

\[
\text{SituationDetermined}(\delta, s) \equiv \forall s', \delta', s'' . \text{Trans}^*(\delta, s, \delta', s') \land \text{Trans}^*(\delta, s, \delta'', s') \supset \delta' = \delta''
\]

### ConGolog program execution in ND domains.
To specify agent behaviors in nondeterministic domains, (DL21) use ConGolog agent programs, which are composed as usual, but only involve agent (reaction-suppressed) atomic actions. The semantics for such agent programs is a variant of the one above where a transition for an atomic agent action may occur whenever there exists a reaction that can produce it.

### Strategic reasoning in executing programs in ND domains.
To represent the ability of the agent to execute an agent program in a ND domain (DL21) introduce \( \text{AgtCanForceBy}(\delta, s, f) \) as an adversarial version of \( \text{Do} \) in presence of environment reactions. This predicate states that strategy \( f \), a function from situations to agent actions (including the special action \( \text{stop} \)), executes SD ConGolog agent program \( \delta \) in situation \( s \) considering its nondeterminism angelic, as in the standard \( \text{Do} \), but also considering the nondeterminism of environment reactions devilish/ adversarial:

\[
\text{AgtCanForceBy}(\delta, s, f) \equiv \forall P \ldots \supset P(\delta, s) \]

where \( \ldots \) stands for:

\[
[(f(s) = \text{stop} \land \text{Final}(\delta, s)) \supset P(\delta, s)] \land
[\exists A . \exists f(s) = A(t) \neq \text{stop} \land
\exists e . \exists \delta' . \exists s' . \text{Trans}(\delta, s, \delta', \text{do}(A(t), e), s') \land
\forall e . \exists \delta' . \exists s' . \text{Trans}(\delta, s, \delta', \text{do}(A(t), e), s') \supset
\exists \delta'' . \exists s'' . \text{Trans}(\delta, s, \delta', \text{do}(A(t), e), s') \land
P(\delta'', s'') \supset P(\delta, s)]
\]

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

### Example cont.
Suppose we have a program \( \delta \ll (l) \) which lets the agent drive to any location, or fix a flat any number of times until a destination \( l \) is reached: \( \delta \ll (l) = \langle \pi \sigma . \text{drive}(a, d) \mid \pi \delta . \text{fixFlatTire}(d) \rangle^* . \text{At}_\ll (l) \).

Now suppose the agent has been assigned the task of going to location 13. We can show that she has a strategy for executing the task \( \delta \ll (13) \):

\[
D^*_{13} \models \text{AgtCanForceBy}(\delta \ll (13), S_0, f)
\]

Here, the strategy \( f \) to execute the task is same as \( f \) above.

### 3 Abstraction in Nondeterministic Domains
In this section, we show how we can extend the agent abstraction framework of (BDL17) to handle nondeterministic domains. As in (BDL17), we assume that there is a high-level/abstract (HL) action theory \( D_h \) and a low-level/concrete (LL) action theory \( D_l \) representing the agent’s possible behaviors at different levels of detail. In (BDL17), these are standard BATS; here, we assume that they are both NDBATs. \( D_h \) (resp. \( D_l \)) involves a finite set of primitive action types \( A_h \) (resp. \( A_l \)) and a finite set of primitive fluent predicates \( F_h \) (resp. \( F_l \)). The terms of sort Object are assumed be a countably infinite set \( N \) of standard names for which we have the unique name assumption and domain closure. We also assume that Reaction is a sub-sort of Object. Also, \( D_h \) and \( D_l \) are assumed to share no domain specific symbols except for the set of standard names for objects \( N \). For simplicity and w.l.o.g., it is assumed that there are no functions other than constants and no non-fluent predicates.

### Refinement mapping
As in (BDL17), we assume that one relates the HL and LL theories by defining a refinement mapping that specifies how HL atomic actions are implemented at the low level and how HL fluents can be translated into LL state formulas. In deterministic domains, one can simply map HL atomic actions to a LL program that the agent uses to implement the action. In nondeterministic domains however, we additionally need to specify what HL reaction the environment performs in each LL refinement of the HL action.

### Definition 1 (NDBAT Refinement Mapping)
A NDBAT refinement mapping \( m \) is a triple \( \langle m_a, m_s, m_f \rangle \) where \( m_a \) associates each HL primitive action type \( A \in A_h \) to a SD ConGolog agent program \( \delta_{\alpha}^A \) defined over the LL theory that implements the agent (i.e., reaction-suppressed) action, i.e., \( m_a(A(\bar{x})) = \delta_{\alpha}(\bar{x}) \), \( m_s \) associates each \( A \) to a SD ConGolog system program \( \delta^S_{\alpha} \) defined over the LL theory that implements the system action, i.e., \( m_s(A(\bar{x}, e)) = \delta^S_{\alpha}(\bar{x}, e) \), thus specifying when the HL reaction occurs (system programs are interpreted using the standard transition semantics), and (as in (BDL17)) \( m_f \) maps each situation-suppressed HL fluent \( F(\bar{x}) \in F_h \) to a situation-suppressed formula \( \phi(\bar{x}) \) defined over the LL theory that characterizes the concrete conditions under which \( F(\bar{x}) \) holds in a situation, i.e., \( m_f(\bar{x}) = \phi(\bar{x}) \).

