Situation Calculus Temporally Lifted Abstractions for Program Synthesis - Extended Version

Giuseppe De Giacomo[1,3] , **Yves Lespérance**[2] , **Matteo Mancanelli**[3]
[1]University of Oxford, Oxford, UK
[2]York University, Toronto, ON, Canada
[3]Sapienza University, Rome, Italy
giuseppe.degiacomo@cs.ox.ac.uk, lesperan@eecs.yorku.ca, mancanelli@diag.uniroma1.it

Abstract

We address a program synthesis task where one wants to automatically synthesize a controller that operates on data structures to accomplish an objective. We develop a framework that allows a programmer to provide an abstract description of how the data structure should evolve to satisfy the objectives and then derive a concrete program from that abstraction. The framework is based on the nondeterministic situation calculus, extended with LTL trace constraints. We propose a notion of *temporally lifted abstraction* to address the scenario in which we have a single high-level action theory/model with incomplete information and nondeterministic actions and a concrete action theory with several models and complete information. LTL formulas are used to specify the temporally extended goals as well as assumed trace constraints on the data structures used that hold at the concrete level. We show show that if we have such a temporally lifted abstraction and the agent has a strategy to achieve a LTL goal under some trace constraints at the abstract level, then there exists a refinement of the strategy to achieve the refinement of the goal at the concrete level. If the abstract theory is propositional, we can try to obtain such a strategy through LTL synthesis. We present several examples where we use our framework to solve problems involving data structures. We discuss the relationship of our approach to work on generalized planning.

1 Introduction

Program synthesis is the problem of generating a program that accomplishes a task given its specification. The foundation of program synthesis was set by (Green 1969; Waldinger and Lee 1969; Church 1963; Abadi, Lamport, and Wolper 1989; Pnueli and Rosner 1989). In this paper, we focus on program synthesis for tasks that involve the manipulation of data structures such as lists, trees, and graphs. (Bonet et al. 2020) (B20) proposed an approach for solving such problems that views them as a form of generalized planning, i.e., planning for solving multiple instances at once (Srivastava, Immerman, and Zilberstein 2008; Hu and De Giacomo 2011; Belle and Levesque 2016). For example, we may want to synthesize a program for finding the minimum value in a list for lists of any lengths. Their method works by having the programmer design a LTL observation abstraction of the required data structures behavior and then performing LTL synthesis to obtain a controller that completes the task. They provide several examples of how their method can be used to solve data structure manipulation problems. However, they do not provide complete formal specifications of the data structures used and formal proofs that the assumed temporal constraints and goal specifications hold for them.

Inspired by this work, in this paper, we present a program synthesis framework based on the situation calculus (McCarthy and Hayes 1969; Reiter 2001) that allows one to formally specify the low-level data structures and the high-level abstraction and associated LTL trace constraints (Bonet et al. 2017; Aminof et al. 2019), and prove that a controller synthesized for the abstract theory can be refined into one that achieves the goal at the low level. Our framework is based on the *nondeterministic situation calculus* (De Giacomo and Lespérance 2021) (DL21) where each agent action is accompanied by an environment reaction outside the agent's control that determines the action's outcome, e.g., a flipped coin may fall head or tail. (Banihashemi, De Giacomo, and Lespérance 2023) (BDL23) have proposed an account of abstraction for nondeterministic basic action theories (NDBATs) in this language. They relate a high-level NDBAT to a low-level NDBAT through a *refinement mapping* that specifies how a high-level action is implemented by a ConGolog program at the low level and how a high-level fluent can be defined by low-level state formula. They then define notions of sound and/or complete abstraction for such NDBATs in terms of a notion of bisimulation wrt such a mapping between their models. They prove that under suitable conditions, if an agent has a strategy to achieve a goal/complete a task at the high level, then one can obtain a strategy to do it at the low level. (Cui, Liu, and Luo 2021; Cui, Kuang, and Liu 2023) have adapted and extended this kind of approach to solve generalized planning problems, focusing on QNP abstractions.

To address the program synthesis problem for tasks that involve data structures, we focus on the case where the programmer designs a (typically propositional) high-level (HL) action theory/model with a limited set of HL fluents and nondeterministic actions, which abstracts over a concrete low-level (LL) action theory with multiple models, with a given refinement mapping m. At the LL, in each model we have complete information about the data structure's state, while at the HL, we have actions that may have several outcomes, e.g., after advancing to the next item in a list, we may

or may not reach the list's end. But we also have some HL LTL trace constraints, e.g., ensuring that if we keep advancing we will eventually reach the list's end. We define a notion of *temporally lifted abstraction* for such theories, where every LL trace that is a refinement of a sequence of HL actions is m-similar to a trace involving this action sequence in the HL model, and where the LTL trace constraints are satisfied by the LL theory. We then show that given such an abstraction, if we can use LTL synthesis on the HL model to obtain a HL strategy to achieve a LTL goal under the given trace constraints, then under certain conditions, we can automatically refine it to get a LL strategy that achieves the mapped LTL goal in all concrete instances of the problem. We illustrate our approach in detail with a list manipulation application and sketch how it can be applied to other examples, one involving a graph data structure.

2 Preliminaries

Situation calculus. The *situation calculus* is a well known predicate logic language designed 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*, which is a sequence of actions. The constant S_0 is used to denote the initial situation, and the function $do(a, s)$ is used to denote 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. The abbreviation $do([a_1, \ldots, a_n], s)$ stands for $do(a_n, do(a_{n-1}, \ldots, do(a_1, s) \ldots))$; for an action sequence $\vec{a}$, we often write $do(\vec{a}, s)$ for $do([\vec{a}], s)$. 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.

Nondeterministic situation calculus. (DL21) propose a simple extension of the standard situation calculus 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, depending on how the environment reacts to the agent's action. This is modeled by having every action type/function $A(\vec{x}, e)$ take an additional environment reaction parameter e, ranging over a new sort *Reaction*. We call the reaction-suppressed version of the action $A(\vec{x})$ an *agent action* and the full version $A(\vec{x}, e)$ a *system action*.

A *nondeterministic basic action theory* (NDBAT) can be seen as a special kind of BAT, where every action function takes an environment reaction argument, and for each agent action $A(\vec{x})$, we specify its agent action precondition $Poss_{ag}(A(\vec{x}), s)$. The NDBAT must entail the *reaction independence* requirement (formally, $\forall e.Poss(A(\vec{x}, e), s) \supset Poss_{ag}(A(\vec{x}), s)$) and the *reaction existence* requirement (formally, $Poss_{ag}(A(\vec{x}), s) \supset \exists e.Poss(A(\vec{x}, e), s)$). Formally, a NDBAT $\mathcal{D}$ is the union of the following disjoint sets: foundational, domain independent, axioms of the situation calculus (Σ), axioms describing the initial situation ($\mathcal{D}_{S_0}$), unique name axioms for actions ($\mathcal{D}_{una}$), successor state axioms (SSAs) describing how fluents change after *system* actions are performed ($\mathcal{D}_{ssa}$), and *system* action precondition axioms, one for each action type, stating when the complete system action can occur ($\mathcal{D}_{poss}$).

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

$$\delta ::= \alpha \mid \varphi? \mid \delta_1; \delta_2 \mid \delta_1 | \delta_2 \mid \pi x.\delta \mid \delta^* \mid \delta_1 \| \delta_2$$

In the above, α is an action term, possibly with parameters, and φ is a situation-suppressed formula. As usual, we denote by $\varphi[s]$ the formula obtained from φ by restoring the situation argument s into all fluents in φ. The sequence of program δ_1 followed by program δ_2 is denoted by $\delta_1; \delta_2$. Program $\delta_1 | \delta_2$ allows for the nondeterministic choice between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* nondeterministic choice of a binding for object variable x. δ^* performs δ zero or more times. Program $\delta_1 \| \delta_2$ expresses the interleaved execution of programs δ_1 and δ_2. Conditional and while-loop constructs are definable as follows: **if** ϕ **then** δ_1 **else** δ_2 **endIf** $= \phi?; \delta_1 | \neg\phi?; \delta_2$ and **while** ϕ **do** δ **endWhile** $= (\phi?; \delta)^*; \neg\phi?$. We also use $nil = True?$.

Formally, the semantics of ConGolog is specified in terms of single-step transitions, using the following two predicates: *(i)* $Trans(\delta, s, \delta', s')$, which holds if one step of program δ in situation s may lead to situation s' with δ' remaining to be executed; and *(ii)* $Final(\delta, s)$, which holds if program δ may legally terminate in situation s. The definitions of $Trans$ and $Final$ we use are as in (De Giacomo, Lespérance, and Pearce 2010); differently from (De Giacomo, Lespérance, and Levesque 2000), the test construct $\varphi?$ does not yield any transition, but is final when satisfied. Predicate $Do(\delta, s, s')$ means that program δ, when executed starting in situation s, has as a legal terminating situation s', and is defined as $Do(\delta, s, s') \doteq \exists \delta'.Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$ where *Trans** denotes the reflexive transitive closure of *Trans*. We use $\mathcal{C}$ to denote the axioms defining the ConGolog programming language.

For simplicity, we use a restricted class of ConGolog programs that are *situation-determined* (SD) (De Giacomo, Lespérance, and Muise 2012), i.e., for every action sequence, the remaining program is uniquely determined by the resulting situation: $SitDet(\delta, s) \doteq \forall s', \forall \delta', \delta''.$ *Trans*$^*(\delta, s, \delta', s') \wedge$ *Trans*$^*(\delta, s, \delta'', s') \supset \delta' = \delta''$

LTL and generalized planning. Linear Temporal Logic (LTL) is one of the most popular formalisms for expressing the temporal properties of reactive systems (Pnueli 1977).

Given a set of atomic propositions P the formulas of LTL are generated by the following grammar:

$$\phi ::= a \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \bigcirc\phi \mid \phi_1 \,\mathcal{U}\, \phi_2$$

where $a \in P$. We use common abbreviations such as *eventually* as $\Diamond\phi \doteq \mathit{True}\,\mathcal{U}\,\phi$ and *always* as $\Box\phi \doteq \neg\Diamond\neg\phi$.

Formulas of LTL are interpreted over infinite sequences (called traces) of truth evaluations of variables in P , i.e. $\pi = \pi_0, \pi_1, \cdots \in (2^P)^\infty$. Given a trace π, we define when an LTL formula ϕ holds at position i on π, written $\pi, i \models \phi$, inductively on the structure of ϕ, as follows:

- $\pi, i \models a$ iff $a \in \pi_i$ (for $a \in P$);
- $\pi, i \models \neg\phi$ iff $\pi, i \not\models \phi$;
- $\pi, i \models \phi_1 \vee \phi_2$ iff $\pi, i \models \phi_1$ or $\pi, i \models \phi_2$;
- $\pi, i \models \bigcirc\phi$ iff $\pi, i+1 \models \phi$;
- $\pi, i \models \phi_1 \,\mathcal{U}\, \phi_2$ iff there exists $j \geq i$ such that $\pi, j \models \phi_2$, and for all k, $i \leq k < j$ we have that $\pi, k \models \phi_1$.

We say that π satisfies ϕ, written $\pi \models \phi$, if $\pi, 0 \models \phi$.

LTL synthesis (Pnueli and Rosner 1989) is the problem of producing a controller that achieves a given property no matter how the environment behaves. If X and Y are two disjoint sets of variables which can be set alternately in discrete stages by the environment and the controller respectively, the goal is to obtain a controller policy μ such that every sequence induced by μ satisfies a given formula ϕ. Formally, a controller policy returns the controller choice Y_i given a prefix sequence $\langle X_0, Y_0, \ldots, X_i\rangle$ as input.

In (B20), it is shown how one can reduce a generalized planning problem to an LTL synthesis problem. A *generalized planning problem* Q is a set of instances P with the same actions, the same observations, and the same observable action preconditions. They define each instance P as a set of states, actions and observations, plus the functions for the state transitions, the observations and the actions available in a state. The infinite set Q of instances can be abstracted into a single nondeterministic problem $Q^A = \langle Q^o, \Gamma_F, \Gamma_G\rangle$, called *observation abstraction,* where Q^o is a (fully-observable nondeterministic) observation projection of the problem that captures the possible transitions among observations, Γ_F is a set of fairness constraints (temporal assumptions), and Γ_G is a set of goal constraints on observation-action trajectories. By representing Q^A with LTL formulas, one can solve a generalized planning problem through an LTL synthesis engine.

LTL constraints in NDBATs. We are interested in imposing some LTL temporal properties that can work as trace constraints to filter the considered world histories in the context of NDBATs. A convenient way for doing this is to leverage on the axiomatization of infinite paths introduced by (Khan and Lespérance 2016) (KL16), which provided a natural way to talk about "infinite future histories". An infinite sequence of situations is called a path, and we have a special sort *paths* and a predicate $OnPath(p, s)$ meaning that situation s is on path p. Additionally, we'll use $Starts(p, s)$, to specify that the path p starts with situation s, and $\mathit{Suffix}(p', p, s)$, which means that the path p' starts with s and contains the same situations as p starting from s.

Based on this, we can define an abbreviation $Holds(\psi, p)$ to specify that a given LTL temporal property ψ holds on path p. Here, we assume that the atomic propositions ϕ in LTL formulas are ground situation-suppressed formulas defined over an NDBAT.

Definition 1. *If ψ is an LTL trace constraint and p is an infinite path, we define $Holds(\psi, p)$ (meaning that the constraint ψ holds on path p) inductively as follows*[1]

$$\begin{aligned}
&Holds(\phi, p) \doteq \exists s.Starts(p, s) \wedge \phi[s]\\
&Holds(\neg\psi, p) \doteq \neg Holds(\psi, p)\\
&Holds(\psi_1 \vee \psi_2, p) \doteq Holds(\psi_1, p) \vee Holds(\psi_2, p)\\
&Holds(\bigcirc\psi, p) \doteq\\
&\quad \exists s, a, s', p'.Starts(p, s) \wedge s' = do(a, s) \wedge\\
&\quad \mathit{Suffix}(p', p, s') \wedge Holds(\psi, p')\\
&Holds(\psi_1 \,\mathcal{U}\, \psi_2, p) \doteq\\
&\quad \exists s, s', p'.Starts(p, s) \wedge s \preceq s' \wedge\\
&\quad \mathit{Suffix}(p', p, s') \wedge Holds(\psi_2, p') \wedge \forall s'', p''.\\
&\quad (s \preceq s'' \prec s' \wedge \mathit{Suffix}(p'', p, s'')) \supset Holds(\psi_1, p'')
\end{aligned}$$

Note that the original axiomatization of infinite paths does not consider nondeterministic actions but it can be easily extended also to the NDBATs considering system actions.

3 Temporally Lifted Abstractions

We want to specify program synthesis tasks involving data structures and synthesize controllers for them. We will specify such data structure at the concrete level by a BAT $\mathcal{D}_l$, with the various instances of the task being models of this BAT. We will then define an abstract version of the task by providing a typically propositional NDBAT $\mathcal{D}_h$, for which we have a single model and complete information, with the nondeterminism capturing the differences between the different concrete task instances. We will also specify some LTL trace constraints on the abstract model that characterize the traces that can actually arise in the concrete theory/task instances. We call these temporally lifted abstractions. Then we will be able to try to solve the general program synthesis tasks through LTL synthesis on the abstraction.

For this to work, we need to ensure that executions of refinements HL actions in models of the LL theory correspond to executions in the HL theory/model. So we will specify the relationship between the HL NDBAT and the LL BAT by a refinement mapping m. For this, we extend the notion of refinement mapping for NDBAT abstractions from (BDL23) to handle LTL trace constraints. We will then ensure that executions of HL actions in the models corresponds through a form of simulation relative to the refinement mapping m.

NDBAT Refinement Mapping with Trace Constraints. In (BDL23), an 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 to a ConGolog agent program δ_A^{ag} defined over the LL theory that implements the agent action (i.e., $m_a(A(\vec{x})) = \delta_A^{ag}(\vec{x})$)

[1] $s \prec s'$ denotes that the actions performed to reach s' from s were all executable; $s \preceq s'$ stands for $s \prec s' \vee s = s'$

- m_s associates each A to a ConGolog system program δ_A^{sys} defined over the LL theory that implements the system action (i.e., $m_s(A(\vec{x}, e)) = \delta_A^{sys}(\vec{x}, e)$)
- m_f maps each situation-suppressed HL fluent $F(\vec{x})$ to a situation-suppressed formula $\phi_F(\vec{x})$ defined over the LL theory that characterizes the conditions under which $F(\vec{x})$ holds in a situation (i.e., $m_f(F(\vec{x})) = \phi_F(\vec{x})$)

To ensure that the reaction existence requirement and the reaction independence requirement hold at LL, we require the refinement mapping m to be proper wrt $\mathcal{D}_l$:

Constraint 2 (Proper Refinement Mapping)**.** *For every HL system action sequence $\vec{\alpha}$ and every HL action A, we have:*

$$\mathcal{D}_l \cup \mathcal{C} \models \forall s.(Do(m_s(\vec{\alpha}), S_0, s) \supset \forall \vec{x}, s'.$$
$$(Do_{ag}(m_a(A(\vec{x})), s, s') \equiv \exists e.Do(m_s(A(\vec{x}, e)), s, s'))$$

We want to extend the NDBAT refinement mapping to also handle HL LTL trace constraints, which we want to be able to map to equivalent LL trace constraints. For this, we introduce an additional constraint from (Lespérance et al. 2024) so that the LL theory tracks when refinements of HL actions end using a state formula $Hlc(s)$, meaning that a HL action has just completed in situation s:

Constraint 3. *$\mathcal{D}_l \cup C \models Hlc(s)$ if and only if there exists a HL system action sequence $\vec{\alpha}$ such that $\mathcal{D}_l \cup C \models Do(m(\vec{\alpha}), S_0, s)$.*

There are various ways to define $Hlc(s)$ to satisfy this constraint; see the example for some discussion.

The revisited definition of NDBAT mapping maintains the previous elements and includes a new component m_t, which specifies how HL trace constraints are mapped to the LL:

Definition 4 (Refinement Mapping for Trace Constraints)**.** *Let ψ be an LTL trace constraint and Hlc a distinguished symbol which signals that a HL action is completed. A NDBAT refinement mapping m is a tuple $\langle m_a, m_s, m_f, m_t \rangle$, where m_a, m_s and m_f are defined as usual and m_t is a mapping for trace constraints defined as follows:*

$$m_t(\psi) \doteq m_f(\psi)$$
$$m_t(\neg\psi) \doteq \neg m_t(\psi)$$
$$m_t(\psi_1 \vee \psi_2) \doteq m_t(\psi_1) \vee m_t(\psi_2)$$
$$m_t(\bigcirc\psi) \doteq \bigcirc(\neg Hlc\,\mathcal{U}(Hlc \wedge m_t(\psi)))$$
$$m_t(\psi_1\,\mathcal{U}\,\psi_2) \doteq (Hlc \supset m_t(\psi_1))\,\mathcal{U}(Hlc \wedge m_t(\psi_2))$$

Here, we say that if $\bigcirc\psi$ holds at the HL, then it must hold at the LL in the first situation when a HL action has been completed. For $\psi_1\,\mathcal{U}\,\psi_2$, ψ_1 must hold for every situation where a HL action has been completed until ψ_2 becomes true in a LL situation where a HL action has been completed.

Temporally lifted abstractions. Finally, we can present the concept of temporally lifted abstractions. To relate the HL and LL models/theories, we first define:

Definition 5 (m-isomorphic situations)**.** *We say that situation s_h in M_h is m-isomorphic to situation s_l in M_l, written $s_h \simeq_m^{M_h, M_l} s_l$, if and only if*

$$M_h, v[s/s_h] \models F(\vec{x}, s) \textit{ iff } M_l, v[s/s_l] \models m(F(\vec{x}))[s]$$

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

If $s_h \simeq_m^{M_h, M_l} s_l$, then s_h and s_l evaluate all HL fluents the same.

(BDL23) and the earlier (Banihashemi, De Giacomo, and Lespérance 2017) (BDL17) define a variant of bisimulation (Milner 1971; Milner 1989) in order to establish a relation based on the refinement mapping. In our framework, we stick to a unidirectional version:

Definition 6 (m-simulation)**.** *A relation $R \subseteq \Delta_S^{M_h} \times \Delta_S^{M_l}$ is an m-simulation relation between M_h and M_l if $\langle s_h, s_l \rangle \in R$ implies*

1. *s_h is m-isomorphic to s_l*
2. *for every HL system action A, if there exists s_l' such that $M_l, v[s/s_l, s'/s_l'] \models Do(m(A(\vec{x})), s, s')$, then there exists s_h' such that $M_h, v[s/s_h, s'/s_h'] \models Poss(A(\vec{x}), s) \wedge s' = do(A(\vec{x}), s)$ and $\langle s_h', s_l' \rangle \in R$*

We say that M_h is m-similar to M_l wrt the mapping m (written $M_h \sim_m^{\leftarrow} M_l$) iff there exists an m-simulation relation R between M_h and M_l such that $\langle S_0^{M_h}, S_0^{M_l} \rangle \in R$. We also say that a situation s_h in M_h is m-similar to situation s_l in M_l (written $s_h \sim_m^{M_h, M_l, \leftarrow} s_l$) iff there exists an m-simulation relation R between M_h and M_l and $\langle s_h, s_l \rangle \in R$.