We can extend a mapping to a sequence of agent actions in the obvious way, i.e., \( m_a(\alpha_1, \ldots, \alpha_n) = m_a(\alpha_1) \ldots m_a(\alpha_n) \) for \( n \geq 1 \) and \( m_a(\bar{e}) = \text{nil} \), and similarly for sequences of system actions. Note that \( m_a(\bar{\alpha}) \) is well defined even if the action parameters are not ground; but
the action functions must be given. We also extend the notation so that \( m_f(\phi) \) stands for the result of substituting every fluent \( F(x) \) in situation-suppressed formula \( \phi \) by \( m_f(F(x)) \).

Agent actions and system actions must be mapped in a consistent way. To ensure this, we require the following:

**Constraint 2. (Proper Refinement Mapping)**
For every high-level system action sequence \( \alpha \) and every high-level action \( A \), we have that:

\[
\forall x, \exists s : (Do(m_A(\alpha)), S_0, s) \supset \forall x, \exists s' : (Do(m_A(A(x), e)), s, s') \equiv \exists \alpha, Do(m_A(A(x, e)), S_0, s, s')
\]

This ensures that (1) for every situation \( s' \) that can be reached from \( s \) by executing a refinement of the HL agent action \( A(x, e) \), there is a HL reaction \( e \) that generates it, and (2) that for every situation \( s' \) that can be reached from \( s \) by executing a refinement of the HL system action \( A(x, e) \), there is a refinement of the HL agent action that generates it (in the above, \( s' \) is any situation reached by executing a sequence of HL system actions). Note that (1) essentially states that the reaction existence requirement holds at the low level for refinements of HL actions and (2) amounts to having the reaction independence requirement hold. We say that an NDBAT refinement mapping \( m \) is proper wrt low-level NDBAT \( D_l \) if this constraint holds.

**Example cont.** NDBAT \( D^H_l \) abstracts over driving and fixing potential flat tires. Action \( driveAndTryFix(o, d, r) \) can be performed to drive from origin \( o \) to destination \( d \), and fix a flat if one occurs and a spare is available. This action is executable when \( driveAndTryFix(o, d, r) \) is executable, i.e., when the agent is at \( o \), a road connects \( o \) to \( d \), and agent does not have a flat tire; the environment reaction \( r \) may take on the following values: \( DriveNoFlat \), if the tire does not go flat, \( DriveFlat \) if the tire goes flat and it is not possible to fix it, or \( DriveFlatFix \) if the tire went flat and the flat was fixed. Action \( wait_{HL} \) lets the agent remain idle at a location. \( D^H_l \) includes the following system and agent action precondition axioms:

\[
Poss_a(driveAndTryFix(o, d, r), s) \equiv o \neq n \land \forall A_{HL}(o, s) \land Road_{HL}(o, d, s) \land \neg Flat_{HL}(s)
\]

\[
Poss_s(driveAndTryFix(o, d, r), s) \equiv Poss_a(driveAndTryFix(o, d, r), s) \land \neg DriveNoFlat \lor \neg DriveFlat \lor DriveFlatFix
\]

\[
Poss(wait_{HL}(r), s) \equiv Poss_a(wait_{HL}(r), s) \land r = Success_{HL}
\]

\[
D^H_l \text{ also includes the following SSAs:}
\]

\[
\begin{align*}
At_{HL}(l, do(a, s)) & \equiv \exists a, r.a = driveAndTryFix(o, l, r) \lor At_{HL}(l, s) \land \forall d, r.a \neq driveAndTryFix(l, d, r) \\
Flat_{HL}(do(a, s)) & \equiv \exists a, \forall d, a.d = driveAndTryFix(o, d, DriveFlat) \lor Flat_{HL}(s) \\
Visited_{HL}(l, do(a, s)) & \equiv \exists a, r.a = driveAndTryFix(o, l, r) \lor Visited_{HL}(l, s)
\end{align*}
\]

For the other fluents, we have SSAs specifying that they are unaffected by any action.

The initial state axioms of \( D^H_l \) are same as \( D^H_l \), with high-level fluents having a distinct name (with HL suffix); e.g., \( Road_{HL} \) is axiomatized exactly as \( Road_{LL} \).