Having an m-simulation means that if a refinement of a HL action can occur, so can the HL action. A similar idea is present also in (Cui, Liu, and Luo 2021), where the definition of m-bisimulation is decomposed into m-simulation and m-back-simulation to represent the two directions.

At last, exploiting m-simulation together with the use of LTL trace constraints, we can talk about the notion of temporally lifted abstractions. Intuitively, we have a temporally lifted abstraction if there is an m-simulation between an HL model/theory and all the models of a LL theory, and every trace constraint is satisfied on some path at the HL and on all paths at the LL.

Definition 7 (Temporally Lifted Abstraction)**.** *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. We say that $(\mathcal{D}_h, M_h, \Psi)$ is a temporally lifted abstraction wrt m if and only if*

- *M_h m-simulates every model M_l of D_l*
- *for every high-level LTL trace constraint $\psi \in \Psi$, $M_h \models \exists p_h.Starts(p_h, S_{0_h}) \wedge Holds(\psi, p_h)$ 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). To illustrate our framework, we use common programming problems and data structures. As a first example, we consider the task of finding the minimum value inside a singly-linked list. NDBAT $\mathcal{D}_l^{sll}$ is the LL action theory that describes the operations that can be performed and how they affect the predicates representing lists. Here, we restrict the discussion to the actions and the fluents related to the examined task. Note that at the LL, we assume we have complete information and deterministic actions, thus we have only one possible environment reaction *Success* for each action.

In detail, we use $sll(head, situation)$, to identify the head of the list, $node(node\_id, value, next\_id, situation)$,

to store the value of each node of the list and a pointer to the following node, $iterator(cursor, situation)$, to represent a cursor that points to a certain node and can scan the list, and $min\_register(value, situation)$, the register used to store the minimum value. Wlog, we assume the domains of the arguments of all list fluents are integers (except for situations); a special constant END is used for the end of the list. The actions necessary for our task are it_set, which creates an iterator pointing to the head of the list, it_next, which moves the cursor from the pointed node to the next one, and it_get, to write the value of the pointed node into the register. Additionally, we have a no_op action, with no precondition and no effect, and an action $mark(i, t)$, that affects the fluent $marked$ and is used to assign a label t to a node i (in our case the label will be *visited*, that indicates when a node has been processed).

$\mathcal{D}_l^{sll}$ includes the following action precondition axioms (throughout the paper, we assume that free variables are universally quantified from the outside):

$$
\begin{aligned}
&Poss_{ag}(it\_set, s) \doteq \exists h.sll(h, s)\\
&Poss_{ag}(it\_next, s) \doteq \exists c, v, k.iterator(c, s) \wedge\\
&\quad node(c, v, k, s) \wedge k \neq \text{END}\\
&Poss_{ag}(it\_get, s) \doteq \exists c, v, k.iterator(c, s) \wedge node(c, v, k, s)\\
&Poss_{ag}(mark(i, t), s) \doteq True\\
&Poss_{ag}(no\_op, s) \doteq True\\
&Poss(it\_set(r), s) \equiv Poss_{ag}(it\_set, s) \wedge\\
&\quad r = Success_{IS}\\
&Poss(it\_next(r), s) \equiv Poss_{ag}(it\_next, s) \wedge\\
&\quad r = Success_{IN}\\
&Poss(it\_get(r), s) \equiv Poss_{ag}(it\_get, s) \wedge r = Success_{IG}\\
&Poss(mark(i, t, r), s) \equiv Poss_{ag}(mark(i, t), s) \wedge\\
&\quad r = Success_{MD}\\
&Poss(no\_op(r), s) \equiv Poss_{ag}(no\_op, s) \wedge r = Success_{NO}
\end{aligned}
$$

$\mathcal{D}_l^{sll}$ also includes the following successor state axioms:

$$
\begin{aligned}
&iterator(c, do(a, s)) \equiv\\
&\quad a = it\_set(Success_{IS}) \wedge [\exists h.sll(h, s) \wedge c = h] \vee\\
&\quad a = it\_next(Success_{IN}) \wedge [\exists c_0, v.iterator(c_0, s) \wedge\\
&\qquad node(c_0, v, c, s)] \vee\\
&\quad iterator(c, s) \wedge a \neq it\_set(Success_{IS}) \wedge\\
&\qquad a \neq it\_next(Success_{IN})\\
&min\_register(v, do(a, s)) \equiv\\
&\quad a = it\_get(Success_{IG}) \wedge [\exists c, k.iterator(c, s) \wedge\\
&\qquad node(c, v, k, s)] \vee\\
&\quad min\_register(v, s) \wedge a \neq it\_get(Success_{IG})\\
&marked(i, t, do(a, s)) \equiv a = mark(i, t, Success_{MD})
\end{aligned}
$$

At the HL, we can abstract details such as the creation of the iterator and we use nondeterministic actions to handle incomplete information about the states. ND-BAT $\mathcal{D}_h^{sll}$ models the HL action theory with two fluents: $hasNext(situation)$, signaling whether the cursor points at the last node of the list, and $lowerThan(situation)$, signaling whether the value of the node pointed by the cursor is lower than the value stored in the register. Note that, if we want to perform LTL synthesis to solve the task, we need both fluents and actions to be propositional (apart from the situation argument). The actions used for our task will be $next$, which moves the cursor if possible, $checkValue$, which performs the comparison between the node and register values, and $update$, which updates the register's value to the node's. In every HL theory, we introduce also the action $stop$, used in infinite paths when the goal is already reached.

The action precondition axioms in $\mathcal{D}_h^{sll}$ are:

$$
\begin{aligned}
&Poss_{ag}(next, s) \doteq hasNext(s)\\
&Poss_{ag}(checkValue, s) \doteq True\\
&Poss_{ag}(update, s) \doteq lowerThan(s)\\
&Poss_{ag}(stop, s) \doteq True\\
&Poss(next(r), s) \equiv\\
&\quad Poss_{ag}(next, s) \wedge (r = End \vee r = NotEnd)\\
&Poss(checkValue(r), s) \equiv\\
&\quad Poss_{ag}(checkValue, s) \wedge (r = \text{LT} \vee r = \text{GEQ})\\
&Poss(update(r), s) \equiv\\
&\quad Poss_{ag}(update, s) \wedge r = Success_{HU}\\
&Poss(stop(r), s) \equiv Poss_{ag}(stop, s) \wedge r = Success_{HS}
\end{aligned}
$$

Since we have no knowledge of the node components of the list at the HL, we obtain the necessary information via the environment reactions. The action $next$ collects observations on reaching the end of the list, while $checkValue$ collects observations on whether the node's value is less than the register's. The SSAs are straightforward:

$$
\begin{aligned}
&hasNext(do(a, s)) \equiv hasNext(s) \wedge a \neq next(End)\\
&lowerThan(do(a, s)) \equiv\\
&\quad a = checkValue(\text{LT}) \vee\\
&\quad lowerThan(s) \wedge a \neq checkValue(\text{GEQ}) \wedge\\
&\qquad \forall r.a \neq next(r) \wedge a \neq update(Success_{HU})
\end{aligned}
$$

We specify the relationship between the HL and LL ND-BATs through the following refinement mapping m^{sll}:

$$
\begin{aligned}
&m_a(next) = it\_next\\
&m_a(checkValue) =\\
&\quad \textbf{if } \neg\exists c.iterator(c) \textbf{ then } it\_set \textbf{ else } nil \textbf{ endIf}\\
&\quad (\pi c).[iterator(c)?; mark(c, \text{visited})]\\
&m_a(update) = it\_get\\
&m_a(stop) = no\_op\\
&\\
&m_s(next(r_h)) =\\
&\quad it\_next(Success_{IN});\\
&\quad \textbf{if } \exists c, v.iterator(c) \wedge node(c, v, \text{END})\\
&\qquad \textbf{then } r_h = \text{End ? else } r_h = \text{NotEnd ? endIf}\\
&m_s(checkValue(r_h)) =\\
&\quad \textbf{if } \neg\exists c.iterator(c) \textbf{ then } it\_set(Success_{IC}) \textbf{ else } nil \textbf{ endIf}\\
&\quad (\pi c).[iterator(c)?; mark(c, \text{visited}, Success_{MD})];\\
&\quad \textbf{if } \exists c, v, k, v'.iterator(c) \wedge node(c, v, \text{END}) \wedge\\
&\qquad min\_register(v') \wedge v < v'\\
&\qquad \textbf{then } r_h = \text{LT ? else } r_h = \text{GEQ ? endIf}\\
&m_s(update(r_h)) = it\_get(Success_{IG}); r_h = Success_{HU}?\\
&m_s(stop(r_h)) = no\_op(Success_{NO}); r_h = Success_S?\\
&\\
&m_f(hasNext) = \exists c, v, k.iterator(c) \wedge\\
&\quad node(c, v, k) \wedge k \neq \text{End}\\
&m_f(lowerThan) = \exists c, v, k.iterator(c) \wedge node(c, v, k) \wedge\\
&\quad marked(c, visited) \wedge \exists v'.min\_register(v') \wedge v < v'
\end{aligned}
$$

$checkValue$ also abstracts the creation of the iterator, i.e., creates the iterator if it does not exist.

To properly deal with trace constraints with a refinement mapping as described in Definition 4, we need to present some more components to both HL and LL theories. First we specify how to deal with the predicate $Hlc(s)$ that denotes at the LL when an HL action is completed. A simple way to do that is to introduce two LL actions $startHLAction$ and $endHLAction$

which make $Hlc(s)$ false and true respectively, and place them at the beginning and at the end of the refinement program of each HL action (e.g., $m_a(next) = startHLAction; it\_next; endHLAction$). In this way Constraint 3 is satisfied by construction. We also use an additional HL fluent for each action, namely $doneNext$, $doneCheckValue$ and $doneUpdate$, to signal the last action executed. The axioms for these fluents are straightforward. The same can be done for the LL, adding new dedicated actions to make refinement consistent.

Finally, we can consider the HL LTL trace constraint:

$$(\Box\Diamond doneNext) \rightarrow \Diamond\neg hasNext$$

It specifies that moving repeatedly to the next node of the list eventually leads to the last one.

We can prove the following (see appendix for proofs):

Proposition 8. *NDBAT refinement mapping m^{sll} is proper wrt $\mathcal{D}_l^{sll}$.*

Proposition 9. *Let M_h^{sll} be a model of $\mathcal{D}_h^{sll}$ and Ψ the set of trace constraints. $(\mathcal{D}_h^{sll}, M_h^{sll}, \Psi)$ is a temporally lifted abstraction of $\mathcal{D}_l^{sll}$ wrt m^{sll}.*

4 Results about Action Executions

Next, we present some results about action executions when using abstraction and m-similar models. Unless stated otherwise, we assume we have NDBATs $\mathcal{D}_h$ and $\mathcal{D}_l$, that $M_h \models \mathcal{D}_h \cup \mathcal{C}$ and $M_h \models \mathcal{D}_h \cup \mathcal{C}$, and that m is proper wrt $\mathcal{D}_l$.

First, as in (BDL17) and (BDL23), we can show that m-isomorphic situations satisfy the same high-level situation-suppressed formulas:

Lemma 10. *If $s_h \simeq_m^{M_h, M_l, \leftarrow} s_l$, then for any high-level situation-suppressed formula ϕ, we have that:*

$$M_h, v[s/s_h] \models \phi[s] \text{ if and only if } M_l, v[s/s_l] \models m(\phi)[s].$$

Secondly, we can also show that in m-similar models, the same sequences of high-level actions are executable, and that the resulting situations are m-similar:

Lemma 11. *If $M_h \sim_m^{\leftarrow} M_l$, then for any sequence of high-level system actions $\vec{\alpha}$, we have that*

$$\begin{array}{l} \textit{if } M_l, v[s'/s_l] \models Do(m(\vec{\alpha}), S_0, s') \\ \textit{then } M_h, v[s'/s_h] \models s' = do(\vec{\alpha}, S_0) \wedge Executable(s') \\ \quad \textit{and } s_h \sim_m^{M_h, M_l, \leftarrow} s_l \end{array}$$

It follows that if some refinement of a sequence of high-level system actions $\vec{\alpha}$ can be executed at the low level and the refinement of some formula ϕ hold afterwards, then the same sequence of actions can also be executed at the high level and ϕ holds afterwards:

Theorem 12. *If $M_h \sim_m^{\leftarrow} M_l$, then for any sequence of ground high-level actions $\vec{\alpha}$ and any high-level situation-suppressed formula ϕ, we have that*

$$\begin{array}{l} \textit{if } M_l \models \exists s'.Do(m(\vec{\alpha}), S_0, s') \wedge m(\phi)[s'] \\ \textit{then } M_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)] \end{array}$$

Next, we show a result about LTL formulas in m-similar models. First, we define a notion of refinement over paths:

Definition 13. *We say that a LL path p_l in M_l* is a refinement of *a HL path p_h in M_h wrt mapping m iff for every finite HL action sequence $\vec{a_h}$*

$$\begin{array}{l} \textit{if } M_h \models \exists s.OnPath(p_h, s) \wedge Do(\vec{a_h}, S_0, s) \\ \textit{then } M_l \models \exists s.OnPath(p_l, s) \wedge Do(m(\vec{a_h}), S_0, s) \end{array}$$

That is, p_l is a refinement of p_h if it contains a refinement of the infinite action sequence that occurs over p_h.

We can then show that if a LTL formula ψ holds on a path p_h in a HL model M_h, then the mapped version of ψ must hold at the LL on refinements of p_h in m-similar models:

Lemma 14. *Consider $M_h \sim_m^{\leftarrow} M_l$ for which Constraint 3 holds and an LTL formula ψ. If $M_h \models Holds(\phi, p_h)$ and p_l in M_l is a refinement of p_h in M_h wrt m, then $M_l \models Holds(m_t(\phi), p_l)$.*

Proof. Consider a given HL action sequence $\vec{a_h}$. Since $M_h \sim_m^{\leftarrow} M_l$, we know that every time an action from $\vec{a_h}$ is completed both at the HL and (refined) at the LL, the paths p_h and p_l have m-similar situations. Since $M_h \models Holds(\phi, p_h)$ by hypothesis, we can show that $M_l \models Holds(m(\phi), p_l)$, i.e. the refinement of ϕ holds in path p_l, by induction on the structure of ϕ using definition 1. □

5 Results about Strategic Reasoning

Now we want to address the problem of synthesis in the context of temporally lifted abstractions, that is, generating strategies that achieve given goals at the abstract and at the concrete level.

For NDBATs, a strong plan is a strategy for the agent that guarantees the achievement of a goal no matter how the environment reacts. (DL21) formalize this notion for state goals and finite traces. They define a strategy as a function from situations to agent actions, i.e. $f(s) = A(\vec{x})$ (note that the value may depend on the entire history). The special agent action $stop$ (with no effects and preconditions) may be returned when the strategy wants to stop (for a finite strategy).

Here, we extend their definition to handle LTL goals and LTL trace constraints over infinite paths. We define $AgtCanForceByIf(Goal, Cstr, f, s)$, meaning that the agent can force a LTL $Goal$ to hold no matter how the environment responds to her actions by following strategy f in situation s if we assume that the LTL trace/path constraint $Cstr$ holds, as follows:

$$\begin{array}{l} AgtCanForceByIf(Goal, Cstr, f, s) \doteq \\ \quad \forall p.Out(p, f, s) \wedge Holds(Cstr, p) \supset \\ \quad Holds(Goal, p) \end{array}$$

where

$$\begin{array}{l} Out(p, f, s) \doteq \\ \quad \forall a.\forall s.OnPath(p, s) \wedge OnPath(p, do(a, s)) \supset \\ \quad Do_{ag}(f(s), s, do(a, s)) \end{array}$$

$Out(p, f, s)$ means that path p is a possible outcome of the agent executing strategy f in situation s. Note that if we have a finite set of constraints, we can simply consider $Cstr$ as the conjunction of them. We also define:

$$\begin{array}{l} AgtCanForceIf(Goal, Cstr, s) \doteq \\ \quad \exists f.AgtCanForceByIf(Goal, Cstr, f, s) \end{array}$$

Since in (KL16) paths are defined as infinite sequences of executable situations, we should also require that the strategy is certainly executable, i.e., never prescribes an action that is not executable. It is possible to capture this by defining co-inductively a predicate $CertainlyExecutable(f, s)$, meaning that strategy f is certainly executable in situation s, as follows:

$$CertainlyExecutable(f, s) \doteq \exists P.[\forall s.P(s) \supset \ldots] \wedge P(s)$$
where $\ldots$ stands for
$$[Poss_{ag}(f(s), s)] \wedge$$
$$[\forall s'.Do_{ag}(f(s), s, s') \supset P(s')]$$

To include this requirement, we would write:

$$AgtCanForceByIf(Goal, Cstr, f, s) \doteq$$
$$CertainlyExecutable(f, s) \wedge \forall p.Out(p, f, s) \wedge$$
$$Holds(Cstr, p) \supset Holds(Goal, p)$$

We also need to consider whether the agent is able to execute a program to completion, i.e., the implementation of a HL action, no matter how the environment reacts. For this, (DL21) introduce $AgtCanForceBy(\delta, s, f)$, meaning that the agent can ensure that it executes program δ to completion by following strategy f:

$$AgtCanForceBy(\delta, f, s) \doteq \forall P.[\ldots \supset P(\delta, s)]$$
where $\ldots$ stands for
$$[(f(s) = stop \wedge Final(\delta, s)) \supset P(\delta, s)] \wedge$$
$$[\exists A.\exists \vec{t}.(f(s) = A(\vec{t}) \neq stop \wedge$$
$$\exists e.\exists \delta'.Trans(\delta, s, \delta', do(A(\vec{t}, e), s)) \wedge$$
$$\forall e.(\exists \delta'.Trans(\delta, s, \delta', do(A(\vec{t}, e), s))) \supset$$
$$\exists \delta'.Trans(\delta, s, \delta', do(A(\vec{t}, e), s)) \wedge P(\delta', do(A(\vec{t}, e), s))$$
$$\supset P(\delta, s)]$$

Now we can talk about planning with abstractions and how a plan at the abstract level is related to one at the concrete level. As in (BDL23), we impose an additional constraint on action implementation which requires that for any HL agent action that is executable at the LL, the agent has a strategy to execute it no matter how the environment reacts:

Constraint 15 (Agent Can Always Execute HL actions)**.** *For every HL action A, there exists a LL strategy f_A such that for every HL system action sequence $\vec{\alpha}$:*

$$\mathcal{D}_l \models \forall s.Do(m(\vec{\alpha}, S_0, s) \supset$$
$$(\forall \vec{x}.\exists s'.Do_{ag}(m_a(A(\vec{x})), s, s') \supset$$
$$AgtCanForceBy(m_a(A(\vec{x})), f_A, s))$$

Then, we can prove our main result, that is, given a temporally lifted abstraction, if the agent has a strategy to achieve a LTL goal assuming some LTL constraints at the high level, then there exists a refinement of the HL strategy that ensures it achieves the refinement of the goal at the low level:

Theorem 16. *Let $(\mathcal{D}_h, M_h, Cstr)$ be a temporally lifted abstraction of $\mathcal{D}_l$ wrt refinement mapping m s.t. Constraints 3 and 15 hold, and Goal be an LTL goal. Then we have that:*

if $M_h \models AgtCanForceIf(Goal, Cstr, S_0)$,
then there exist a LL strategy f_l such that
$\mathcal{D}_l \models AgtCanForceBy(m(Goal), \mathit{True}, f_l, S_0)$

Proof. Since $M_h \models AgtCanForceIf(Goal, Cstr, S_0)$, it follows that there is a HL strategy f_h such that $M_h \models AgtCanForceByIf(Goal, Cstr, f_h, S_0)$. We have that $(\mathcal{D}_h, M_h, Cstr)$ is a temporally lifted abstraction of $\mathcal{D}_l$ wrt refinement mapping m. Let M_l be a model of $\mathcal{D}_l$. By the definition of temporally lifted abstraction, $M_h \sim_m^{\leftarrow} M_l$.

By Constraint 15, for every HL action A, there exists a LL strategy f_A such that for every HL system action sequence $\vec{\alpha}$, we have $\mathcal{D}_l \cup \mathcal{C} \models \forall s.Do(m(\vec{\alpha}), S_0, s) \supset (\forall \vec{x}.\exists s'.Do_{ag}(m_a(A(\vec{x})), s, s') \supset AgtCanForceBy(m_a(A(\vec{x})), f_A, s))$.

Given the HL strategy f_h, we can define a corresponding LL strategy f_l as follows: $f_l(s_l) = f_{f_h(s_h)}(s_l)$ where $s_h \sim_m^{M_h, M_l, \leftarrow} s_l'$, $s_l' \leq s_l$ and for every s_l'' such that $s_l' < s_l'' \leq s_l$, it is not the case that $s_h \sim_m^{M_h, M_l, \leftarrow} s_l''$ (s_h is the HL situation that is similar to the latest predecessor of s_l that has such a similar situation at the HL).