We specify the relationship between the HL and LL NDBATs through the following refinement mapping \( m^{HL} \):

\[
m^{HL}_F(driveAndTryFix(o, d)) =
\begin{cases}
\text{drive}(o, d) & \text{if } o \neq Flat_{LL} \text{ then nil else} \\
\text{fixFlatTire}(d) & \text{endIf}
\end{cases}
\]

\[
m^{HL}_s(wait_{HL}) = wait_{LL}
\]

\[
m^{HL}_F(driveAndTryFix(o, d, r)) =
\begin{cases}
\pi_{s}, \text{drive}(o, d, r) & \text{if } o \neq Flat_{LL} \text{ then } r_{h} = \text{DriveFlat} \text{ else} \\
\text{fixFlatTire}(d, Success_{LL}); r_{h} = \text{DriveFlatFix} & \text{endIf}
\end{cases}
\]

Thus, the HL agent action \( driveAndTryFix(o, d) \) is implemented by an LL program where the agent first performs \( drive(o, d) \), and depending on whether the tire has gone flat and a spare exists at location \( d \), fixes the tire or does nothing. Fluents \( At_{HL}(l) \), \( Spare_{HL}(l) \), \( Road_{HL}(o, d) \), \( Dest_{HL}(l) \), and \( Visited_{HL}(l) \) are mapped to their low-level counterparts similar to \( Flat_{LL} \). We can show that: ²

**Proposition 3.** NDBAT refinement mapping \( m^{HL} \) is proper wrt \( D^H_l \).

**m-Bisimulation.** To relate the HL and LL models/theories, (BDL17) define a variant of bisimulation [Milner, 1971; Milner, 1989]. The base condition for the bisimulation is:

\[
M_h, [v/s_{s_{h}}] = F(x, s) \text{ if } M_l, [v/s_{s_{l}}] = m(F(x))[s]
\]

for every high-level primitive fluent \( F(x) \) in \( F_h \) and every variable assignment \( v \).

If \( s_{h} \sim_{m, M_{h}} s_{l} \), then \( s_{h} \) and \( s_{l} \) evaluate all HL fluents the same.

**Definition 4 (m-isomorphic situations).** Let \( M_{h} \) be model of the HL theory \( D_{h} \), and \( M_{l} \) a model of the LL theory \( D_{l} \cup C \). We say that situation \( s_{h} \) in \( M_{h} \) is m-isomorphic to situation \( s_{l} \) in \( M_{l} \), written \( s_{h} \sim_{m, M_{h}} M_{l} s_{l} \), if and only if:

\[
M_{h}, [v/s_{s_{h}}] = F(x, s) \text{ iff } M_{l}, [v/s_{s_{l}}] = m(F(x))[s]
\]

For proofs, see [Banihashemi et al., 2023]

²As usual, \( M, v \models \phi \) means that model \( M \) and assignment \( v \) satisfy formula \( \phi \) (where \( \phi \) may contain free variables that are interpreted by \( v \); also \( v[x/e] \) represents the variable assignment that is just like \( v \) but assigns variable \( x \) to entity \( e \).

3116
\(s'_h\) such that \(M_h, v[s/s_h, s'/s'_h] \models \text{Poss}(A(x), s) \land s' = \text{Do}(A(x), s)\) and \(s'_h, s'_h' \in B\). \(M_h\) is \(m\)-bisimilar to \(M_h\), written \(M_h \sim_m M_h\), if and only if there exists an \(m\)-bisimulation relation \(B\) between \(M_h\) and \(M_h\) such that \((S^{M_h}_0, s^{M_h}_0) \in B\).

The definition of \(m\)-bisimulation can remain as in (BDL17) where conditions (ii) and (iii) are applied to high-level system primitive actions and their mapping \(m\).

The definition of \(m\)-bisimulation ensures that performing a \(H\) system action results in \(m\)-bisimilar situations. We can show that with the restriction to proper mappings, this automatically carries over to \(H\) agent actions (so there is no need to change the definition of \(m\)-bisimulation to get this):

**Theorem 6.** Suppose that \(M_h \sim_m M_h\), where \(M_h \models D_h\), \(M_i \models D_i \cup C\), \(D_h\) and \(D_i\) are both NDBATs, and \(m\) is proper wrt \(D_i\). Then for any \(H\) system action sequence \(\alpha\) and any HL primitive action \(\lambda\), we have that:

1. If there exist \(s_h\) and \(s'_h\) such that \(M_h, v[s/s_h, s'/s'_h] \models \text{Do}(\alpha, S_0, s) \land \text{Do}(\alpha, A(x), s, s', s'_h)\), then there exist \(s_i\) and \(s'_i\) such that \(M_i, v[s/s_i, s'/s'_i] \models \text{Do}(m_s(\alpha), S_0, s) \land \text{Do}(m_s(A(x)), s, s', s'_i)\), \(s_h \sim_m s_i\), and \(s'_h \sim_m s'_i\).

2. If there exist \(s_i\) and \(s'_i\) such that \(M_i, v[s/s_i, s'/s'_i] \models \text{Do}(m_s(\alpha), S_0, s) \land \text{Do}(m_s(A(x)), s, s', s'_i)\), then there exist \(s_h\) and \(s'_h\) such that \(M_h, v[s/s_h, s'/s'_h] \models \text{Do}(\alpha, S_0, s) \land \text{Do}(\alpha, A(x), s, s', s'_h)\), \(s_h \sim_m s_i\), and \(s'_h \sim_m s'_i\).