Since $M_h \models AgtCanForceByIf(Goal, Cstr, f_h, S_0)$, every path p_h produced by f_h in M_h that satisfies $Cstr$ also satisfies $Goal$. Since $M_h \sim_m^{\leftarrow} M_l$, for any path p_l such that $M_l \models Out(p_l, f_l, S_0)$, p_l is a refinement of a path p_h in M_h. Then the thesis follows by Lemma 14 (we use the fact that that $(Holds(Cstr, p) \supset Holds(Goal, p)) \equiv Holds(\neg Cstr \vee Goal, p)$). □

Example Cont. Continuing with our running example, let's discuss the HL and LL strategies for finding the minimum value in a singly-linked list. First, we can show the following proposition:

Proposition 17. *NDBAT $\mathcal{D}_h^{sll}$ and $\mathcal{D}_l^{sll}$ and mapping m^{sll} satisfy constraint 15.*

The HL LTL goal constraints that must be satisfied are:

$$\Diamond\Box\neg hasNext$$
$$\Box(doneNext \rightarrow \bigcirc doneCheckValue)$$
$$\Box(lowerThan \leftrightarrow \bigcirc doneUpdate)$$

The first says that the list must be scanned till the end; the second forces the execution of the action $checkValue$ each time a new node is encountered; the third says that the value of the register must be updated iff the pointed node has a lower value than the register.

A HL strategy that guarantees satisfaction the goals is:

$$f_h(s) = \begin{cases} checkValue & \text{if } doneNext(s) \\ update & \text{if } lowerThan(s) \\ next & \text{if } \neg lowerThan(s) \wedge hasNext(s) \\ stop & \text{otherwise} \end{cases}$$

This strategy prescribes to update the value of the register whenever the node has a lower value, move the cursor when

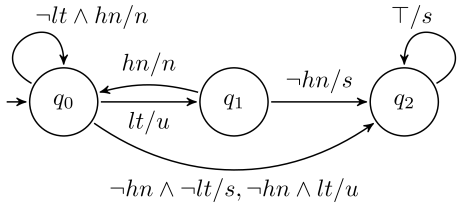

Figure 1: Controller for finding minimum in a list.

it is not at the end of the list and perform the comparison after each cursor move. In (B20), authors addressed the same task using LTL synthesis engine Strix (Meyer, Sickert, and Luttenberger 2018) and obtained the strategy in figure 1. Note that the comparison between the node and the register is implicitly assumed after each cursor move. Despite this difference, it's easy to see that our strategy is consistent with theirs.

At the LL the strategy can be refined as follows (it, md, vis abbreviate $iterator$, $marked$, $visited$):

$$f_l(s) = \begin{cases} it\_set & \text{if } \neg\exists c.it(c,s) \\ mark(c, \text{vis}) & \text{if } \exists c.it(c,s) \wedge \neg md(c, \text{vis}, s) \\ it\_get & \text{if } \exists c, v, k, v'.it(c,s) \wedge \\ & \quad node(c,v,k,s) \wedge md(c, \text{vis}, s) \wedge \\ & \quad min\_register(v', s) \wedge v < v' \\ it\_next & \text{if } \exists c, v, k, v'.it(c,s) \wedge \\ & \quad node(c,v,k,s) \wedge k \neq \text{END} \wedge \\ & \quad min\_register(v', s) \wedge v \geq v' \\ no\_op & \text{otherwise} \end{cases}$$

Let M_h be a model of $\mathcal{D}_h^{sll}$, $Goal$ the conjunction of LTL goal constraints at the HL and $Cstr$ the conjunction of LTL trace constraints at the HL. Then we can use Theorem 16 to show that:

$$M_h \models AgtCanForceByIf(Goal, Cstr, f_h, S_0)$$
$$\text{and } \mathcal{D}_l \models AgtCanForceBy(m(Goal), \textit{True}, f_l, S_0)$$

6 Program synthesis tasks

In this section, we consider two simple tasks to illustrate how we can use our framework and LTL synthesis to solve various kinds of program synthesis problems.

Tree chopping problem. As a first example of program synthesis, we consider the tree chopping problem described in (De Giacomo et al. 2003). An agent wants to cut down a tree and has a primitive action (*chop*) to chop at the tree. If the tree is repeatedly chopped, it will eventually come down and the agent can always see if this is the case, but it does not know a priori how many times it must be chopped to come down. It is possible to model this example using the HL NDBAT $\mathcal{D}_h^{sll}$ for singly-linked lists of the previous example and some of the LTL constraints already discussed. The notion that the tree will eventually come down after an indeterminate number of cuts can be mapped into the trace constraint for which the end of the list is eventually reached after an indeterminate number of cursor moves (i.e. $(\Box\Diamond doneNext) \rightarrow \Diamond\neg hasNext$), and the goal of cutting down the tree can be mapped into the goal of reaching the end of the list (i.e. $\Diamond\Box\neg hasNext$). Thus, at the abstract level, the problem can be reduced to the task of traversing a singly-linked list with an unknown number of nodes, where $next$ represents the action *chop* and $hasNext$ represents the observation about the state of the tree. The strategy for this task is $f_h(s) = next$ if $hasNext(s)$ and $f_h(s) = stop$ otherwise. A controller policy derived through LTL synthesis for the task of traversing a list is provided by (B20).

Swamp crossing problem. We can also handle problems that involve non-linear data structures such as graphs. This is the case for the swamp crossing problem, discussed also in (B20). A swamp is a matrix containing either water or land in its cells and the task is to check if there is a path of adjacent land cells from a leftmost cell to a rightmost cell. From this specification, we can model the swamp as an undirected graph where the nodes are the land cells and the edges connect two adjacent land cells, and the task becomes traversing the graph and looking for a node that is (one of) the rightmost.

At the concrete level, a graph has fluents for representing its nodes and edges, $vertex(vertex\_id, label, situation)$ and $edge(edge\_id, out\_v, in\_v, situation)$, in addition to another fluent used to denote what is the node we are visiting in a given situation, $visiting(id, situation)$. We also consider the use of a memory with finitely many registers that we represent using the formalization provided for the singly-linked lists with $memory(head, situation)$ and $register(id, next\_id, situation)$ replacing sll and $node$ respectively. Hence, we introduce also the actions for reading the first element of the memory, $read\_memory$, and for writing a new element at the end, $write\_memory(element)$. Finally, as in the list example, we have the fluent $marked$ and the actions $mark$, used to correctly traversing the graph without visiting the same node multiple times, and the action no_op.

At the abstract level, we use the actions $extractNode$, to extract a node from the memory, $visitNode$, to inspect (and perform some computation on) the node just extracted, and $putNeighbors$, to put the neighbors of the current node into memory. The environment reaction of $extractNode$ gives information about whether the extracted node was the last one in memory, while the environment reaction of $visitNode$ tells us whether the visiting node is the rightmost and whether it has neighbors (we'll have four constants to cover the possible cases). We also use the fluents $emptyMemory$, which indicates whether the memory is empty, $hasNeighbors$, which indicates whether the current node has neighbors to be put into memory, and $rightmost$, which indicates whether the current node is rightmost.

At the HL. the action precondition axioms and the successor state axioms in $\mathcal{D}_h^{graph}$ are:

$$Poss_{ag}(extractNode, s) \doteq \neg emptyMemory(s)$$
$$Poss_{ag}(visitNode, s) \doteq True$$
$$Poss_{ag}(putNeighbors, s) \doteq hasNeighbors(s)$$
$$Poss_{ag}(stop, s) \doteq True$$
$$Poss(extractNode(r), s) \equiv$$
$$\quad Poss_{ag}(extractNode, s) \wedge (r = Last \vee r = NotLast)$$
$$Poss(visitNode(r), s) \equiv Poss_{ag}(visitNode, s) \wedge$$
$$\quad (r = \text{R\_HN} \vee r = \text{R\_NHN} \vee r = \text{NR\_HN} \vee r = \text{NR\_NHN})$$
$$Poss(putNeighbors(r), s) \equiv$$
$$\quad Poss_{ag}(putNeighbors, s) \wedge r = Success_{PN}$$
$$Poss(stop(r), s) \equiv Poss_{ag}(stop, s) \wedge r = Success_{HS}$$

$$emptyMemory(do(a, s)) \equiv$$
$$\quad a = extractNode(Last) \vee$$
$$\quad emptyMemory(s) \wedge a \neq extractNode(NotLast) \wedge$$
$$\quad\quad a \neq putNeighbors(Success_{PN})$$

$hasNeighbors(do(a,s)) \equiv$
$a = visitNode(\text{R\_HN}) \vee a = visitNode(\text{NR\_HN}) \vee$
$hasNeighbors(s) \wedge a \neq visitNode(\text{R\_NHN}) \wedge$
$a \neq visitNode(\text{NR\_NHN}) \wedge$
$a \neq putNeighbors(Success_{PN})$
$rightmost(do(a,s)) \equiv$
$a = visitNode(\text{R\_HN}) \vee a = visitNode(\text{R\_NHN}) \vee$
$rightmost(s) \wedge a \neq visitNode(\text{NR\_HN}) \wedge$
$a \neq visitNode(\text{NR\_NHN})$

Then we must specify the refinement mapping m^{graph} by which we describe the relationship between the HL and LL:

$m_a(extractNode) = read\_memory$
$m_a(visitNode) = (\pi c).[visiting(c)?; mark(c, \text{visited})]$
$m_a(putNeighbors) =$
while $\exists c, d, c'.visiting(c) \wedge edge(d,c,c') \wedge$
$\neg(marked(c', \text{visited}) \vee \exists k.register(c',k))$ **do**
$(\pi c').[(\exists c, d.visiting(c) \wedge edge(d,c,c') \wedge$
$\neg(marked(c', \text{visited}) \vee \exists k.register(c',k)))?;$
$write\_memory(c')]$ **endWhile**
no_op
$m_a(stop) = no\_op$

$m_s(extractNode(r_h)) =$
if $\exists h, k.memory(h) \wedge register(h, \text{END})$
then $r_h = \text{Last}$? **else** $r_h = \text{NotLast}$?**endIf**
$read\_memory(Success_{RM})$
$m_s(visitNode(r_h)) =$
$(\pi c).[visiting(c)?; mark(c, \text{visited}, Success_{MD})];$
if $\exists c, d, c'.visiting(c) \wedge edge(d,c,c') \wedge$
$\neg(marked(c', \text{visited}) \vee \exists k.register(c',k))$ **then**
if $\exists c.visiting(c) \wedge vertex(c, \text{rightmost})$
then $r_h = \text{R\_HN}$? **else** $r_h = \text{NR\_HN}$? **endIf** **else**
if $\exists c.visiting(c) \wedge vertex(c, \text{rightmost})$
then $r_h = \text{R\_NHN}$? **else** $r_h = \text{NR\_NHN}$? **endIf**
endIf
$m_s(putNeighbors(r_h)) =$
while $\exists c, d, c'.visiting(c) \wedge edge(d,c,c') \wedge$
$\neg(marked(c', \text{visited}) \vee \exists k.register(c',k))$ **do**
$(\pi c').[(\exists c, d.visiting(c) \wedge edge(d,c,c') \wedge$
$\neg(marked(c', \text{visited}) \vee \exists k.register(c',k)))?;$
$write\_memory(c', Success_{WM})]$ **endWhile**
$r_h = Success_{PN}$
$m_s(stop(r_h)) = no\_op(Success_{NO}); r_h = Success_S$

$m_f(emptyMemory) = \neg\exists h, k.register(h,k)$
$m_f(hasNeighbors) = \exists c, d.visiting(c) \wedge$
$marked(c, \text{visited}) \wedge edge(d,c,c') \wedge$
$\neg(marked(c', \text{visited}) \vee \exists k.register(c',k))$
$m_f(rightmost) = \exists c.visiting(c) \wedge$
$marked(c, \text{visited}) \wedge vertex(c, \text{rightmost})$

The most interesting action is $putNeighbors$, which consists of repeatedly checking if there exists a neighbor of the visiting node that is not in memory and has not been visited, and adding it into memory, until no more are left.