(BDL17) use \(m\)-bisimulation to define notions of sound/complete abstraction between a high-level action theory and a low-level one: \(D_h\) is a sound abstraction of \(D_i\) relative to refinement mapping \(m\) if and only if, for all models \(M_i\) of \(D_i \cup C\), there exists a model \(M_h\) of \(D_h\) such that \(M_h \sim_m M_i\). With a sound abstraction, whenever the high-level theory entails that a sequence of actions is executable and achieves a certain condition, then the low level must also entail that there exists an executable refinement of the sequence such that the “translated” condition holds afterwards. Moreover, whenever the low level considers the executability of a refinement of a high-level action is satisfiable, then the high level does also. A dual notion is also defined: \(D_h\) is a complete abstraction of \(D_i\) relative to refinement mapping \(m\) if and only if, for all models \(M_h\) of \(D_h\), there exists a model \(M_i\) of \(D_i \cup C\) such that \(M_i \sim_m M_h\).

The notion of \(m\)-bisimulation provides the semantic underpinning of these notions of abstraction. (BDL17) also prove the following results that identify a set of properties that are necessary and sufficient to have a sound/complete abstraction (here we adjust the notation to use system actions) and can be used to verify that one has a sound/complete abstraction, e.g., by model checking or theorem proving techniques:

**Theorem 7 (BDL17).** \(D^h_{m1} \cup D^h_{m2} \cup D^h_{m3} \models m_f(\phi)\), for all \(\phi \in D^h_{m1}\).

**Theorem 7 (BDL17).** \(D^h \models \forall s. \text{Do}(m_s(\alpha), S_0, s) \land \text{Do}(m_s(A(x)), s, s', s'_i)\)

**Theorem 7 (BDL17).** \(D^h \models \forall s. \text{Do}(m_s(\alpha), S_0, s) \land m_f(\phi)[s']\) if and only if \(M_h, v \models \text{Execute}(\text{Do}(\alpha, S_0)) \land \phi[do(\alpha, S_0)]\).

**Theorem 7 (BDL17).** \(D^h \models \forall s. \text{Do}(m_s(\alpha), S_0, s) \land m_f(\phi)[s']\) if and only if \(M_h, v \models \text{Execute}(\text{Do}(\alpha, S_0)) \land \phi[do(\alpha, S_0)]\).

**Theorem 7 (BDL17).** \(D^h \models \forall s. \text{Do}(m_s(\alpha), S_0, s) \land m_f(\phi)[s']\) if and only if \(M_h, v \models \text{Execute}(\text{Do}(\alpha, S_0)) \land \phi[do(\alpha, S_0)]\).

**Theorem 7 (BDL17).** \(D^h \models \forall s. \text{Do}(m_s(\alpha), S_0, s) \land m_f(\phi)[s']\) if and only if \(M_h, v \models \text{Execute}(\text{Do}(\alpha, S_0)) \land \phi[do(\alpha, S_0)]\).

**Theorem 7 (BDL17).** \(D^h \models \forall s. \text{Do}(m_s(\alpha), S_0, s) \land m_f(\phi)[s']\) if and only if \(M_h, v \models \text{Execute}(\text{Do}(\alpha, S_0)) \land \phi[do(\alpha, S_0)]\).

**Theorem 7 (BDL17).** \(D^h \models \forall s. \text{Do}(m_s(\alpha), S_0, s) \land m_f(\phi)[s']\) if and only if \(M_h, v \models \text{Execute}(\text{Do}(\alpha, S_0)) \land \phi[do(\alpha, S_0)]\).
This means that if the agent has a weak plan to achieve a goal at the high level, then there exists a refinement of it that is a weak plan to achieve the mapped goal at the low level.

**Example cont.** At the high level, a weak plan to achieve the goal of getting to location 13 is to first drive to 12 and then to 13:

\[
D^l_{12} \models \exists s'. D_{oag}(\bar{a}, S_0, s') \land A_{HL}(13, s')
\]

where 
\[
\bar{a} = driveAndTryFix(11, 12); driveAndTryFix(12, 13)
\]

This plan works provided no flat occurs when driving to 12, as there is no spare there. By Th. 12, there exists a weak plan to achieve a goal at the low level that refines this high-level plan:

\[
D^l_{13} \models \exists s'. D_{oag}(m_a(\bar{a}), S_0, s') \land A_{HL}(13, s') \land D_{oag}([drive(11, 12); drive(12, 13)], S_0, s')
\]

5 Results about Strategic Reasoning

Let us now discuss how NDBAT abstractions can be used in FOND domains to synthesize strategies to fulfill reachability/achievement goals as well as temporally extended goals/tasks. First of all, we need to consider how much strategic reasoning the agent needs to do to execute a high-level atomic action at the low level. A given HL agent action \(A(\bar{x})\) is mapped to a LL agent program \(m_a(A(\bar{x}))\) that implements it. As we have seen, in \(m\)-bisimilar models, Constraint 2 ensures that if \(A(\bar{x})\) is executable at the HL, then there exists a terminating execution of \(m_a(A(\bar{x}))\) at the LL (this holds for HL system actions as well). But this does not mean that all executions of \(m_a(A(\bar{x}))\) terminate, as some may block or diverge, due to either agent or environment choices. In general, the agent must do strategic reasoning to ensure that the execution of \(m_a(A(\bar{x}))\) terminates (and the environment may need to cooperate as well). But we can impose further constraints on the mapping of HL actions to avoid this or ensure that a strategy exists. Note that ensuring that the execution of \(m_a(A(\bar{x}))\) terminates does not mean that the agent controls the action’s outcome (e.g., not having a flat); the outcome is still determined by the environment reactions.

**Constraints on HL action implementation.** First, we may want to require that the mapping of HL actions is such that the implementation program always terminates and no LL strategic reasoning is required to ensure termination. To do this, we first define:

\[
InevTerminates(\delta, s) \equiv \forall P.[\ldots \supset P(\delta, s)]
\]

where \(\ldots\) stands for

\[
[(\text{Final}(\delta, s)) \lor P(\delta, s)] \land \exists s'.3\delta'.Trans(\delta, s, s', \delta') \land \forall s'.3\delta'.(Trans(\delta, s, s', \delta') \lor P(\delta', s')) \lor P(\delta, s])
\]

\(InevTerminates(\delta, s)\) means that program \(\delta\) executed starting in situation \(s\) inevitably terminates, i.e., all its executions eventually terminate.

Then we can ensure that for any HL agent action that is possibly executable at the low level, all executions terminate (i.e., they never block or diverge) by requiring the following:

**Constraint 13.** (HL actions Inevitably Terminate)

For every high-level system action sequence \(\bar{a}\) and every high-level action \(A\), we have that:

\[
D^l_1 \cup C \models \forall s_i. \left(D_{oag}(m_a(\bar{a}), S_0, s) \supset \right.

\[
\forall x_i. \left(\exists s'. D_{oag}(m_a(A(\bar{x})), s, s') \lor \right.

\[
\text{InevTerminates}(m_a(A(\bar{x})), s)\right).
\]

**Proposition 14.** NDBAT \(D^l_1\) and mapping \(m^l\) satisfy Constraint 13.

If we impose Constraint 13, then the agent that executes the program that implements the HL action can be a dumb executor. But this may seem too restrictive. An alternative is to impose the weaker requirement that for any HL agent action that is possibly executable at the LL, the agent has a strategy to execute it to termination no matter how the environment reacts (even if not controlling its outcome). Formally:

**Constraint 15.** (Agt Can Always Execute HL actions)

For every high-level system action sequence \(\bar{a}\) and every high-level action \(A\), we have that:

\[
D^l_1 \cup C \models \forall s_i. \left(D_{oag}(m_a(\bar{a}), S_0, s) \supset \right.

\[
\forall x_i. \left(\exists s'. D_{oag}(m_a(A(\bar{x})), s, s') \lor \right.

\[
AgtCanForce(m_a(A(\bar{x}))), s)\right).
\]

Note that if we only impose Constraint 15, then there is no guarantee that when a HL system action \(A(\bar{x}, e)\) is possibly executable, the environment can actually ensure that reaction \(e\) occurs when the agent executes \(m_a(A(\bar{x}))\). We can define an additional constraint that ensures this; see [Banihashemi et al., 2023] for details.

Planning for achievement goals. Returning to planning, we can now show that if Constraint 15 holds (i.e., the agent always knows how to execute high-level primitive actions at the low level, even if not controlling their outcome) and the agent has a strong plan to achieve a goal at the high level, then there exists a refinement of the high-level plan that is a strong plan to achieve the refinement of the goal at the low level:

**Theorem 16.** If \(M_h \sim_m, M_l\) and Constraint 15 holds, then for any high-level system action sequence \(\bar{a}\) and any high-level situation-suppressed formula \(\phi\), we have that:

\[
if M_h, v \models \text{Executable}(do(\bar{a}, S_0)) \land \right.

\[
AgtCanForce(\phi, do(\bar{a}, S_0))
\]

\[
then M_l, v \models \exists s_i. D_{oag}(m_a(\bar{a}), S_0, s) \land

\[
\forall s_i. D_{oag}(m_a(\bar{a}), S_0, s) \supset \right.

\[
AgtCanForce(m_f(\phi), s)
\]

**Proof sketch.** The proof is by induction on the length of the HL strategy \(f_h\) at \(do(\bar{a}, S_0)\), where the length of a strategy is the length of its longest branch, i.e.,

\[
\text{length}(f, s) = 0 \text{ if } f(s) = \text{stop} \text{ and } \text{length}(f, s) = \max_{D_{oag}(f(s), s')} \text{length}(f', s') \text{ otherwise.}
\]