Now, we have to specify the LTL trace constraints and the goals for our task. First, we want to state that if we keep extracting nodes from memory, it eventually becomes empty, written $\Box\Diamond doneExtract \rightarrow \Diamond(emptyMemory \wedge \neg hasNeighbors)$. Then, since the objective is to visit all reachable land nodes once until one of the rightmost land nodes (if any) is reached, we write the goal as follows: $\Diamond\Box rightmost \vee \Diamond\Box(emtyMemory \wedge \neg hasNeighbors)$. Additionally, we say that the visit of a node always follows the extraction of that node from memory, i.e., $\Box(doneExtract \rightarrow \bigcirc doneVisit)$; we don't extract a new node until the neighbors of the one we are visiting have been put into memory, i.e., $\Box((hasNeighbors \wedge \neg rightmost) \rightarrow (\neg extractNode\, \mathcal{U}\, putNeighbors))$; we stop when a rightmost node is found, i.e., $\Box(rightmost \rightarrow \bigcirc doneStop)$.

A HL strategy that guarantees satisfying the goals is:

$$f_h(s) = \begin{cases} extractNode & \text{if } \neg emptyMemory(s) \wedge \\ & \quad \neg hasNeighbors(s) \\ visitNode & \text{if } doneExtract(s) \\ putNeighbors & \text{if } hasNeighbors(s) \\ stop & rightmost(s) \end{cases}$$

7 Discussion

The framework we proposed is inspired by (B20) but there are some significant differences. They assume that every planning instance in a generalized planning problem is a finite-state deterministic classical planning problem and that the set of actions at the abstract and concrete levels are the same (each instance could be specified in PDDL). But how the set of planning instances in a generalized planning problem is specified and how one proves that the temporal assumptions and goal constraints are sound is left open. As mentioned earlier, they give several examples of LTL observation abstractions that can be used to obtain controllers for program synthesis tasks involving data structures, but they don't specify the associated generalized planning problems formally and they don't prove that the stated temporal assumptions and goal constraints are sound. In our situation calculus-based framework, the generalized planning problem is specified formally by a basic action theory and the planning instances are models of this theory, which need not be finite-state and can refer to data. The abstract actions can be implemented by programs at the concrete level. One can use situation calculus reasoning techniques to show that the LTL trace assumptions and goals in our temporally lifted abstractions are satisfied.

The work of (Cui, Liu, and Luo 2021; Cui, Kuang, and Liu 2023) on using abstractions in generalized planning, which builds on (BDL17), is also closely related to ours. But their account seems more complex than ours. They don't use the nondeterministic situation calculus. Their notion of sound abstraction for generalized planning problems requires both m-simulation and m-back-simulation between models of the theories. Their notion of strong solution also only guarantees achieving the goal if the plan does not block.

Our approach to program synthesis requires showing that one has a suitable temporally lifted abstraction and we have developed some techniques for doing this (see appendix). One could develop a library of verified temporally lifted abstractions associated with various data structures. These could then be extended/customized to solve specific problem instances.

Acknowledgements

This work has been supported by PNRR MUR project PE0000013-FAIR.

References

Abadi, M.; Lamport, L.; and Wolper, P. 1989. Realizable and unrealizable specifications of reactive systems. In *ICALP*, volume 372 of *Lecture Notes in Computer Science*, 1–17. Springer.

Aminof, B.; De Giacomo, G.; Murano, A.; and Rubin, S. 2019. Planning under LTL environment specifications. In *ICAPS*, 31–39.

Banihashemi, B.; De Giacomo, G.; and Lespérance, Y. 2017. Abstraction in situation calculus action theories. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, 1048–1055. AAAI Press.

Banihashemi, B.; De Giacomo, G.; and Lespérance, Y. 2023. Abstraction of nondeterministic situation calculus action theories. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, 3112–3122.

Belle, V., and Levesque, H. J. 2016. Foundations for generalized planning in unbounded stochastic domains. In *KR*, 380–389.

Bonet, B.; De Giacomo, G.; Geffner, H.; and Rubin, S. 2017. Generalized planning: non-deterministic abstractions and trajectory constraints. In *IJCAI*, 873–879.

Bonet, B.; De Giacomo, G.; Geffner, H.; Patrizi, F.; and Rubin, S. 2020. High-level programming via generalized planning and ltl synthesis. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 17, 152–161.

Church, A. 1963. Logic, arithmetics, and automata. In *Proc. Int. Congress of Mathematicians, 1962*.

Cui, Z.; Kuang, W.; and Liu, Y. 2023. Automatic verification for soundness of bounded qnp abstractions for generalized planning. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, 3149–3157.

Cui, Z.; Liu, Y.; and Luo, K. 2021. A uniform abstraction framework for generalized planning. In *IJCAI*, 1837–1844.

De Giacomo, G., and Lespérance, Y. 2021. The nondeterministic situation calculus. In Bienvenu, M.; Lakemeyer, G.; and Erdem, E., eds., *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, 216–226.

De Giacomo, G.; Lespérance, Y.; Levesque, H. J.; and Sardina, S. 2003. On deliberation under incomplete information and the inadequacy of entailment and consistency-based formalizations. In *Proceedings of the First Programming Multiagent Systems Languages, Frameworks, Techniques and Tools Workshop (PROMAS-03)*.

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

De Giacomo, G.; Lespérance, Y.; and Muise, C. J. 2012. On supervising agents in situation-determined ConGolog. In *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012, Valencia, Spain, June 4-8, 2012 (3 Volumes)*, 1031–1038. IFAAMAS.

De Giacomo, G.; Lespérance, Y.; and Pearce, A. R. 2010. Situation calculus based programs for representing and reasoning about game structures. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press.

Green, C. C. 1969. Application of theorem proving to problem solving. In *IJCAI*, 219–240.

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI*, 918–923.

Khan, S. M., and Lespérance, Y. 2016. Infinite paths in the situation calculus: Axiomatization and properties. In Baral, C.; Delgrande, J. P.; and Wolter, F., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, 565–568.

Lespérance, Y.; Giacomo, G. D.; Rostamigiv, M.; and Khan, S. M. 2024. Abstraction of situation calculus concurrent game structures. In *Proceedings of the AAAI Conference on Artificial Intelligence, AAAI 2024*, 10624–10634.

McCarthy, J., and Hayes, P. J. 1969. Some Philosophical Problems From the Standpoint of Artificial Intelligence. *Machine Intelligence* 4:463–502.

Meyer, P. J.; Sickert, S.; and Luttenberger, M. 2018. Strix: Explicit reactive synthesis strikes back! In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, 578–586. Springer.

Milner, R. 1971. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, 481–489. William Kaufmann.

Milner, R. 1989. *Communication and concurrency*. PHI Series in computer science. Prentice Hall.

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

Pnueli, A. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 46–57. ieee.

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

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *AAAI*, 991–997.

Waldinger, R. J., and Lee, R. C. T. 1969. PROW: A step toward automatic program writing. In *IJCAI*, 241–252.

A Proofs

Lemma 18. *Suppose that $M_h \models \mathcal{D}^h$ for some high-level theory $\mathcal{D}^h$ and $M_l \models \mathcal{D}^l \cup \mathcal{C}$ for some low-level theory $\mathcal{D}^l$ and m is a mapping between the two theories. Then if*

(a) $S_0^{M_h} \simeq_m^{M_h,M_l} S_0^{M_l}$

(b) *for all high-level action sequences $\vec{\alpha}$,*

$$M_l \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}.(\exists s' Do(m_s(A_i(\vec{x})), s, s')) \supset m_f(\phi_{A_i}^{Poss}(\vec{x}))[s]$$

(c) *for all high-level action sequences $\vec{\alpha}$,*

$$M_l \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'.(Do(m_s(A_i(\vec{x})), s, s') \supset \bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m_f(\phi_{F_i,A_i}^{ssa}(\vec{y}, \vec{x}))[s] \equiv m_f(F_i(\vec{y}))[s']))$$

then $M_h \sim_m^{\leftarrow} M_l$,
where $\phi_{A_i}^{Poss}(\vec{x})$ is the right hand side of the precondition axiom for action $A_i(\vec{x})$, and $\phi_{F_i,A_i}^{ssa}(\vec{y}, \vec{x})$ is the right hand side of the successor state axiom for F_i instantiated with action $A_i(\vec{x})$ where action terms have been eliminated using D_{ca}^h.

Proof. Assume that the antecedent. Let us show that $M_h \sim_m^{\leftarrow} M_l$. Let R be the relation over $\Delta_S^{M_h} \times \Delta_S^{M_l}$ such that

$\langle s_h, s_l \rangle \in R$
if and only if
there exists a ground high-level action sequence $\vec{\alpha}$
such that $M_l, v[s/s_l] \models Do(m(\vec{\alpha}), S_0, s)$
and $s_h = do(\vec{\alpha}, S_0)^{M_h}$.