Assume the antecedent. It follows that there is a strategy \(f_h\) such that \(AgtCanForceBy(\phi, s_h, f_h)\), where \(s_h = do(\bar{a}, S_0)\). By proof of Theorem 11 we have that at the LL \(D_{oag}(m_a(\bar{a}), S_0, s)\) and for all such \(s_i\) we have that \(s_i \sim_m M^i\) \(s_i\). Take an arbitrary such \(s_i\). We show that there exists a LL strategy \(f_i\) such that \(AgtCanForceBy(m_f(\phi), s_i, f_i)\).

Base case, when \(\text{length}(f_h, s_h) = 0\), and \(f_h(s_h) = \ldots\)
At the HL, by the definition of HL strategy that there exists an high-level action that $AgtCanForceBy(m_f(\phi), s_i, f_i)$ for any strategy $f_i$ such that $f_i(s_i) = \text{stop}$. 

Inductive case: Assume the result holds for any HL strategy $f_h$ of length at most $k$. We show that it must also hold for a HL strategy $f_h$ of length $k+1$. At the HL, by the definition of $AgtCanForceBy$, we have that there exists a high-level action $A(\bar{t})$ such that $f_h(s_h) = A(\bar{t}) \land \exists s'_h, D_{\text{HL}}(A(\bar{t}), s_h, s'_h) \land \forall s'_h, D_{\text{HL}}(A(\bar{t}), s_h, s'_h) \supset AgtCanForceBy(\phi, s'_h, f_h)$. By Constraint 15, there exists a LL strategy $g_i$ such that $AgtCanForceBy(m_a(\phi), s_i, g_i)$. For any $s'_h$ such that $D_{\text{HL}}(g_i, s_i, s'_i)$, it follows by Constraint 2 that there exists $s'_h$ such that $s'_h \sim_s M_s, M_h \sim_s s'_i$. The length of $f_h$ at $s'_h$ is at most $k$. Thus in the induction hypothesis, there exists a LL strategy $f'_i$ such that $AgtCanForceBy(m_f(\phi), s_i, f'_i)$ for $f_i = f'_i \circ g_i$. 

Example cont. As strategies use the stop action, we first define a stop$_{\text{HL}}$ action at the high level which is mapped to the stop$_{\text{LL}}$ action at the concrete level; stop$_{\text{HL}}$ always terminates with the success$_{\text{HS}}$ reaction. By Proposition 14, Constraint 13 (and thus also the weaker Constraint 15) is satisfied for our example domain.

We can show that the agent has a strong plan to achieve the goal of being at location 13 at the high level (df abbreviates driveAndTryFix): 

$$D_{\text{HL}}^t \models AgtCanForceBy(At_{\text{HL}}(13), S_0, f_h)$$

$$f_h(s) \triangleq \begin{cases} 
\text{stop}_{\text{HL}} & \text{if } At_{\text{HL}}(13, s) \\
\text{df}(a, d) & \text{if } At_{\text{HL}}(a, s) \land a \neq 13 \land \text{Road}_{\text{HL}}(a, d, s) \\
\& \sim_{\text{Visited}}_{\text{HL}}(d, s) \land \text{Spare}_{\text{HL}}(d, s) & \text{wait}_{\text{HL}} \\
\text{otherwise} 
\end{cases}$$

The agent’s high-level strategy $f_h$ is to stop if she is already at location 13, otherwise to drive to a location which has not been visited previously and has a spare tire and fix a flat if one occurs; in all other cases she waits.

By Th. 16 there exists a strategy at the low level that is refinement of the high-level strategy; this strategy is the same as strategy $f_i$ in Section 2. It is easy to show that $f_i$ is a refinement of the high-level strategy $f_h$.

Planning for temporally extended goals/tasks. We can also show a similar result to the previous theorem for high-level programs/tasks/temporally extended goals: if Constraint 15 holds and the agent has a strategy to successfully execute an agent program (without concurrency) at the high level, then the agent also has a strategy to successfully execute some refinement of it at the low level:

**Theorem 17.** If $M_h \sim_m M_l$ and Constraint 15 holds, then for any high-level system action sequence $\delta$ and any SD ConGolog high-level agent program $\delta$ without the concurrent composition construct, we have that:

$$(M_h, v \models \text{Executable}(do(\delta, S_0))) \land AgtCanForceBy(\delta, do(\delta, S_0))$$

then $M_l, v \models \exists s. Do(m_a(\delta), S_0, s) \land \forall s. Do(m_a(\delta), S_0, s) \supset AgtCanForceBy(m_a(\delta), s)$.