Let us show that R is an m-simulation relation between M_h and M_l. We need to show that if $\langle s_h, s_l \rangle \in R$, then it satisfies the two conditions in the definition of m-simulation. We prove this by induction n, the number of actions in s_h.
Base case ($n = 0$): By (a), we have that $S_0^{M_h} \simeq_m^{M_h,M_l} S_0^{M_l}$, so condition 1 holds. Take an arbitrary HL action $A(\vec{x})$ and suppose that $M_l, v[s/s_l] \models \exists s'.Do(m(A(\vec{x})), s, s')$. Then by (b), we have that $M_l, v[s/s_l] \models m(\phi_A^{Poss}(\vec{x}))[s]$. By Lemma 10, it follows that $M_h, v[s/s_h] \models \phi_A^{Poss}(\vec{x})[s]$. Thus, by the action precondition axiom for action $A(\vec{x})$, $M_h, v[s/s_h] \models Poss(A(\vec{x}), s)$. By the definition of relation R, $\langle do([\vec{\alpha}, A(\vec{x})], S_0)^{M_h,v}, s_l' \rangle \in R$ if and only if $M_l, v[s/s_l, s'/s_l'] \models Do(m(A(\vec{x})), s_l, s_l')$ (note that we have standard names and domain closure for objects and actions, so we can always ground $\vec{x}$). Thus condition (2) holds for $\langle s_h, s_l \rangle$.
Induction step: Assume that if $\langle s_h, s_l \rangle \in R$ and the number of actions in s_h is no greater than n, then $\langle s_h, s_l \rangle$ satisfies the three conditions in the definition of m-simulation. We have to show that this must also hold for any $\langle s_h, s_l \rangle \in R$ where s_h contains $n + 1$ actions. First we show that condition 1 in the definition of m-simulation holds. If $\langle s_h, s_l \rangle \in R$ and s_h contains $n + 1$ actions, then due to the way R is defined, there exists a ground high level action sequence $\vec{\alpha}$ of length n and a ground high level action $A(\vec{c})$ such that $s_h = do(A(\vec{c}), do(\vec{\alpha}, S_0))^{M_h}$, $s_h' = do(\vec{\alpha}, S_0)^{M_h}$, $M_l, v[s/s_l'] \models Do(m(\vec{\alpha}), S_0, s)$, and $\langle s_h', s_l' \rangle \in R$. s_h' contains n actions so by the induction hypothesis, $\langle s_h', s_l' \rangle$ satisfies the three conditions in the definition of m-simulation, in particular $s_h' \sim_m^{M_h,M_l} s_l'$. By Lemma 10, it follows that $M_h, v[s/s_h'] \models \phi_{F,A}^{ssa}(\vec{y}, \vec{c})[s]$ if and only if $M_l, v[s/s_l'] \models m(\phi_{F,A}^{ssa}(\vec{y}, \vec{c}))[s]$ for any high-level fluent $F \in \mathcal{F}_h$. Thus by the successor state axiom for F, $M_h, v[s/s_h'] \models F(\vec{y}, do(A(\vec{c}), s))$ if and only if $M_l, v[s/s_l'] \models m(\phi_{F,A}^{ssa}(\vec{y}, \vec{c}))[s]$. By condition (c), we have that $M_l, v[s/s_l'] \models m(\phi_{F,A}^{ssa}(\vec{y}, \vec{c}))[s]$ if and only if $M_l, v[s/s_l] \models m(F(\vec{y}))[s]$. Thus $M_h, v[s/s_h'] \models F(\vec{y}, do(A(\vec{c}), s))$ if and only if $M_l, v[s/s_l] \models m(F(\vec{y}))[s]$. Therefore, $s_h \sim_m^{M_h,M_l} s_l$, i.e., condition 1 in the definition of m-simulation holds.
We can show that $\langle s_h, s_l \rangle$, where s_h contains $n + 1$ actions, satisfies condition 2 in the definition of m-simulation, by exactly the same argument as in the base case. □

Theorem 19. *Suppose that we have a HL NDBAT $\mathcal{D}_h$, a model M_h of $\mathcal{D}_h$, a set of HL LTL state constraints Ψ, a LL NDBAT $\mathcal{D}_l$, and a refinement mapping m. If*

(a) *$\mathcal{D}_{S_0}^h$ is a complete theory, i.e. the initial state is completely specified,*

(b) *$\mathcal{D}_{S_0}^l \cup \mathcal{D}_{ca}^l \cup \mathcal{D}_{coa}^l \models m(\phi)$, for all $\phi \in D_{S_0}^h$,*

(c) *for all high-level action sequences $\vec{\alpha}$,*

$$\mathcal{D}^l \cup \mathcal{C} \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}.(\exists s' Do(m_s(A_i(\vec{x})), s, s')) \supset m_f(\phi_{A_i}^{Poss}(\vec{x}))[s]$$

where $\phi_{A_i}^{Poss}(\vec{x})$ is the right hand side (RHS) of the precondition axiom for action $A_i(\vec{x})$,

(d) *for all high-level action sequences $\vec{\alpha}$,*

$$\mathcal{D}^l \cup \mathcal{C} \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'.(Do(m_s(A_i(\vec{x})), s, s') \supset \bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m_f(\phi_{F_i,A_i}^{ssa}(\vec{y}, \vec{x}))[s] \equiv m_f(F_i(\vec{y}))[s']))$$

where $\phi_{F_i,A_i}^{ssa}(\vec{y}, \vec{x})$ is the RHS of the successor state axiom for F_i instantiated with action $A_i(\vec{x})$ where action terms have been eliminated using $\mathcal{D}_{ca}^h$, and

(e) *for every high-level LTL trace constraint $\psi \in \Psi$, $M_h \models \exists p_h.Starts(p_h, S_{0_h}) \wedge Holds(\psi, p_h)$ and $D_l \models \forall p_l.Starts(p_l, S_{0_l}) \supset Holds(m_t(\psi), p_l)$,*

then $(\mathcal{D}_h, M_h, \Psi)$ is a temporally lifted abstraction of $\mathcal{D}_l$ wrt m.

Proof. (a) and (b) imply that for every model M_l of $\mathcal{D}_l$, $S_0^{M_h} \simeq_m^{M_h,M_l} S_0^{M_l}$. Together with (c) and (d), this implies by Lemma 18 that for every model M_l of $\mathcal{D}_l$, $M_h \sim_m^{\leftarrow} M_l$. The result then follows by (e) and the definition of temporally lifted abstraction. □

Proposition 8. NDBAT refinement mapping m^{sll} is proper wrt $\mathcal{D}_l^{sll}$.
Proof. For the HL action $checkValue$, we need to show that

$$\mathcal{D}_l \cup \mathcal{C} \models \forall s.(Do(m_s(\vec{\alpha}), S_0, s) \supset \forall s'.(Do_{ag}(m_a(checkValue), s, s') \equiv \exists r_h.Do(m_s(checkValue(r_h)), s, s')))$$

($\Rightarrow$)

Let M_l and v such that $M_l \models \mathcal{D}_l \cup \mathcal{C}$ and $M_l, v \models Do(m_s(\vec{\alpha}), S_0, s) \wedge Do_{ag}(m_a(checkValue), s, s')$. Suppose that $M_l, v \models Do(m_s(\vec{\alpha}), S_0, s) \wedge \neg\exists c.iterator(c, s)$. In this case, we have that there exists s_i such that $M_l, v[s''/s_i] \models Do_{ag}(it\_set, s, s'')$. By definition of Do_{ag}, we have that $M_l, v[s''/s_i] \models \exists r_l.Poss(it\_set(r_l), s) \wedge r_l = Success_{IS}$. Now we know there exists s_j such that $M_l, v[s''/s_i, s'''/s_j] \models \exists c.iterator(c, s'') \wedge Do_{ag}(mark(c, \text{visited}), s'', s''')$, and again we have $M_l, v[s''/s_i, s'''/s_j] \models \exists c, r_l'.iterator(c, s'') \wedge Poss(mark(c, \text{visited}, r_l'), s'') \wedge r_l' = Success_{MD}$.
We distinguish two cases:

- Suppose that

$$M_l, v[s''/s_i, s'''/s_j] \models \exists c, v, k, v'.iterator(c, s''') \wedge node(c, v, k, s''') \wedge min\_register(v', s''') \wedge v < v'$$

Then we have

$$\begin{aligned} &M_l, v[s''/s_i, s'''/s_j] \models \\ &\exists \delta', \delta'', r_l, r_l', r_h.Trans(m_a(checkValue), s, \delta', s'') \wedge \\ &s'' = do(it\_set(r_l), s) \wedge r_l = Success_{IS} \wedge \\ &Trans(\delta', s'', \delta'', s''') \wedge \\ &s''' = do(mark(c, \text{visited}, r_l'), s'') \wedge \\ &r_l' = Success_{MD} \wedge r_h = LT \wedge \\ &Final(\delta'', s''') \wedge s''' = s' \end{aligned}$$

Hence, $M_l, v \models \exists r_h.Do(m_s(checkValue(r_h), s, s')$.

- Suppose that

$$M_l, v[s''/s_i, s'''/s_j] \models \exists c, v, k, v'.iterator(c, s''') \wedge node(c, v, k, s''') \wedge min\_register(v', s''') \wedge v \geq v'$$

Then we have

$$\begin{aligned} &M_l, v[s''/s_i, s'''/s_j] \models \\ &\exists \delta', \delta'', r_l, r_l', r_h.Trans(m_a(checkValue), s, \delta', s'') \wedge \\ &s'' = do(it\_set(r_l), s) \wedge r_l = Success_{IS} \wedge \\ &Trans(\delta', s'', \delta'', s''') \wedge \\ &s''' = do(mark(c, \text{visited}, r_l'), s'') \wedge \\ &r_l' = Success_{MD} \wedge r_h = GEQ \wedge \\ &Final(\delta'', s''') \wedge s''' = s' \end{aligned}$$

Hence, $M_l, v \models \exists r_h.Do(m_s(checkValue(r_h), s, s')$.

Now, suppose that $M_l, v \models Do(m_s(\vec{\alpha}), S_0, s) \wedge \exists c.iterator(c, s)$. Then, we know there exists s_i such that $M_l, v[s''/s_i] \models \exists c.iterator(c, s) \wedge Do_{ag}(mark(c, \text{visited}), s, s'')$ and we have $M_l, v[s''/s_i] \models \exists c, r_l.iterator(c, s) \wedge Poss(mark(c, \text{visited}, r_l), s) \wedge r_l = Success_{MD}$. Again we distinguish the two cases as before and we found that $M_l, v \models \exists r_h.Do(m_s(checkValue(r_h), s, s')$.

A similar proof can be derived for the other actions.

($\Leftarrow$)

Dual proof. □

Proposition 9. Let M_h^{sll} be a model of $\mathcal{D}_h^{sll}$ and Ψ the set of trace constraints. $(\mathcal{D}_h^{sll}, M_h^{sll}, \Psi)$ is a temporally lifted abstraction of $\mathcal{D}_l^{sll}$ wrt m^{sll}.