Here, we apply the mapping to a high-level agent program $\delta$ without concurrency to produce a low level agent program $m_a(\delta)$; this can be defined in the obvious way, using $m_a$ to map atomic actions and $m_f$ for tests as usual, and by mapping the components and composing the result for other constructs, e.g., $m_a(\delta_1|\delta_2) = m_a(\delta_1)|m_a(\delta_2)$ (we discuss concurrency in [Banihashemi et al., 2023]).

**Proof sketch.** The proof is by induction on the length of the high-level strategy $f_h$ for executing $\delta$ at $do(\delta, S_0)$ and is similar to that of the above theorem. Note that since $\delta$ is SD, does not involve concurrency, and the high-level atomic actions are all mapped to SD programs, $m_a(\delta)$ must also be SD (we execute a high-level action using the appropriate strategy essentially as a procedure call and the high-level program continues after the call has completed).

**Example cont.** Consider the high-level program $\delta_{\text{HL}}^H(13) = (\pi_0, \text{driveAndTryFix}(a, d))$; $At_{\text{HL}}(13)$? which goes to location $l$ by repeatedly picking adjacent locations and doing $\text{driveAndTryFix}(a, d)$ until the agent is at $l$. We can show that the agent has a strategy to successfully execute this program to get to location 13:

$$D_{\text{HL}}^t \models AgtCanForceBy(\delta_{\text{HL}}^H(13), S_0, f_h)$$

Here, the high-level strategy $f_h$ to execute the program is same as the one we saw above. It can also be shown that the low-level strategy $f_i$ seen earlier is a refinement of $f_h$ and can be used to to execute $m_a(\delta_{\text{HL}}^H(13))$. To further illustrate the use of high-level programs, notice that we could give the agent the program $\delta_{\text{HL}}^H(31); \delta_{\text{HL}}^H(13)$ to execute. This is a more complex task as she must first go to location 31 and then to 13; but it is easier to to find a strategy to do it.

In [Banihashemi et al., 2023], we show how we can handle more complex domains where reactions represent (sets of) exogenous actions.

6 Monitoring and Explanation

Being able to monitor what a low-level agent is doing and describe it in abstract terms (e.g., to a client or manager) is particularly important in the context of Explainable AI. (BDL17) show how a sound abstraction wrt a mapping $m$ can be used to do this when certain conditions hold. They define an inverse mapping $m^{-1}$ that takes a sequence of low-level actions $\delta$ and returns the sequence of high-level actions $\delta$ that it refines in a given model. In [Banihashemi et al., 2023], we show that we can easily adapt this approach for NDBAT abstractions.

**Example cont.** Consider a variant of our running example where, if the agent has a flat and there is no spare tire in a location (i.e., at locations 11, 12 and 13), she can either buy one, or she can use a roadside assistance service to get one provided the location is within the covered area. We can have HL actions for these with the mapping:

$$m_{\text{HL}}^{t+}(\text{buyAndFix}(l, r)) = \pi_r.(\text{order}(l, r), \text{pay}(l, r), \text{fixFlatTire}(l, r))$$

$$m_{\text{HL}}^{t+}(\text{serviceAndFix}(l, r)) = \pi_r.(\text{callService}_{\text{HL}}(l, r), \text{fixFlatTire}(l, r), \text{pay}(l, r), \text{buyAndFix}(l, r), \text{fixFlatTire}(l, r))$$

$\text{success}_{\text{Buy}} ?$
Suppose that at the LL, the agent knows that both locations 12 and 13 fall under the roadside service coverage, while at the HL, the agent only knows this for 13 (we have imperfect information). We can define NDBATs for this and show \( \mathcal{D}_{ht}^{tt+} \) is a sound abstraction of \( \mathcal{D}_{ht}^{tt} \) [Banihashemi et al., 2023].

Now suppose that at the LL, the sequence of system actions \( \bar{a} = \text{drive}(11, 21, \text{NoFlatTire}); \text{drive}(21, 12, \text{FlatTire}) \) has occurred. In location 12, no spare tire is available, and if the agent wants to move on with her journey, she needs to either call for service or order a new tire. The inverse mapping tells us that HL system action sequence \( \bar{a} = \text{driveAndTryFix}(11, 21, \text{DrvNoFlat}); \text{driveAndTryFix}(21, 12, \text{DrvFlat}) \) has occurred. Since \( \mathcal{D}_{ht}^{tt+} \models \text{At}_{HL}(12, \text{do}(\bar{a}, S_0)) \), we can conclude at the HL that the agent is now at location 12. Since \( \mathcal{D}_{ht}^{tt+} \cup \{ \text{Poss}(\text{serviceAndFix}(12, \text{Succ}_{H\text{Serv}}), \text{do}(\bar{a}, S_0)) \} \) is satisfiable, we can also conclude that \( \text{serviceAndFix}(12, \text{Succ}_{H\text{Serv}}) \) might occur next, and similarly for \( \text{buyAndFix}(12, \text{Succ}_{H\text{Buy}}) \); in fact, we know at the HL that the latter is executable: \( \mathcal{D}_{ht}^{tt} \models \text{Poss}(\text{buyAndFix}(12, \text{Succ}_{H\text{Buy}}), \text{do}(\bar{a}, S_0)) \).