Proof. We prove that $(\mathcal{D}_h^{sll}, M_h^{sll}, \Psi)$ is a temporally lifted abstraction using theorem 19.

(a) and (b) are trivially satisfied providing a complete initial state for $\mathcal{D}_l^{sll}$ that entails all refinements of the fluents in $\mathcal{D}_h^{sll}$.

(c) For the HL action $checkValue(r_h)$ we need to show that for any sequence of high-level system actions $\vec{\alpha}$:

$$\begin{aligned} &\mathcal{D}_l \cup \mathcal{C} \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \\ &\quad \forall r_h.\exists s'.(Do(m_s(checkValue(r_h)), s, s') \supset \\ &\quad m_f(r_h = LT \vee r_h = GEQ)) \end{aligned}$$

that is

$$\begin{aligned} &\mathcal{D}_l \cup \mathcal{C} \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \\ &\quad \forall r_h.\exists s'.(Do(\\ &\quad\quad \textbf{if } \neg\exists c.iterator(c) \\ &\quad\quad\quad \textbf{then } it\_set(Success_{IC}) \textbf{ else } nil \\ &\quad\quad \textbf{endIf} \\ &\quad\quad (\pi c).[iterator(c)?; mark(c, \text{visited}, Success_{MD})]; \\ &\quad\quad \textbf{if } \exists c, v, k, v'.iterator(c) \wedge node(c, v, \text{END}) \wedge \\ &\quad\quad\quad min\_register(v') \wedge v < v' \\ &\quad\quad\quad \textbf{then } r_h = \text{LT ? } \textbf{else } r_h = \text{GEQ ?} \\ &\quad\quad \textbf{endIf}, s, s') \supset \\ &\quad (r_h = LT \vee r_h = GEQ)) \end{aligned}$$

It's easy to see that this holds because the agent action has no precondition ($Poss_{ag}(checkValue, s) = \mathit{True}$). Before executing the program **if** $\neg\exists c.iterator(c)$ **then** $it\_set(Success_{IC})$ **else** nil **endIf**, we can have both $\exists c.iterator(c)$ and $\neg\exists c.iterator(c)$. After that execution, that assure us the existence of the iterator, we can perform $(\pi c).[iterator(c)?; mark(c, \text{visited}, Success_{MD})]$ and subsequent conditional statement without any preconditions other than the possible reactions of the environment $(r_h = LT \vee r_h = GEQ)$.

A similar proof can be derived for the other actions.

(d) For the HL action $checkValue(r_h)$ we need to show that for any sequence of high-level system actions $\vec{\alpha}$:

$$\begin{aligned} &\mathcal{D}_l \cup \mathcal{C} \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \\ &\quad \forall r_h, s'.(Do(m_s(checkValue(r_h)), s, s') \supset \\ &\quad\quad \textstyle\bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i, A_i}(\vec{y}, \vec{x}))[s] \equiv m(F_i(\vec{y}))[s'])) \end{aligned}$$

Consider the fluent $lowerThan$ (note that $checkValue$ does not affect $hasNext$). We must show that

$$\begin{aligned} &\mathcal{D}_l \cup \mathcal{C} \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \\ &\quad \forall r_h, s'.(Do(m_s(checkValue(r_h)), s, s') \supset \\ &\quad\quad (m_f(checkValue(r_h) = checkValue(LT) \vee \\ &\quad\quad\quad lowerThan[s] \wedge \\ &\quad\quad\quad checkValue(r_h) \neq checkValue(GEQ) \wedge \\ &\quad\quad\quad \forall r_h'.checkValue(r_h) \neq next(r_h') \wedge \\ &\quad\quad\quad checkValue(r_h) \neq update(Success_{HU})) \equiv \\ &\quad\quad m_f(lowerThan[s'])) \end{aligned}$$

that is

$$\begin{aligned} &\mathcal{D}_l \cup \mathcal{C} \models \forall s.Do(m_s(\vec{\alpha}), S_0, s) \supset \\ &\quad \forall r_h, s'.(Do(m_s(checkValue(r_h)), s, s') \supset \\ &\quad\quad (r_h = LT \vee lowerThan[s] \wedge r_h \neq GEQ) \equiv \\ &\quad\quad (\exists c, v, k.iterator(c, s') \wedge node(c, v, k, s') \wedge \\ &\quad\quad marked(c, \text{visited}, s') \wedge \\ &\quad\quad \exists v'.min\_register(v', s') \wedge v < v')) \end{aligned}$$

This follows from the successor state axioms of $lowerThan$ and refinement of $checkvalue(r_h)$.

A similar proof can be derived for the other actions.

(e) Consider the trace constraint $(\Box\Diamond doneNext) \rightarrow \Diamond\neg hasNext$. Using definition 4, we get the following refinement at the LL:

$$[\Box\Diamond(Hlc \wedge m_f(doneNext))] \rightarrow \Diamond(Hlc \wedge \neg m_f(hasNext))$$

Thus, we want to prove that

$$\begin{array}{l} M_h \models \exists p_h.Starts(p_h, S_{0_h}) \wedge \\ \quad Holds((\Box\Diamond doneNext) \rightarrow \Diamond\neg hasNext, p_h) \text{ and} \\ D_l \models \forall p_l.Starts(p_l, S_{0_l}) \supset \\ \quad Holds([\Box\Diamond(Hlc \wedge m_f(doneNext))] \rightarrow \\ \quad\quad \Diamond(Hlc \wedge \neg m_f(hasNext)), p_l) \end{array}$$

From definition 1, we rewrite the previous statements as follows:

$$\begin{array}{l} M_h \models \exists p_h.Starts(p_h, S_{0_h}) \wedge \\ \quad [(\forall s'.\exists s''.s \preceq s' \preceq s'' \wedge doneNext[s'']) \supset \\ \quad\quad (\exists s'''.s \preceq s''' \wedge \neg hasNext[s'''])] \text{ and} \\ D_l \models \forall p_l.Starts(p_l, S_{0_l}) \supset \\ \quad [(\forall s'.\exists s''.s \preceq s' \preceq s'' \wedge Hlc[s''] \wedge m_f(doneNext)[s'']) \supset \\ \quad\quad (\exists s'''.s \preceq s''' \wedge Hlc[s'''] \wedge \neg m_f(hasNext)[s'''])] \end{array}$$

Let's consider the second statement. At the LL, we have the iterator that points to a certain node, written $\exists c, v, k.iterator(c) \wedge node(c, v, k)$, and every time the agent performs the action it_next, the cursor c is updated with the ID k of the following node. We know that we have finite lists, that is there must exist a node such that $k = END$. This means that, given a path p_l, if there is always a future situation in which action it_next is executed (i.e., it is continuously executed), then there will be a future situation in which $\exists c, v, k.iterator(c) \wedge node(c, v, END)$, i.e. $\neg m_f(hasNext)$ (the iterator point to the last node and there is no successor). At HL level, one progresses in the same way. □

Proposition 17. NDBATs $\mathcal{D}_h^{sll}$ and $\mathcal{D}_l^{sll}$ and mapping m^{sll} satisfy constraint 15.

Proof For the HL action $checkValue$, we need to show that for every high-level system action sequence $\vec{\alpha}$, all executions of the program $m_a^{sll}(checkValue)$ terminate by following strategy f_A:

$$\begin{array}{l} \mathcal{D}_l \models \forall s.Do(m_s(\vec{\alpha}, S_0, s) \supset \\ \quad\quad (\exists s'.Do_{ag}(m_a(checkValue), s, s') \supset \\ \quad\quad\quad AgtCanForceBy(m_a(checkValue), f_A, s)) \end{array}$$

Let M_l and v such that $M_l \models \mathcal{D}_l \cup \mathcal{C}$ and $M_l, v \models Do(m_s(\vec{\alpha}), S_0, s) \wedge Do_{ag}(m_a(checkValue), s, s')$. Suppose that $M_l, v \models Do(m_s(\vec{\alpha}), S_0, s) \wedge \neg\exists c.iterator(c, s)$. In this case, we have that there exists s_i such that $M_l, v[s''/s_i] \models Do_{ag}(f(s), s, s'') \wedge f(s) = it\_set$. By definition of Do_{ag}, we have that $M_l, v[s''/s_i] \models \exists r_l. \wedge Poss(it\_set, s) \wedge s'' = do(it\_set, s) \wedge r_l = Success_{IS}$.

Now we know there exists s_j such that $M_l, v[s''/s_i, s'''/s_j] \models \exists c.iterator(c, s'') \wedge Do_{ag}(f(s''), s'', s''') \wedge f(s'') = mark(c, \text{visited})$, and again we have $M_l, v[s''/s_i, s'''/s_j] \models \exists c, r_l'.iterator(c, s'') \wedge Poss(mark(c, \text{visited}, r_l'), s'') \wedge r_l' = Success_{MD}$.

Finally, we have that

$$\begin{array}{l} M_l, v[s''/s_i, s'''/s_j] \models \\ \quad \exists\delta', \delta'', r_l, r_l'.Trans(m_a(checkValue), s, \delta', s'') \wedge \\ \quad f(s) = it\_set \wedge s'' = do(it\_set(r_l), s) \wedge \\ \quad r_l = Success_{IS} \wedge Trans(\delta', s'', \delta'', s''') \wedge \\ \quad f(s'') = mark(c, \text{visited}) \wedge \\ \quad s''' = do(mark(c, \text{visited}, r_l'), s'') \wedge r_l' = Success_{MD} \wedge \\ \quad f(s''') = stop \wedge Final(\delta'', s''') \wedge s''' = s' \end{array}$$

Now, suppose that $M_l, v \models Do(m_s(\vec{\alpha}), S_0, s) \wedge \exists c.iterator(c, s)$. Then, we know there exists s_i such that $M_l, v[s''/s_i] \models \exists c.iterator(c, s) \wedge Do_{ag}(f(s), s, s'') \wedge f(s) = mark(c, \text{visited})$, and we have $M_l, v[s''/s_i] \models \exists c, r_l.iterator(c, s) \wedge Poss(mark(c, \text{visited}, r_l), s) \wedge r_l = Success_{MD}$. Thus

$$\begin{array}{l} M_l, v[s''/s_i, s'''/s_j] \models \\ \quad \exists\delta', r_l.Trans(m_a(checkValue), s, \delta', s'') \wedge \\ \quad f(s) = mark(c, \text{visited}) \wedge \\ \quad s'' = do(mark(c, \text{visited}, r_l), s) \wedge r_l = Success_{MD} \wedge \\ \quad f(s'') = stop \wedge Final(\delta', s'') \wedge s'' = s' \end{array}$$

A similar proof can be derived for the other actions. □