7 Discussion
There has been much previous work on fully/partially observable nondeterministic planning, e.g., [Gallah et al., 2004; Cimatti et al., 2003; Muise et al., 2014], strategy logics such as ATL [Alur et al., 2002], and strategic reasoning [Xiong and Liu, 2016b; Xiong and Liu, 2016a]. But such work assumes that the domain is represented at a single level of abstraction.

In planning, approaches such as hierarchical task networks (HTNs) [Erol et al., 1994] support abstract tasks that facilitate search. [Kuter et al., 2009] propose an algorithm which combines HTN-based search-control strategies with Binary Decision Diagram-based state representations for planning in nondeterministic domains. [Chen and Bercher, 2021] extend HTN planning with nondeterministic primitive tasks. [Bonet et al., 2017] investigate nondeterministic abstractions for generalized planning, where one looks for a (typically iterative) plan that solves a whole class of related planning problems. [Cui et al., 2021] apply the abstraction framework of (BDL17) to generalized planning. They use Golog programs to represent nondeterministic actions, and our results should make it possible to simplify their framework.

[Banihashemi et al., 2018] generalized the approach of (BDL17) to deal with agents that may acquire new knowledge during execution, and formalized (piecewise) hierarchical refinements of an agent’s ability (i.e., strategy) to achieve a goal. However, this work does not consider nondeterministic actions and is based on a more complex framework involving online executions.

Abstraction is important for efficient reasoning and explainability. This paper presents foundational results, i.e., a generic framework for abstraction in nondeterministic domains, which can be used for many reasoning tasks, such as planning, execution monitoring, and explanation. One potential practical application area is smart manufacturing. For instance, [De Giacomo et al., 2022] present a “manufacturing as a service” framework where facilities (made up of concrete resources) bid to produce products, given an abstract product recipe, which is based on an abstract information model in the cloud. Our work shows how one can develop an abstract domain model where planning for arbitrary high-level goals is easier and where we have guarantees that a high-level plan can be refined into a low-level one. It also allows one to monitor the low-level system by generating high-level descriptions of low-level system executions from which one can reason at the high level about what might happen next. Note that often manufacturing processes depend on the data and objects (parts) they produce and consume; to formalize this aspect, the situation calculus which provides a first-order representation of the state of the processes can indeed be used. Another potential practical application area is business process management, to support execution monitoring, explanation, and failure handling, see e.g., [Marrella et al., 2017].

As discussed in Section 3, (BDL17) identifies a set of properties that are necessary and sufficient to have a sound/complete abstraction wrt a mapping, which can be used to verify that one has such an abstraction. This means that in the finite domain/propositional case, verifying that one has a sound abstraction is decidable and one can use theorem proving techniques to do it. Moreover, when one has complete information and a single model, one can use model-checking techniques. Bounded basic action theories [De Giacomo et al., 2016] constitute an important class of infinite domain theories where verification is decidable. In the general infinite-domain case, there are sound but incomplete reasoning methods that can be used [De Giacomo et al., 2010].

Automatically synthesizing sound/complete abstractions is another interesting problem. [Luo et al., 2020] show that one can use the well-explored notion of forgetting (of low-level fluent and action symbols) to automatically obtain a sound and complete high-level abstraction of a low-level BAT for a given mapping under certain conditions. They also show that such an abstraction is computable in the propositional case. However in general, there may be many different abstractions of a low-level theory, each of which may be useful for a different purpose. So defining an abstract language and mapping for a domain is not a trivial problem. Some human intervention is likely to be required, e.g., the modeler might specify the goals of the abstraction, or which details can be considered unimportant.

In this paper, we have not considered fairness constraints and strong cyclic plans [D’Ippolito et al., 2018; Aminof et al., 2020]; this is a topic for future work. Another interesting direction for future research is to consider a notion of multistage planning [Ciolek et al., 2020], where planning may be done using a simple model without unlikely contingencies, but where one can fall back to a more complex model when such contingencies do occur or more robustness is required. Related to this is the notion of “best effort strategies” (in presence of multiple [contradictory] assumptions about the environment) as proposed by [Aminof et al., 2021]. These are agent plans which, for each of the environment specifications individually, realize the agent’s goal against a maximal set of environments satisfying that specification. We are also interested in extending the current framework to deal with partial observability and sensing.
Acknowledgments

Work supported by the ERC Advanced Grant WhiteMech (No. 834228), by the EU ICT-48 2020 project TAILOR (No. 952215), as well as by the National Science and Engineering Research Council of Canada.

References


[De Giacomo et al., 2010] Giuseppe De Giacomo, Yves Lespérance, and Adrian R. Pearce. Situation calculus based programs for representing and reasoning about game structures. In Principles of Knowledge Representation and


